Lecture Notes in Computer Science 1683
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Jörg Flum Mario Rodriguez-Artalejo (Eds.)

Computer Science Logic

13th International Workshop, CSL'99
8th Annual Conference of the EACSL
Madrid, Spain, September 20-25, 1999
Proceedings

Springer

Volume Editors

Jörg Flum
Universität Freiburg, Mathematisches Institut
Eckerstr. 1, D-79104 Freiburg, Germany
E-mail: flum@ruf.uni-freiburg.de

Mario Rodríguez-Artalejo
Universidad Complutense de Madrid
Dpto. de Sistemas Informáticos y Programación
Edificio Fac. Matemáticas
Av. Complutense s/n, E-28040 Madrid, Spain
E-mail: mario@eucmos.sim.ucm.es

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Computer science logic : 13th international workshop ; proceedings / CSL'99, Madrid, Spain, September 20 - 25, 1999. Jörg Flum ; Mario Rodriguez-Artalejo (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1999
(Annual Conference of the EACSL ... ; 1999)
(Lecture notes in computer science ; Vol. 1683)
ISBN 3-540-66536-6

CR Subject Classification (1998): F.4, I.2.3-4, F.3

ISSN 0302-9743
ISBN 3-540-66536-6 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN 10704371 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

The 1999 Annual Conference of the European Association for Computer Science Logic, CSL'99, was held in Madrid, Spain, on September 20-25, 1999. CSL'99 was the 13th in a series of annual meetings, originally intended as International Workshops on Computer Science Logic, and the 8th to be held as the Annual Conference of the EACSL. The conference was organized by the Computer Science Departments (DSIP and DACYA) at Universidad Complutense in Madrid (UCM).

The CSL'99 program committee selected 34 of 91 submitted papers for presentation at the conference and publication in this proceedings volume. Each submitted paper was refereed by at least two, and in almost all cases, three different referees. The second refereeing round, previously required before a paper was accepted for publication in the proceedings, was dropped following a decision taken by the EACSL membership meeting held during CSL'98 (Brno, Czech Republic, August 25, 1998).

In addition to the contributed papers, the scientific program of CSL'99 included five invited talks (J.L. Balcázar, J. Esparza, M. Grohe, V. Vianu and P.D. Mosses) and two tutorials on "Interactive Theorem Proving Using Type Theory" by D.J. Howe and "Term Rewriting" by A. Middeldorp. Four of the invited speakers have provided papers that have been included in this volume. For the remaining invited speaker, as well as the tutorialists, one-page abstracts have been included. The contents of the invited and contributed papers fall mainly under the following topics: concurrency, descriptive complexity, lambda calculus, linear logic, logic programming, modal and temporal logic, mu calculus, specification, type theory and verification.

We are most grateful to the members of the program committee and all the referees for their work. Finally, we are indebted to all the members of the local organizing committee for their support, which included maintenance of Web pages and assistance to the editing work needed to prepare this proceedings volume according to Springer's instructions.

July 1999 Jörg Flum and Mario-Rodríguez-Artalejo

Sponsors

Comisión Interministerial de Ciencia y Tecnología (CICYT) - MEC
Departamento Arquitectura de Computadores y Automática (DACYA) - UCM
Departamento Sistemas Informáticos y Programación (DSIP) - UCM
Escuela Superior de Informática - UCM
Esprit Working Group EP 22457 (CCL II)
European Research Consortium for Informatics and Mathematics (ERCIM)
Facultad de Matemáticas - UCM
Vicerrectorado de Investigación - UCM
Vicerrectorado de Relaciones Internacionales - UCM

Organizing Committee

J.C. González-Moreno
T. Hortalá-González
J. Leach-Albert (**Chair**)
F.J. López-Fraguas
F. Sáenz-Pérez
E. Ullán-Hernández

Program Committee

S. Abramsky (UK)
M. Bezem (The Netherlands)
P. Clote (Germany)
H. Comon (France)
J. Flum (Germany) (**Co-Chair**)
H. Ganzinger (Germany)
N. Immerman (USA)
N. Jones (Denmark)
J. Maluszynski (Sweden)
M. Maher (Australia)
C. Palamidessi (USA)
M. Rodríguez-Artalejo (Spain) (**Co-Chair**)
W. Thomas (Germany)
J. Tiuryn (Poland)
M. Wirsing (Germany)

Referees

J.M. Almendros-Jiménez
R. Alur
G. Antoniou
P. Arenas-Sánchez
A. Arnold
F. Baader
D. Basin
H. Baumeister
M. Benke
S. Berardi
U. Berger
R. Berghammer
I. Bethke
P. Blackburn
A. Blass
A. Bove
J. Bradfield
K. Bruce
W. Buchholz
S. Buss
I. Cervesato
W. Charatonik
H. Cirstea
J. Dix
V. Danos
S. Debray
G. Delzanno
G. Dowek
W. Drabent
D. Driankov
H. Dubois
P.M. Dun
H.-D. Ebbinghaus
J.E. Santo
R. Fagin
L. Fribourg
T. Fruehwirth
F. Gadducci
A. Gavilanes
A. Gil-Luezas
M. Grohe
K. Grue
S. Grumbach
J. Harland
J. Hatcliff
N. Heintze
D. Hendriks
T. Henzinger
C. Hermida
W. van der Hoek
M. Hofmann
F. Honsell
T. Hortalá-González
R. Jagadeesan
D. Jeffery
P. Johannsen
M. Jurdziński
J. Jurjens
J.-P. Katoen
B. Koenig
P.G. Kolaitis
P. Kosiuczenko
J. Krajicek
F. Kröger
G. Kucherov
W. Küchlin
A. Kurz
Y. Lafont
J. Laird
P. Lambrix
K. Laufer
C. Lautemann
J. Leach-Albert
M. Lenisa
M. Leuschel
J. Levy
L. Liquori
L.F. Llana-Díaz
A. Middeldorp
D. Miller
E. Moggi
B. Nebel

F. Neven
R. Nieuwenhuis
S. Nieva
U. Nilsson
T. Nipkow
H. de Nivelle
D. Niwiński
A. Nonnengart
V. van Oostrom
F. Orejas
M. Otto
L. Pacholski
E. Pimentel
T. Pitassi
A. Pnueli
R. Pollack
F. van Raamsdonk
H. Reichel
B. Reus
P. Rodenburg
E. Sandewallv
P. Schnoebelen
M. Schorlemmer
A. Schubert
T. Schwentick
A. Simpson
S. Skalberg
V. Sofronie-Stokkermans
H. Sondergaard
M.H. Sørensen
I. Stark
T. Strahm
T. Streicher
J. Stuber
D. Therien
R. Topor
J. Torán
A. Urquhar
P. Urzyczyn
M. Vardi
M. Veanes
Y. Venema
A. Visser
I. Walukiewicz
J. Warners
M. Wehr
F. Wiedijk
G. Winskel
W. Yi

Table of Contents

Invited Papers

Topological Queries in Spatial Databases
V. Vianu 1
The Consistency Dimension, Compactness, and Query Learning
J.L. Balcázar 2
Descriptive and Parameterized Complexity
M. Grohe 14
Logical Specification of Operational Semantics
P.D. Mosses 32
Constraint-Based Analysis of Broadcast Protocols
G. Delzanno, J. Esparza and A. Podelski 50

Contributed Papers

Descriptive Complexity, I

Descriptive Complexity Theory for Constraint Databases
E. Grädel and S. Kreutzer 67
Applicative Control and Computational Complexity
D. Leivant 82

Verification

Applying Rewriting Techniques to the Verification of Erlang Processes
T. Arts and J. Giesl 96
Difference Decision Diagrams
J. Møller, J. Lichtenberg, H.R. Andersen and H. Hulgaard 111
Analysis of Hybrid Systems: An Ounce of Realism Can Save an Infinity of States
M. Fränzle 126
Verifying Liveness by Augmented Abstraction
Y. Kesten and A. Pnueli 141

Temporal Logic

Signed Interval Logic
T.M. Rasmussen 157
Quantitative Temporal Logic
Y. Hirshfeld and A. Rabinovich 172
An Expressively Complete Temporal Logic without Past Tense Operators for Mazurkiewicz Traces
V. Diekert and P. Gastin 188

Lambda Calculus, Linear Logic

Using Fields and Explicit Substitutions to Implement Objects and Functions in a de Bruijn Setting
E. Bonelli 204
Closed Reductions in the λ-Calculus
M. Fernández and I. Mackie 220
Kripke Resource Models of a Dependently-Typed, Bunched λ-Calculus
S. Ishtiaq and D.J. Pym 235
A Linear Logical View of Linear Type Isomorphisms
V. Balat and R. Di Cosmo 250

Logic Programming, Modal Logic, Description Logic

Choice Logic Programs and Nash Equilibria in Strategic Games
M. De Vos and D. Vermeir 266
Resolution Method for Modal Logic with Well-Founded Frames
S. Hagihara and N. Yonezaki 277
A NExpTime-Complete Description Logic Strictly Contained in C^2
S. Tobies 292
A Road-Map on Complexity for Hybrid Logics
C. Areces, P. Blackburn and M. Marx 307

Descriptive Complexity, II

Monadic**NLIN** and Quantifier-Free Reductions
C. Lautemann and B. Weinzinger 322
Directed Reachability: From Ajtai-Fagin to Ehrenfeucht-Fraïssé Games
J. Marcinkowski 338
Fixpoint Alternation and the Game Quantifier
J.C. Bradfield 350

Logic and Complexity

Lower Bounds for Space in Resolution
J. Torán 362
Program Schemes, Arrays, Lindström Quantifiers and Zero-One Laws
I.A. Stewart 374
Open Least Element Principle and Bounded Query Computation
L.D. Blekemishev 389

Lambda Calculus, Type Theory

A Universal Innocent Game Model for the Böhm Tree Lambda Theory
A.D. Ker, H. Nickau and C.-H.L. Ong 405
Anti-Symmetry of Higher-Order Subtyping
A. Compagnoni and H. Goguen 420

Safe Proof Checking in Type Theory with Y
H. Geuvers, E. Poll and J. Zwanenburg 439
Monadic Presentations of Lambda Terms Using Generalized Inductive Types
T. Altenkirch and B. Reus 453

Linear Logic, Mu Calculus, Concurrency

A P-Time Completeness Proof for Light Logics
L. Roversi 469
On Guarding Nested Fixpoints
H. Seidl and A. Neumann 484
A Logical Viewpoint on Process-Algebraic Quotients
A. Kučera and J. Esparza 499
A Truly Concurrent Semantics for a Simple Parallel Programming Language
P. Gastin and M. Mislove 515

Specification, Data Refinement

Specification Refinement with System F
J.E. Hannay 530
Pre-logical Relations
F. Honsell and D. Sannella 546
Data-Refinement for Call-By-Value Programming Languages
Y. Kinoshita and J. Power 562

Tutorials

Term Rewriting
A. Middeldorp 577
Interactive Theorem Proving Using Type Theory
D.J. Howe 578

Author Index 579

Topological Queries in Spatial Databases

Victor Vianu

Univ. of California at San Diego, CSE 0114, La Jolla, CA 92093-0114

Abstract. Handling spatial information is required by many database applications, and each poses different requirements on query languages. In many cases the precise size of the regions is important, while in other applications we may only be interested in the TOPOLOGICAL relationships between regions —intuitively, those that pertain to adjacency and connectivity properties of the regions, and are therefore invariant under homeomorphisms. Such differences in scope and emphasis are crucial, as they affect the data model, the query language, and performance. This talk focuses on queries targeted towards topological information for two-dimensional spatial databases, where regions are specified by polynomial inequalities with integer coefficients. We focus on two main aspects: (i) languages for expressing topological queries, and (ii) the representation of topological information. In regard to (i), we study several languages geared towards topological queries, building upon well-known topological relationships between pairs of planar regions proposed by Egenhofer. In regard to (ii), we show that the topological information in a spatial database can be precisely summarized by a finite relational database which can be viewed as a topological annotation to the raw spatial data. All topological queries can be answered using this annotation, called topological invariant. This yields a potentially more economical evaluation strategy for such queries, since the topological invariant is generally much smaller than the raw data. We examine in detail the problem of translating topological queries against the spatial database into queries against the topological invariant. The languages considered are first-order on the spatial database side, and fixpoint and first-order on the topological invariant side. In particular, it is shown that fixpoint expresses precisely the PTIME queries on topological invariants. This suggests that topological invariants are particularly well-behaved with respect to descriptive complexity. (Based on joint work with C.H.Papadimitriou, D. Suciu and L. Segoufin.)

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, p. 1, 1999.

The Consistency Dimension, Compactness, and Query Learning

José L. Balcázar[1]

[1] Departament LSI, Universitat Politècnica de Catalunya, Campus Nord, 08034 Barcelona, Spain

Abstract. The consistency dimension, in several variants, is a recently introduced parameter useful for the study of polynomial query learning models. It characterizes those representation classes that are learnable in the corresponding models. By selecting an abstract enough concept of representation class, we formalize the intuitions that these dimensions relate to compactness issues, both in Logic and in a specific topological space. Thus, we are lead to the introduction of Quantitative Compactness notions, which simultaneously have a clear topological meaning and still characterize polynomial query learnable representation classes of boolean functions. They might have relevance elsewhere too. Their study is still ongoing, so that this paper is in a sense visionary, and might be flawed.

Compactness is to topology as finiteness is to set theory.

H. Lenstra (cited in [7])

Polynomial versus exponential growth corresponds in some sense to countability versus uncountability.

M. Sipser [8]

1 Introduction

This somewhat nonstandard paper discusses, mostly at an intuitive level, recent and ongoing work of the author and colleagues in a rather unclassificable research area.

The basic connections between Logic and Topology are long well understood, to the extent that the central Compactness Theorems in Logic are widely known through their natural topological name. By restricting ourselves to the very easy propositional case, we want to trascend here these currently known connections by encompassing other mathematically formalized areas of combinatorial nature.

The reason of being of this text is as follows. Recently, an intuition born from considerations similar to compactness in logic led to some recent advances in query learning [1]. However, eventually, these intuitions became more a hindrance than a help, and were dropped from that paper. They became labeled as "to be discussed elsewhere"; namely, here.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 2−13, 1999.

1.1 Topology Versus Computation

The long history of implications of logical concepts on computation-theoretic issues suggests that compactness, having a clear logical meaning, might be relevant too for other applications in computation theory. However, whereas compactness is a qualitative, yes/no property, many mathematical issues of computational flavor are inherently quantitative; thus, interesting intuitions might be gleaned from topological concepts such as compactness, but *quantitative compactness* notions of some sort seem likely to be necessary.

The main contribution of this paper is the proposal of a formalization of a quantitative notion of compactness. Indeed, comparing different compact subspaces of a fixed topological space we might as well find that some of them are "more compact" than others.

Thus, specifically, we propose first a quantitative notion of compactness, the *compactness rate*, and explain that, on a specific, pretty natural topology Ξ, it has a close relationship with the learnability of representation classes of boolean functions through equivalence queries. Then we also discuss some slightly unconvincing aspects of our proposal, we suggest a second one, and explain that, in the same space Ξ, this second approach has a similarly close relationship with learnability from membership queries.

In a final section we hint at wide areas of open questions whose answers, we feel, might provide illustrative intuitions on the combinatorial material handled by these, and other, popular computational learning models.

1.2 Disclaimer

As ongoing work, the materials included here at the time of going to press have undergone no peer review process at all, and even some parts of the less detailed discussion have not been duly formalized yet; thus this text may well be full of mistakes. The home page of the author on the web (`www.lsi.upc.es/~balqui`) will be in the near future (read a few months) a reasonable source for the less unfaithful version of the results available at each moment (if any).

2 Preliminaries

We denote the set of the natural numbers as $\mathbb{N}$, or as ω to emphasize its use as an ordinal. We consider binary words of a length n, when necessary, as indices into an infinite binary word such as a characteristic function of a formal language; or, alternatively, as values for n boolean variables, or attributes, and thus inputs to a boolean n-ary function.

2.1 Topology

We recall briefly some topological concepts. Topological spaces consist of a domain and a family of subsets of this domain, closed under arbitrary unions and finite intersections. The sets in the family are called open sets; the empty subset and the whole space must always be open sets. Frequently only an open basis is given, and open sets are those obtained from the given basis through these operations. The complements of open sets are called closed sets. The topology can be given as well as the family of closed sets, which should be closed under finite unions and arbitrary intersections, or by a closed basis. Subspaces are defined by subsets of the domain, restricting all open or closed sets to their traces on the subspace, i.e. to their intersections with the subset.

Compact (sub)spaces play an important role since they guarantee certain convergence facts for successions and, more generally, for filters. A topological space is compact if the following axiom holds in it: every family of open sets that covers the whole space, in the sense that their union coincides with it, has a finite subfamily that already covers the whole space. This is known as the Borel-Lebesgue axiom (see, however, the remarks about the work of Cousin in [7]), and can be stated equivalently as follows: every family of closed sets with empty intersection has a finite subfamily that already has empty intersection. Two other characterizations of compactness are: every filter has an adherence value, and every ultrafilter converges. (Here we are glossing over some separateness conditions that turn out to be irrelevant for our purposes.) It is not difficult to see that we can restrict our attention to families of sets that are finite unions of basic closed sets.

The discrete topology on any given space is rather trivial: every subset is accepted as an open set. We will obtain from it more sophisticate topologies through the product construction.

A product space is a topological space $X = \prod_{i \in \omega} X_i$, endowed with the standard product topology of the factor topological spaces X_i; that is, open sets of X are products of finitely many proper open sets from the factors, all the other factors being equal to the whole factor spaces X_i; or arbitrary unions thereof. Of course, arbitrary index sets can be used instead of ω to construct product spaces; but we will limit ourselves to product spaces consisting of ω components.

Actually, in such a product topology, we can (and will) consider a closed set basic if only one projection on a factor space X_i is a closed set different from the whole factor X_i, and it is a basic closed set there too. The corresponding dimension will be called the nontrivial axis of the basic set. Clearly, all closed sets are intersections of basic sets.

The following three facts will be important. First, a discrete topological space is compact iff it is finite. Second, a product space is compact iff all its factors are; this is Tychonoff's theorem. Third, in a compact space, the compact subspaces are exactly the closed sets. See [7] for an extremely instructive historical perspective of the original development of the notion of compactness, and for additional references (beyond your own favorites) on the technicalities of the topological materials.

From now on, we will focus on countably compact spaces, where the condition of empty intersection of some finite subfamily only applies to countable families of closed sets; so that ω suffices as index set. I believe that this restriction is not really relevant; it is not, certainly, for the specific space Ξ on which we focus later on.

2.2 Logic

We restrict ourselves to propositional logic in this text.

The language of propositional logic includes propositional variables, which we will consider as atomic; this means that their meanings, or truth values, can vary among exactly the two boolean constants True and False. They get tied together with connectives such as conjunction, disjunction, and negation, making up propositional formulas. We assume a fixed infinite supply of propositional variables $X = \{x_i \mid i \in \mathbb{N}\}$.

Models in propositional logic consist simply on a truth value interpretation for each propositional variable; through the standard rules, this assigns a truth value to each formula. A model satisfies a set of formulas if all of them get value True under that model.

Trusting that the reader will not get confused, we will use the symbols $\{0, 1\}$ to abbreviate the truth values False and True respectively. Then we consider the finite topological space $\{0, 1\}$ endowed with the discrete topology; by the facts enumerated above, it is obviously compact.

Then the space of models is formed by infinite sequences of boolean values, with component i being the truth value of the propositional variable x_i; and, as such, it can be endowed with the corresponding product topology: $\prod_{i\in\omega}\{0, 1\}$, the product space of ω copies of the binary discrete space. By Tychonoff's theorem, it is compact.

In the corresponding product topology, the basic open sets correspond to finitely many restrictions of components to a single bit, leaving all the others free; that is, each basic open set is the set of models of a term. We have actually taken the slightly more restricted basis of open sets in which a single component is restricted to a single bit, i.e. models for literals. Automatically these are as well the basic closed sets.

It is not difficult to see that now the clopen sets (sets that are simultaneously open and closed) are exactly finite unions of finite intersections of these, i.e., the set of models of a DNF-like formula (a boolean polynomial); or, equivalently, finite intersections of finite unions of basic open sets, i.e., the set of models of a CNF-like formula.

Similarly, closed sets are arbitrary intersections of these, thus sets of models of a possibly infinite set of boolean formulas; and, now, families of clopen sets having empty intersection correspond to unsatisfiable sets of formulas, and therefore the statement that the space is compact literally corresponds to the fact that, if every finite subset of a set of formulas is satisfiable, then the whole set is satisfiable. Hence the name Compactness Theorem.

2.3 Computational Learning

Some areas of Computational Learning Theory study models for the learnability properties of boolean functions. Each model provides a framework to present in unified manners several learning algorithms, and, more importantly, allows us to prove negative results by which no algorithm can exist to learn some representation of knowledge within a given learning model. See [6].

Many models take into account resource limitations, such as time or space. We will be working with polynomial query models, where computation resources are not limited but the amount of information provided about the concept to be learned is.

We focus on learning representations of boolean functions, as an extremely basic form of knowledge. A boolean function of arity n is a function from $\{0,1\}^n \mapsto \{0,1\}$; identifying $2 = \{0,1\}$ as usual and identifying 2^A with the power set operator $\mathcal{P}(A)$ also as usual, and, for finite A, with the set of binary characteristic sequences of subsets of A, we see that the set of boolean functions of arity n is 2^{2^n}; each function being defined by a member of the set $2^n = \{0,1\}^n$, that is, a sequence of 2^n bits (its truth table).

There is a large choice of means of representing boolean functions: boolean circuits, boolean formulas, and their CNF-like and DNF-like depth-two subclasses are fundamental ones, but decision trees, branching programs, OBDDs, and even formal language models such as finite automata are popular for diverse applications. Some of them are able to represent functions f from a variable number of binary arguments; then we will mostly consider their restrictions to a fixed number of arguments, n, and we denote such restrictions as $f|_n$.

Generally, representation classes are frequently defined as tuples $\langle R, \rho, |.| \rangle$ where $R \subseteq \{0,1\}^*$ is a formal language whose strings are considered as syntactically correct descriptions of some computing device; $\rho : R \mapsto \mathcal{P}(\{0,1\}^*)$ is the semantic function that indicates what is the boolean function or formal language $\rho(c)$ described by the device $c \in R$; and $|.| : R \mapsto \mathbb{N}$ measures the size $|c|$ of each description $c \in R$. Frequently $|c|$ is simply its length as a string. Sometimes even the syntactic and semantic alphabets are allowed to vary and get therefore included in the tuples forming the representation classes.

"Honesty" conditions are usually imposed to ensure that it is reasonably feasible to decide whether a given string belongs to the concept described by a given description, as well as computing the size of a description.

In essence, in our polyquery learning models, a hidden concept (the boolean function or formal language $\rho(c)$) has to be identified, by finding, through some sort of interaction, an alternative, not too large, representation c' for the concept: $\rho(c') = \rho(c)$. We will be interested in the interaction, or quantity of information available, as a computational resource; thus, we will mostly ignore up to polynomial space computations. It turns out that this implies that most of the information provided by the representation class is unimportant. The only central issue is that a representation class provides a "size" for each boolean function f of any fixed arity n: the size is $|c|$, where c is a smallest (with respect to $|.|$) description of f: that is, $\rho(c)$ behaves as f on $\{0,1\}^n$, or $\rho(c)|_n = f|_n$.

Thus, for us, a representation class will be simply a size function $R = \bigcup_{n\in\omega} R_n$ for boolean functions: $R_n : 2^{2^n} \mapsto \mathbb{N}$. For $f \in 2^{2^n}$, we frequently abbreviate $R_n(f)$ as $|f|_R$, the size of f as measured by the representation class R. We can allow f to reach some $\infty \notin \mathbb{N}$ value in case we want to deal with representations that cannot describe all boolean functions; this issue is not relevant to our work here, and simply one has to adjust the universe of allowed concepts accordingly.

In the equivalence queries model, a learner interacts with a teacher as follows: in each round, the learner queries about a representation from the class (in our case, this only means that the learner will pay the size of its hypothesis, measured according to the size function defining the class). The teacher either answers YES, if the hypothesis represents exactly the target concept, or gives a counterexample otherwise. We assume that all our learning algorithms are provided with the number n of attributes of the function to be learned and with a bound m on the size of the target function under the chosen representation.

Polynomial query means that the number of queries and the size of each query have to be polynomially bounded in n and m; we ignore how difficult is it, computationally, to find the representation the learner wants to query; however, it can be seen that PSPACE suffices [5]. In the membership query model, the learner simply asks the teacher to evaluate the target concept on a given binary word, and gets a binary answer. The combined model allows both sorts of queries.

We will impose a final technical restriction, which makes our results weaker than they seem. We assume that our learning algorithms do *not* query equivalence queries larger than the bound m provided. We call these algorithms *mean*. This is a strong restriction since some learning algorithms do not obey it, mostly for other models such as equivalence and membership queries; but many of the most relevant learning algorithms with only equivalence queries do actually obey this restriction.

Note finally that this restriction is not so for membership queries, where the only role the representation class plays is to provide the initial bound on the size of the concept.

3 Compactness: A Quantitative Approach

For most of the paper, we assume that the topological space X under consideration is a product space of ω spaces, $X = \prod_{i\in\omega} X_i$, and we assume as fixed a family of basic closed sets on each factor X_i. Then we can select, and fix from now on, our family of basic closed sets as indicated above: just one component is allowed to differ from its corresponding whole factor, and must be a basic closed set in it.

We will be interested in the long-run behavior of points in X. To help concentrate on it, the following technical notion will be useful. For a set A, its spread up to dimension n (or to axis n) is the set $A^{(n)}$ formed by all points in X that coincide with a point in A in all components beyond n, including n itself. Thus $A^{(0)} = A$.

Proposition. *A is compact iff all its spreads are compact.*

The purpose of that definition is to make up some sort of parameter to serve as a scale against which we can quantify the compactness of A.

Specifically: let $A \subseteq X$ be compact. We define its compactness rate $d_A(n)$ as follows. Let $\mathcal{F} = \{F_i \mid i \in \omega\}$ be an infinite family of basic sets, such that $A \cap \bigcap_{i\in\omega} F_i = \emptyset$; then, by the compactness of the spreads, for each $n \in \mathbb{N}$, there is a finite subfamily of $\mathcal{F}$, $\{F_{i_1}, \ldots, F_{i_m}\} \subseteq \mathcal{F}$ of size say m, such that $A^{(n)} \cap \bigcap_{j=1}^{m} F_{i_j} = \emptyset$. Define $d_A(n)$ to be the smallest such m fulfilling this condition for all $\mathcal{F}$ (if it exists). Then we say that the compactness rate of A is the function $d_A : \mathbb{N} \mapsto \mathbb{N}$. Surely it could be undefined, in case no such m exists.

An alternative definition proposal will be discussed in a later section.

4 Learnability from Equivalence Queries

Now we select a pretty specific product topological space, $\Xi = \prod_{i\in\omega} X_i$, where each component is $X_i = 2^{2^i} = \mathcal{P}(2^i) = \mathcal{P}(\{0,1\}^i)$. As described above, we can see each point of X_i as a set of strings of i bits each; alternatively, as a boolean function on i boolean variables; or, by looking at its truth table, as a single string of length 2^i.

Thus, this product space $\Xi = \prod_{i\in\omega} X_i$ can be seen as a space of boolean functions, where each point defines one boolean function for each arity; or as a space of formal languages, each point being the characteristic function of a language $L \subseteq \{0,1\}^*$, where the i-th component provides the values of the characteristic function of L restricted to $\{0,1\}^i$.

Each X_i is endowed here with the discrete topology, and is finite, so that Ξ is compact. The basic closed sets we select in each component are those of the form $F_{w,b}$ defined as follows: for $w \in \{0,1\}^i$, the points of X_i in $F_{w,b}$ are those where the bit indexed by w is b. Equivalently, seeing each point as an i-ary boolean function f, $f \in F_{w,b}$ iff $f(w) = b$.

Proposition. *The space Ξ just defined is homeomorphic to the space $\prod_{i\in\omega}\{0,1\}$, the product space of ω copies of the binary discrete space.*

However, currently our results depend on the family of closed sets selected, and we cannot obtain the same theorems (yet?) for arbitrary homeomorphic copies of Ξ. In any case, we are working with a quite familiar space; but the way we present it is tailored to reflect, within the product topology, considerations corresponding to a length-wise treatment of formal languages over a binary alphabet; or to parallel the structure given by a family of boolean functions, one for each arity. Then, a component of a point of Ξ, on a specific dimension, can be seen as a concept that we can try to learn.

4.1 The Consistency Dimension

Many learning models have a combinatorial characterization of what is and what is not learnable, if we ignore up to polynomial space computations. For instance, a representation class is learnable from polynomially many labeled examples if and only if its Vapnik-Chervonenkis dimension is polynomial [2]. We consider now a similar parameter due to Guijarro [4] (see also [1], where this definition appears as the variant called "sphere number").

An example of length n for a concept is just a pair $\langle w, b\rangle$ where $w \in \{0,1\}^n$ and $b \in \{0,1\}$. A subset $A \subseteq \{0,1\}^n$ is consistent with it if $b \iff w \in A$; this is naturally extended to define consistency with a set of examples. The consistency dimension δ_R of a representation class R is the function that gives, for each m, the cardinality $\delta_R(n,m)$ of the smallest set of examples that is not consistent with any subset of $\{0,1\}^n$ assigned size at most m by the representation class; but such that it is minimally so. That is, removing any single example from it yields a set of examples that is consistent with some subset of $\{0,1\}^n$ (or: concept) representable within size at most m.

The name stems from the fact that it can be rewritten equivalently in the following form [1]: within $\{0,1\}^n$, and for $d = \delta_R(n,m)$, if all the subsamples of cardinality at most d of any arbitrary sample (or: set of examples) S are consistent with some concept of size m under R, then S itself is consistent with some concept of size m under R. Here the analogy to the Compactness Theorem is apparent, and motivates the theorem we state below. In the same reference, the following is proved: a representation class is learnable with polynomially many equivalence queries of polynomial size if and only if it has polynomially bounded consistency dimension.

Given a representation class R, each size bound $h : \mathbb{N} \mapsto \mathbb{N}$ defines a set $A(R,h) = \{f : \{0,1\}^* \mapsto \{0,1\} \mid R(f|_n) \leq h(n)\}$. By viewing f as consisting of ω components, $f = \langle f|_0, f|_1, \ldots, f|_n, \ldots\rangle$, each being a boolean function of arity n, being specified by 2^n bits, and thus a member $f|_n \in 2^{2^n} = X_n$, we see that $A(R,h) \subseteq \Xi$. The connection is now completed as follows.

Theorem. *As a subspace of Ξ, the set $A(R,h)$ is always compact, and the consistency dimension of R corresponds to the compactness rate of $A(R,h)$, in the following sense:*

a/ For all n, $\delta_R(n, h(n)) \leq d_{A(R,h)}(n)$

b/ The bound is tight: for infinitely many n, $\delta_R(n, h(n)) = d_{A(R,h)}(n)$

Thus, for the model of learning from equivalence queries, we obtain the following topological characterization (which we state only half-formally for now) of the representation classes that can be learned:

Corollary. *A representation class is polynomial-query learnable from* equivalence *queries iff all the compact sets obtained from it by bounding the size have* polynomially growing *(w.r.t. the size bound)* compactness rates.

5 Compactness in Terms of Convergence

We consider now the following natural alternative. Instead of considering empty intersections of closed sets, one can consider families of closed sets that have nonempty intersection, and this is equivalent (via the adherence operator) to considering filters. Indeed, compact sets are also characterized by convergence of filters, and it is natural to wonder whether the compactness rate might have a corresponding characterization in terms of some sort of convergence moduli bounds.

On the other hand, the appearance of the parameter n on the spreads can be rightfully critiziced as somewhat unnatural; an "artifact" designed essentially "to make the proof work". On the other hand, in the space Ξ we focus on, the very definition of filter offers, for a natural class of filter bases, a natural choice for the parameter n that we need, and this suggests an alternative, and maybe more pleasing, definition of compactness rate.

We explain it in this section, and state that both are useful and complement each other. Both characterize learnability, but in two related but different learning models. Thus, they are *not* equivalent.

5.1 Prefilters

Limits of filters can be defined equivalently as limits of filter bases consisting only of closed sets. Similarly to the previous case, we want to consider only basic closed sets; thus we consider *prefilters*: these are simply families of basic closed sets of the form $F_{w,b}$ having overall nonempty intersection. They are a particular form of filter bases; thus each prefilter generates the filter of all sets that contain some intersection of sets from the prefilter. Note that the previous section can be reformulated in terms of families of basic closed sets that are *not* prefilters. A prefilter *converges* if the intersection of the whole family is a single point. Convergence in a subspace is defined in the same manner on the traces of all the elements of the prefilter on the subspace, provided that the resulting family is still a prefilter (i.e. has nonempty intersection).

The width of a prefilter $\mathcal{F}$ is the function $w_{\mathcal{F}}(n)$ that, at each n, gives the number of different elements of $\mathcal{F}$ of the form $F_{w,b}$ with $|w| = n$. Note that there are at most 2^n of them since each w cannot appear both with b and $\neg b$ in $\mathcal{F}$, due to the nonempty intersection condition.

Let $\mathcal{F}$ be a prefilter that converges in the subspace A. The convergence delay of $\mathcal{F}$ in A is the width of the smallest subprefilter of $\mathcal{F}$ that converges in A (and, a fortiori, towards the same limit). The general convergence delay of A is the largest convergence delay of a prefilter in A. (In principle, this might as well be undefined.)

Our result regarding this notion (even more subject to potential mistakes than the previous one, though) is:

Theorem. *A representation class is polynomial-query learnable from* membership *queries iff all the compact sets obtained from it by bounding the size have* polynomially growing *(w.r.t. the size bound)* general convergence delays.

Again this characterization lies on a combinatorial characterization of the polyquery learning protocol. For membership queries, the concept of teaching dimension (see [3] and the references therein) was proved in [5] to characterize the model for projection-closed classes, which actually do encompass all classes of interest. Inspired by the definition of consistency dimension, we found a variant that catches membership query learnability also for non-projection-closed classes. While this is uninteresting since no really new reasonable classes get captured, the cleaner form of the statement allows for a translation from the combinatorial setting into topology: it is exactly the bound on the convergence delay of prefilters.

Thus, the way out we found, based on convergence issues, to circumvent the objections made to the concept of spread does not substitute, but in a sense complements, the notion; and certainly in a rather intriguing manner!

6 Work in Progress

We see quite a few additional aspects to be worked out, and we are actively (but, alas, slowly) pursuing some of them.

6.1 Computational Learning Issues

One clear limitation of this work is the restriction to "mean" algorithms. We are studying how to adjust the technicalities to capture algorithms that actually query hypothesis that are larger than necessary, since several of these do exist in the literature. Some of the results of [1] do apply to the general case, but not all.

Another direction where a generalization of this work is needed is to improper learning; this simply means that the hypothesis are allowed to come from a different representation class. Several classes are not known to be learnable in terms of themselves, but become so when the hypothesis space allowed is enlarged.

Starting from certificates and the consistency dimension, we then found new characterizations for membership queries (which lead to our result on prefilters); but moreover the shape of the expressions and their similarities suggest an avenue to work on similar characterizations for other learning protocols, and actually the scratch workpapers on our desks already have a solution for the case of learning from subset queries. It may have also a topological interpretation. Whereas, at the time of writing, all this material is extremely immature, please check with the author at the time of reading...

Globally, our vision would be a topologically suggested notion of "learnable" which would have as particular cases all (or, more modestly, a handful of) the learnability notions currently studied, which are incomparable among them.

6.2 Logic Issues

The major shortcoming of this preliminary work from the logic side is the restriction to the rather trivial propositional case. We want to explore the alley of finite models from this perspective. One natural (maybe too trivial) possibility is as follows: the i-th factor space would not be the i-ary boolean functions but the set of strings encoding finite models for universe size i, and would have a length polynomial in i depending on the relational vocabulary.

The dream at this point is: even though the standard Compactness Theorem of first-order logic is well-known to fail for the finite models world, might it be simply that we need to sharpen it a little with quantitative considerations in order to recover it? More generally, would a quantitative approach reunify Classical Model Theory with its stray offspring, Finite Model Theory, and in this manner enrich its cousin, a favorite of mine, Complexity Theory, with its nice arsenal of deep weapons? But this is just the dream.

6.3 Topological Issues

I am far from being a topologist, to the extent that I might be rediscovering the wheel all around here (although neither Altavista nor Infoseek seemed able to find too close together the words "quantitative" and "compactness" anywhere in the planet). Assume anyone would accept my bold claim that the notions of "topology" and "quantitative" are *not* mutually excluding. Then, clearly there are topological questions here, ranging from careful study of the details (might it be that A is not compact but some, or infinitely many, or almost all of the spreads are? how often can d_A be defined/undefined? does it make sense to speak of "closeness to compact" for noncompact subsets on the basis of these definitions?) to more general questions which I am not in a position to enumerate, but of which I will mention just one that worries me most.

Not only we worked most of the time in a quite specific topological space, but we even fixed the family of basic closed sets. I am sure that a justification for this choice can be found, beyond its to me obvious naturality: something like "this choice gives values for the compactness rates that are in a sense extreme in comparison with any other choice of basic sets for the same topology". This would be a very first step necessary before transfering any of the intuitions that one could get from here into products involving larger ordinals, or even arbitrary abstract topological spaces.

References

1. J Balcázar, J Castro, D Guijarro, H-U Simon. The consistency dimension and distribution-dependent learning from queries. To appear in Algorithmic Learning Theory ALT99, Tokyo.
2. A Blumer, A Ehrenfeucht, D Haussler, M Warmuth: Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM* **36**, 4 (1989), 929–965
3. S Goldman, M Kearns: On the complexity of teaching. *Journal of Computer and System Sciences* **50** (1995), 20–31
4. D Guijarro: Ph D dissertation, in preparation
5. L Hellerstein, K Pillaipakkamnat, V Raghavan, D Wilkins: How many queries are needed to learn? *Journal of the ACM* **43**, 5 (1996), 840–862
6. M Kearns, U Vazirani: *An Introduction to Computational Learning Theory*. MIT Press 1994.
7. M Raman: Understanding compactness: a historical perspective. M. A. thesis in Mathematics, UC Berkeley, 1997; found at the time of writing in the URL: `http://socrates.berkeley.edu/~manya`
8. M Sipser: Borel sets and circuit complexity. 15th ACM Symp. on Theory of Computing (1983), 61–69.

Descriptive and Parameterized Complexity

Martin Grohe

Institut für Mathematische Logik, Eckerstr.1, 79104 Freiburg, Germany
grohe@sun2.mathematik.uni-freiburg.de

Abstract. Descriptive Complexity Theory studies the complexity of problems of the following type:

Given a finite structure $\boldsymbol{A}$ and a sentence φ of some logic L, decide if $\boldsymbol{A}$ satisfies φ?

In this survey we discuss the *parameterized complexity* of such problems. Basically, this means that we ask under which circumstances we have an algorithm solving the problem in time $f(|\varphi|)||\boldsymbol{A}||^c$, where f is a computable function and $c > 0$ a constant. We argue that the parameterized perspective is most appropriate for analyzing typical practical problems of the above form, which appear for example in database theory, automated verification, and artificial intelligence.

1 Introduction

One of the main themes in descriptive complexity theory is to study the complexity of problems of the following type:

> Given a finite structure $\boldsymbol{A}$ and a sentence φ of some logic L, decide if $\boldsymbol{A}$ satisfies φ?

This problem, let us call it the *model-checking problem for* L, has several natural variants. For example, given a structure $\boldsymbol{A}$ and a formula $\varphi(\bar{x})$, we may want to compute the set of all tuples $\bar{a} \in A$ such that $\boldsymbol{A}$ satisfies $\varphi(\bar{a})$, or we may just want to count the number of such tuples. Often, we fix the sentence φ in advance and consider the problem: Given a structure $\boldsymbol{A}$, decide if $\boldsymbol{A}$ satisfies φ?

Model-checking problems and their variants show up very naturally in various applications in computer science. Let us consider three important examples.

Database Query Evaluation. Relational databases are finite relational structures, and query languages are logics talking about these structures. Thus the problem of evaluating a Boolean query φ over a database D is just the model-checking problem for the query language. Evaluating a k-ary query corresponds to the problem of finding all tuples in a structure satisfying a formula with k free variables.

A very important class of queries is the class of *conjunctive queries*. Such a query can be described by a first-order formula of the form $\exists\bar{y}\ \theta(\bar{x}, \bar{y})$, where $\theta(\bar{x}, \bar{y})$ is a conjunction of atomic formulas in the variables $\bar{x}, \bar{y}$.

Chandra and Merlin [CM77] noted that the model checking problem for conjunctive queries is essentially the same as the *homomorphism problem*: Given

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 14–31, 1999.

two finite structures $\boldsymbol{A}$ and $\boldsymbol{B}$ of the same relational vocabulary τ, decide if there is a homomorphism from $\boldsymbol{A}$ to $\boldsymbol{B}$? Remember that a *homomorphism* from $\boldsymbol{A}$ to $\boldsymbol{B}$ is a mapping $h : A \to B$ with the property that for all k-ary $R \in \tau$ and $\bar{a} \in A^k$ such that $R^{\boldsymbol{A}}\bar{a}$ we have $R^{\boldsymbol{B}}(h(\bar{a}))$.

To reduce the model-checking problem for conjunctive queries to the homomorphism problem, with each formula φ of vocabulary τ with variables $x_1, \ldots, x_n$ we associate a τ-structure $\boldsymbol{A}_\varphi$ with universe $A_\varphi = \{x_1, \ldots, x_n\}$, in which for k-ary $R \in \tau$ and $\bar{x} \in A_\varphi^k$ we have $R^{\boldsymbol{A}_\varphi}(\bar{x})$ if, and only if, $R(\bar{x})$ is a subformula of φ. Then if $\varphi(\bar{x})$ is a conjunction of atomic formulas, a structure $\boldsymbol{B}$ satisfies $\exists\bar{x}\varphi(\bar{x})$ if, and only if, there is a homomorphism from $\boldsymbol{A}_\varphi$ to $\boldsymbol{B}$. For the other direction we proceed similarly; for each finite relational structure $\boldsymbol{A}$ we define a conjunctive query $\varphi_{\boldsymbol{A}}$ that is satisfied by a structure $\boldsymbol{B}$ if, and only if, there is a homomorphism from $\boldsymbol{A}$ to $\boldsymbol{B}$.

Thus indeed the model-checking problem for conjunctive queries and the homomorphism problem are essentially the same.

Constraint Satisfaction Problems. Feder and Vardi [FV93] gave the following elegant general formulation of a constraint satisfaction problem: Given two structures $\boldsymbol{I}$, called the *instance*, and $\boldsymbol{T}$, called the *template*, find a homomorphism from $\boldsymbol{I}$ to $\boldsymbol{T}$.

Then more specific problems can be obtained by restricting instances and templates to be taken from certain classes of structures. For example, for the *graph coloring problem* we allow all (undirected, loop-free) graphs as instances and all complete graphs K_k, for $k \geq 1$, as templates. Another example of a constraint satisfaction problem is 3-satisfiability, we leave it as an exercise to the reader to formulate it as a homomorphism problem.

We can conclude that a constraint satisfaction problem is basically the same as the model-checking problem for conjunctive queries. This is true, but for reasons we will explain later a different formulation of a constraint satisfaction problem as model-checking problem is more appropriate. Recall that monadic second-order logic is the extension of first-order logic by quantifiers ranging over sets of elements of a structure. It is easy to see that for each structure $\boldsymbol{T}$ there is a sentence $\varphi_{\boldsymbol{T}}$ of monadic second-order logic of the form $\exists X_1 \ldots \exists X_k\ \psi$, where ψ is a universal first-oder formula, such that a structure $\boldsymbol{I}$ satisfies $\varphi_{\boldsymbol{T}}$ if, and only if, there is a homomorphism from $\boldsymbol{I}$ to $\boldsymbol{T}$ [FV93].

Hence a constraint satisfaction problem is essentially a special case of the model-checking problem for monadic second-order logic.

Model-Checking in Computer-Aided Verification. In the model checking approach to verification of circuits and protocols, the formal design of the system is translated to a Kripke structure representing its state space. Then correctness conditions are formulated in a logic, for example CTL or the modal μ-calculus, and the model-checker automatically tests whether the Kripke structure satisfies the conditions.

For the various logics used in this area, very good complexity bounds for the model-checking problem are known. However, the techniques to prove these

bounds are of a quite different flavor than those needed, say, to analyze first-order model-checking. We will not study such logics in this paper.

Except for the modal and temporal logics used in verification, the complexity of model-checking problems is usually quite high. For example, model-checking for first-order logic is PSPACE-complete [SM73,Var82], and even for conjunctive queries it is NP-complete [CM77]. However, these complexity theoretic results are not really meaningful in many practical situations where the model-checking problems occur. In all the examples we have seen we can assume that usually the input formula φ is quite small, whereas the structure A can be very large. (This, by the way, is the reason that in constraint satisfaction problems we wrote a formula representing the template and not the instance.)

For that reason it is argued, for example in database theory, that we can assume that the length of the queries is bound by some small number l and then more or less neglect it. Indeed, evaluating a conjunctive query of length at most l in a database of size n requires time at most $O(n^l)$. For a fixed l this is in polynomial time and thus seems to be fine. Of course it is not. Even for a query length $l = 5$ this is far too much. On the other hand, a running time $O(2^l n)$, which is still exponential in l, would be acceptable. *Parameterized complexity theory* has been developed to deal with exactly this kind of situation.

Fixed-Parameter Tractability

The idea of parameterized complexity theory is to *parameterize* a problem by some function of the input (such as, for example, the valence of the input graph) and then measure the complexity of the problem not only in terms of the size of the input, but also in terms of the parameter. This leads to a refined analysis of the complexity of the problem, which can be very useful if we have some additional information on the parameter, for example that it is usually "small".

Formally, a *parameterized problem* is a set $P \subseteq \Sigma^* \times \Pi^*$, where Σ and Π are finite alphabets. We usually represent a parameterized problem P in the following form:

Input: $I \in \Sigma^*$
Parameter: $\pi \in \Pi^*$
Problem: Decide if $(I, \pi) \in P$.

In most cases, we let $\Pi = \{0, 1\}$ and consider the parameters $\pi \in \Pi^*$ as natural numbers (in binary). Very often, a parameterized problem P is derived from a (classical decision) problem $L \subseteq \Sigma^*$ by a *parameterization* $p : \Sigma^* \to \mathbb{N}$ in such a way that $P = \{(I, k) \mid I \in L, k = p(I)\}$. Slightly abusing notation, we represent such a P in the form

Input: $I \in \Sigma^*$
Parameter: $p(I)$
Problem: Decide if $I \in L$.

As an example, actually the most important example in this paper, consider the model-checking problem for first-order logic parameterized by the formula-length:

Input: Structure A, FO-sentence φ
Parameter: $\|\varphi\|$
Problem: Decide if A satisfies φ.

The following central definition is motivated by our considerations at the end of the last subsection.

Definition 1. A parameterized problem $P \subseteq \Sigma^* \times \Pi^*$ is *fixed-parameter tractable* if there is a function $f : \Pi^* \to \mathbb{N}$, a constant $c \in \mathbb{N}$ and an algorithm that, given a pair $(I, \pi) \in \Sigma^* \times \Pi^*$, decides if $(I, \pi) \in P$ in time $f(\pi)|I|^c$.

We denote the class of all fixed-parameter tractable problems by FPT. By FPL we denote the class of those problems $P \in$ FPT for which the constant c in Definition 1 can be chosen to be 1.

Note that every decidable problem $L \subseteq \Sigma^*$ has a parameterization $p : \Sigma^* \to \mathbb{N}$ such that the resulting parameterized problem P is trivially in FPL: Just let $p(I) = |I|$. The choice of a good parameterization is hence a crucial part of the complexity analysis of a problem. This remark is further illustrated by the following example. Recall that evaluation of conjunctive queries and the constraint satisfaction problem are both essentially the same as the homomorphism problem. Nevertheless, we decided to consider the former as a special case of the model-checking problem for first-order logic and the latter as a special case of the model-checking problem for monadic second-order logic. Since the "generic" parameterization of model-checking problems is by the length of the input formula, this just corresponds to two different parameterizations of the homomorphism problem, each of which is appropriate in the respective application.

The theory of fixed-parameter tractability and intractability has mainly been developed by Downey and Fellows. For a comprehensive treatment of the theory I refer the reader to their recent monograph [DF99].

In this paper we study the complexity of model-checking and related problems from the parameterized perspective. As argued above, proving that a model-checking problem is in FPT or, even better, in FPL often seems to be a much more meaningful statement than just the fact that for a fixed bound on the formula length a model-checking problem is in PTIME. In the context of database theory, the question of fixed-parameter tractability of query evaluation has first been brought up by Yannakakis [Yan95].

As one might expect, in general most model-checking problems are not fixed-parameter tractable. Therefore, Section 2 is devoted to intractability results. In the Sections 3 and 4 we discuss various restrictions leading to problems in FPT and FPL. We close the paper with a list of open problems.

Although our main motivation is to use parameterized complexity theory for a refined analysis of typical problems of descriptive complexity theory, we will see

that parameterized complexity theory also benefits from the logical perspective of descriptive complexity theory.

Preliminaries

We assume that the reader has some background in logic and, in particular, is familiar with first-order logic FO and monadic second-order logic MSO.

For convenience, we only consider relational vocabularies.[1] τ always denotes a vocabulary. We denote the universe of a structure $\boldsymbol{A}$ by A. The interpretation of a relation symbol $R \in \tau$ in a τ-structure $\boldsymbol{A}$ is denoted by $R^{\boldsymbol{A}}$.

We only consider finite structures. The class of all (finite) structures is denoted by $\mathcal{F}$. If $\mathcal{C}$ is a class of structures, then $\mathcal{C}[\tau]$ denotes the class of all τ-structures in $\mathcal{C}$. We consider *graphs* as $\{E\}$-structures, where E is a binary relation symbol. Graphs are always undirected and loop-free. $\mathcal{G}$ denotes the class of all graphs.

We use RAMs as our underlying model of computations. The *size* of a τ-structure $\boldsymbol{A}$, denoted by $||\boldsymbol{A}||$, is defined to be $|A| + \sum_{R\ \tau} |R^A|$. When doing computations whose inputs are structures, we assume that the structures are given by an adjacency list representation. This is important when it comes to *linear time* complexity. For details on these sensitive issues I refer the reader to [See96].

2 Intractability

In this section we give some evidence that for most of the logics we have discussed so far the model-checking problem is not in FPT. As it is often the case in complexity theory, we can not actually prove this, but only prove that all the model-checking problems are hard for a complexity class W[1], which is conjectured to contain FPT strictly. To do this we need a suitable concept of reduction that we introduce in a moment.

However, before we do so we observe that for the MSO-model-checking problem (parameterized by the formula length) we can show that it is not in FPT unless P = NP without any further knowledge of parameterized complexity theory. We just observe that there is an MSO-sentence χ defining the class of all 3-colorable graphs:

$$\chi := \exists X \exists Y \exists Z \Big(\forall x (Xx \vee Yx \vee Zx) \\ \wedge \forall x \forall y \big(Exy \to \neg((Xx \wedge Xy) \vee (Yx \wedge Yy) \vee (Zx \wedge Zy))\big)\Big).$$

Recall that if the model-checking problem for MSO was in FPT, there would be a function $f : \mathbb{N} \to \mathbb{N}$, a constant $c \in \mathbb{N}$, and an algorithm that, given a graph $\boldsymbol{G}$ and an MSO-sentence φ, would decide whether $\boldsymbol{G}$ satisfies φ in time $f(|\varphi|)n^c$, where n is the size of the input graph. Applied to the sentence χ this

[1] All results we present here can be extended to vocabularies with function and constant symbols, see [FG99a] for details.

would yield an $O(n^c)$-algorithm for 3-colorability and thus imply that P = NP, because 3-Colorability is NP-complete [Kar72].

Definition 2. Let $P \subseteq \Sigma^* \times \Pi^*$ and $P' \subseteq (\Sigma')^* \times (\Pi')^*$ be parameterized problems.

P is *parameterized m-reducible* to P' (we write $P \leq_m^{\text{fp}} P'$), if there is a computable function $f : \Pi^* \to \mathbb{N}$, a constant $c \in \mathbb{N}$, a computable function $g : \Pi^* \to (\Pi')^*$, and an algorithm that, given $(I, \pi) \in \Sigma^* \times \Pi^*$, computes an $I' \in (\Sigma')^*$ in time $f(\pi)|I|^c$ such that

$$(I, \pi) \in P \iff (I', g(\pi)) \in P'.^2$$

Observe that $\leq_m^{\text{fp}}$ is transitive and that if $P \leq_m^{\text{fp}} P'$ and $P' \in \text{FPT}$ then $P \in \text{FPT}$. A *parameterized complexity class* is a class of parameterized problems that is downward closed under $\leq_m^{\text{fp}}$. For a parameterized problem P we let $[P]_{\text{fp}} := \{P' \mid P' \leq_m^{\text{fp}} P\}$, and for a family $\mathcal{P}$ of problems we let $[\mathcal{P}]_{\text{fp}} := \bigcup_{P \in \mathcal{P}} [P]_{\text{fp}}$. Now we can define *hardness* and *completeness* of parameterized problems for a parameterized complexity class (under parameterized m-reductions) in the usual way.

For a class $\mathcal{C}$ of τ-structures and a class L of formulas we let $\text{MC}(\mathcal{C}, \text{L})$ be the following parameterized model-checking problem:

Input: $\boldsymbol{A} \in \mathcal{C}$, $\varphi \in \text{L}$
Parameter: $\lvert\varphi\rvert$
Problem: Decide if $\boldsymbol{A} \models \varphi$.

For an arbitrary class $\mathcal{C}$ of structures, $\text{MC}(\mathcal{C}, \text{L})$ denotes the family of problems $\text{MC}(\mathcal{C}[\tau], \text{L})$, for all τ.

We call a first-order formula *existential* if it contains no universal quantifiers and if negation symbols only occur in front of atomic subformulas. EFO denotes the class of all existential FO-formulas. We let

$$\text{W}[1] := [\text{MC}(\mathcal{F}, \text{EFO})]_{\text{fp}}.^3$$

Recall that $\mathcal{F}$ denotes the class of all structures and $\mathcal{G}$ denotes the class of graphs. Standard encoding techniques show that $\text{MC}(\mathcal{G}, \text{EFO})$ is complete for W[1], or equivalently, that $[\text{MC}(\mathcal{G}, \text{EFO})]_{\text{fp}} = \text{W}[1]$ [FG99a].

Let Clique be the parameterized problem

[2] This is what Downey and Fellows [DF99] call strongly uniformly parameterized m-reducible. They also use various other reduction concepts, most notably a parameterized form of Turing reductions.

[3] This is not Downey and Fellow's original definition. See below for a discussion of the W-hierarchy.

Input:	Graph $\boldsymbol{G}$
Parameter:	$k \in \mathbb{N}$
Problem:	Decide if $\boldsymbol{G}$ has a k-clique.

(A k-clique in a graph is a set of k pairwise connected vertices.)

Theorem 1 ([DF95]). Clique *is complete for* W[1].

Proof. The problem is obviously contained in W[1], because for every k there is an EFO-sentence (actually a conjunctive query) of length bounded by a computable function of k that defines the class of all graphs with a k-clique.

For the hardness, we shall prove that $\mathrm{MC}(\mathcal{G}, \mathrm{EFO}) \leq_m^{\mathrm{fp}} \textsc{Clique}$.

An *atomic k-type* (in the theory of graphs) is a sentence $\theta(x_1, \dots, x_k)$ of the form $\bigwedge_{1 \leq i < j \leq k} \alpha_{ij}(x_i, x_j)$, where $\alpha_{ij}(x_i, x_j)$ is either $x_i = x_j$ or $E(x_i, x_j)$ or $(\neg E(x_i, x_j) \wedge \neg x_i = x_j)$ (for $1 \leq i < j \leq k$).

It is easy to see that there is a computable mapping f that associates with each EFO-sentence φ a sentence $\tilde{\varphi}$ of the form

$$\bigvee_{i=1}^{l} \exists x_1 \dots \exists x_k \theta_i(x_1, \dots, x_k), \tag{1}$$

where each θ_i is an atomic k-type, such that for all graphs $\boldsymbol{G}$ we have $\boldsymbol{G} \models \varphi \iff \boldsymbol{G} \models \tilde{\varphi}$. Furthermore, the mapping $\varphi \mapsto \tilde{\varphi}$ can be defined in such a way that k is precisely the length of φ (this can simply be achieved by first adding "dummy" variables). Let $\tilde{f}(x)$ be an upper bound on the time required to compute $\tilde{\varphi}$ for a formula φ of length x.

For each graph $\boldsymbol{G}$ and each atomic k-type $\theta(\bar{x}) = \bigwedge_{1 \leq i < j \leq k} \alpha_{ij}(x_i, x_j)$ we define a graph $h(\boldsymbol{G}, \theta)$ as follows:

- The universe of $h(\boldsymbol{G}, \theta)$ is $\{1, \dots, k\} \times G$.
- There is an edge between (i, v) and (j, w), for $1 \leq i < j \leq k$ and $v, w \in G$, if $\boldsymbol{G} \models \alpha_{ij}(v, w)$.

Then $h(\boldsymbol{G}, \theta)$ contains a k-clique if, and only if, $\boldsymbol{G} \models \exists \bar{x} \theta(\bar{x})$.

Now we are ready to define the reduction from $\mathrm{MC}(\mathcal{G}, \mathrm{EFO})$ to Clique. We let $c = 1$ and $g(k) = k$. Given a graph $\boldsymbol{G}$ and a sentences $\varphi \in \mathrm{EFO}$, our reduction-algorithm first computes $\tilde{\varphi} = \bigvee_{i=1}^{l} \exists \bar{x} \theta_i(\bar{x})$. Then for $1 \leq i \leq l$ it computes $h(\boldsymbol{G}, \theta_i)$, and the output is the disjoint union of all these graphs. This computation requires time $O(\tilde{f}(p) p n)$, where $n = |G|$ and $p = |\varphi|$. Thus we can let $f(x) := d\tilde{f}(x)x$ for a sufficiently large constant d. □

Letting CQ be the class of all conjunctive queries, we immediately obtain:

Corollary 1 ([PY97]). $\mathrm{MC}(\mathcal{G}, \mathrm{CQ})$ *is complete for* W[1].

There is good reason to conjecture that W[1] ≠ FPT (see [DF99]). If we believe this, model-checking is not even fixed-parameter tractable for existential

FO-formulas or conjunctive queries. How much harder will it be for arbitrary formulas?

Actually, Downey and Fellows defined a whole hierarchy $\mathrm{W}[1] \subseteq \mathrm{W}[2] \subseteq \cdots$ of parameterized complexity classes, and they conjecture that this hierarchy is strict. Their definition of this W-hierarchy is in terms of the circuit value problem for certain classes of Boolean circuits. It is not hard to prove, though, that their definition of W[1] is equivalent to ours.

For $i \geq 1$, we let Σ_i denote the class of all FO-formulas in prenex normal form that have i alternating blocks of quantifiers, starting with an existential quantifier. Coming from our definition of W[1] in terms of model-checking for existential FO-formulas, it is tempting to conjecture that for $i \geq 2$, the class $\mathrm{W}[i]$ coincides with the class $\mathrm{A}[i] := [\mathrm{MC}(\mathcal{F}, \Sigma_i)]_{\mathrm{fp}}$. This is an open question, but I tend to believe that $\mathrm{A}[i] \neq \mathrm{W}[i]$ for all $i \geq 2$ (see [FG99a,DFR98] for a discussion).

In any case, the intuition that the W-hierarchy is closely related to quantifier-alternation in FO-model-checking problems is justified. For $l \geq 1$, $i \geq 2$ we let $\Sigma_{i,l}$ be the set of all Σ_i-formulas with at most l quantifiers in all quantifier blocks except for the first. For example,

$$\exists x_1 \ldots \exists x_k \forall y \exists z_1 \exists z_2 \forall w_1 \forall w_2 \; \theta$$

with a quantifier-free θ is a $\Sigma_{4,2}$-formula.

Theorem 2 ([DFR98,FG99a][4]). *For all $i \geq 1$ we have*

$$\mathrm{W}[i] = \left[\bigcup_{l \geq 1} \mathrm{MC}(\mathcal{F}, \Sigma_{i,l}) \right]_{\mathrm{fp}} = [\mathrm{MC}(\mathcal{F}, \Sigma_{i,1})]_{\mathrm{fp}} .$$

On top of the W-hierarchy, Downey and Fellows have studied lots of other parameterized complexity classes. Notably, Downey, Fellows and Taylor [DFT96] proved that the problem $\mathrm{MC}(\mathcal{G}, \mathrm{FO})$, model-checking for FO, is complete for the class AW[$*$] (for a definition of this class I refer the reader to [DF99]).

3 Tractable Cases I: Simple Structures

In this section we study tractable cases of the model-checking problems that are obtained by restricting the class of input structures. We prove a theorem of Courcelle stating that MSO-model-checking is in FPL if parameterized by the formula length and the *tree-width* of the input structure.

Recall that a tree is a connected acyclic graph.

The *union* $\bigcup_{i \in I} \boldsymbol{A}_i$ of τ-structures $\boldsymbol{A}_i$ is the τ-structure $\boldsymbol{A}$ with universe $A = \bigcup_{i \in I} A_i$ and $R^{\boldsymbol{A}} = \bigcup_{i \in I} R^{\boldsymbol{A}_i}$. If $\boldsymbol{A}$ is a τ-structure and $B \subseteq A$, then $\langle B \rangle^{\boldsymbol{A}}$ denotes the substructure induced by A on B.

[4] The first equality is due to [DFR98], the second due to [FG99a].

Definition 3. (1) A *tree-decomposition* of a structure $\boldsymbol{A}$ is a pair $(\boldsymbol{T}, (A_t)_{t\ T})$ consisting of a tree $\boldsymbol{T}$ and a family A_t of subsets of A (for $t \in T$) such that $\langle A_t \rangle^{\boldsymbol{A}} = \boldsymbol{A}$ and for all $a \in A$, the set $\{t \mid a \in A_t\}$ induces a subtree of $\boldsymbol{T}$ (that is, is connected). The sets A_t are called the *parts* of the decomposition.
(2) The *width* of $(\boldsymbol{T}, (A_t)_{t\ T})$ is defined to be $\max\{|A_t| \mid t \in T\} - 1$.
(3) The *tree-width* of $\boldsymbol{A}$, denoted by $\mathrm{tw}(\boldsymbol{A})$, is the minimal width of a tree-decomposition of $\boldsymbol{A}$.

Trees and forests have tree-width 1 and series-parallel graphs (in particular cycles) have tree-width ≤ 2. Note that a graph of size n has tree-width at most $n-1$. An n-clique has tree-width $(n-1)$, an $(n \times n)$-grid has tree-width n (see [Die97]), and a random graph of order n with edge probability $\gg \frac{1}{n}$ has tree-width $n - o(n)$ almost surely [GL].

Tree-decompositions and tree-width have been introduced by Halin [Hal76], and later independently by Robertson and Seymour [RS86a]. They are of great importance in graph theory and the theory of algorithms. Many NP-hard algorithmic problems on graphs belong to FPL when parameterized by the tree-width of the input graph (see, for example, the survey [Bod97]).

Computing a tree-decomposition of a given graph is NP-complete [ACP87]. However, if parameterized by the tree-width of the input structure, the problem is fixed-parameter tractable by the following deep theorem of Bodlaender.

Theorem 3 ([Bod96]). *There is an algorithm that, given a graph $\boldsymbol{G}$, computes a tree-decomposition of $\boldsymbol{G}$ of minimal width in time $O(2^{p(\mathrm{tw}(\boldsymbol{G}))}|G|)$ (for a suitable polynomial $p(X)$).*

It is not hard to see that for arbitrary τ the analogous result for τ-structures follows.

Theorem 4 ([Cou90]). *For every vocabulary τ, the following parameterized problem is in* FPL*:*

Input:	τ-structure $\boldsymbol{A}$, MSO-sentence φ
Parameter:	$(\mathrm{tw}(\boldsymbol{A}), \lvert\varphi\rvert)$
Problem:	Decide if $\boldsymbol{A} \models \varphi$.

Let us recall what exactly this result means (cf. Page 16 for the precise definitions): Suppose that we encode pairs $(\boldsymbol{A}, \varphi)$ by a suitable mapping $\mathrm{enc} : \mathcal{F}[\tau] \times \mathrm{MSO} \to \{0,1\}^*$. Then formally the parameterized problem we consider is

$$\big\{\big(\mathrm{enc}(\boldsymbol{A}, \varphi), (k, l)\big) \mid \boldsymbol{A} \in \mathcal{F}[\tau], \varphi \in \mathrm{MSO}, k = \mathrm{tw}(\boldsymbol{A}), l = |\varphi|, \boldsymbol{A} \models \varphi\big\}.$$

Thus the theorem states that there is a computable function $f : \mathbb{N}^2 \to \mathbb{N}$ and an algorithm that, given $\big(\mathrm{enc}(\boldsymbol{A}, \varphi), (k, l)\big)$, decides if $k = \mathrm{tw}(\boldsymbol{A})$, $l = |\varphi|$, and $\boldsymbol{A} \models \varphi$ in time at most $f(k, l)n$.

The proof requires some preparation.

Once we have declared a node r in a tree $\boldsymbol{T}$ to be the *root* we can direct the edges and speak of the children of a node and its parent. A *proper binary rooted tree* is a rooted tree $(\boldsymbol{T}, r)$ where every vertex either has two children or none.

It will be convenient for us to work with tree-decompositions of the following form: A *special tree-decomposition (STD)* of width w of a structure $\boldsymbol{A}$ is a triple $(\boldsymbol{T}, r, (\bar{a}^t)_{t\ T})$ where $(\boldsymbol{T}, r)$ is a proper binary rooted tree, $\bar{a}^t := (a_0^t, \dots, a_w^t)$ is a $(w+1)$-tuple of elements of $\boldsymbol{A}$, and $(\boldsymbol{T}, (\{a_0^t, \dots, a_w^t\})_{t\ T})$ is a tree-decomposition of $\boldsymbol{A}$. Let us mention explicitly that two distinct tree-nodes of a tree-decomposition may have identical parts.

It is easy to see that a given tree-decomposition of a graph can be transferred to an STD of the same width in linear time.

The *quantifier-rank* of an MSO-formula φ is the maximal number of nested quantifiers in φ. Let $q \geq 1$. An MSO_q *k-type* (of vocabulary τ) is a set of MSO-formulas of quantifier rank at most q whose free variables are contained in a fixed set $\{x_1 \dots, x_k\}$. The MSO_q *type* $\mathrm{tp}_q(\bar{a}, \boldsymbol{A})$ of a k-tuple $\bar{a} \in A^k$ in a τ-structure $\boldsymbol{A}$ is defined to be the set of all MSO-formulas $\varphi(\bar{x})$ of quantifier-rank at most q such that $\boldsymbol{A} \models \varphi(\bar{a})$.

It is easy to see that, up to logical equivalence, there are only finitely many MSO-formulas of vocabulary τ with free variables in $\{x_1 \dots, x_k\}$ of quantifier rank at most q. Thus for all $k, q \in \mathbb{N}$, up to logical equivalence there are only finitely many MSO_q k-types, and every MSO_q k-types has a finite description.

Proof (of Theorem 4): Let φ be an MSO-sentence of quantifier-rank q. Furthermore, let $\boldsymbol{A}$ be a τ-structure and $(\boldsymbol{T}, r, (\bar{a}^t)_{t\ T})$ an STD of $\boldsymbol{A}$ of width $w := \mathrm{tw}(\boldsymbol{A})$.

For every $t \in T$, we let $A_t := \{a_0^t, \dots, a_w^t\}$ and $B_t := \bigcup_{t \leq^{\boldsymbol{T}} s} A_s$, where $\leq^{\boldsymbol{T}}$ denotes the natural partial order associated with the tree (in which the root is minimal). Note that for leaves t we have $B_t = A_t$ and for parents t with children u and u' we have $B_t = A_t \cup B_u \cup B_{u'}$ and $B_u \cap B_{u'} \subseteq A_t$.

Standard techniques from logic (Ehrenfeucht-Fraïssé games) easily show that for every parent t with children u, u', $\mathit{type}(t) := \mathrm{tp}_q(\bar{a}^t, \langle B_t \rangle^{\boldsymbol{A}})$ only depends on the following finite pieces of information:

(1) The isomorphism type of the substructure induced by $\bar{a}^t$, that is,

$$\begin{aligned}\mathit{part}(t) :=& \big\{(x,y) \mid 0 \leq x < y \leq w, a_x^t = a_y^t\big\} \\ &\cup \big\{R(\bar{x}) \mid R \in \tau \text{ } k\text{-ary}, \bar{x} \in \{0, \dots w\}^k \text{ such that } R^{\boldsymbol{A}}(a_{x_1}, \dots, a_{x_k})\big\}\end{aligned}$$

(2) The q-type of the subtree below u, and how the parts at u and t intersect:

$$\mathit{type}(u), \quad \mathit{is}(u,t) := \{(x,y) \mid 0 \leq x, y \leq w, a_x^u = a_y^t\}.$$

(3) The q-type of the subtree below u', and how the parts at u' and t intersect:

$$\mathit{type}(u'), \quad \mathit{is}(u', t).$$

For a leaf t, the q-type $\mathit{type}(t)$ only depends on $\mathit{part}(t)$.

In other words, there are finite functions $\Pi_{q,w}$ and $\Lambda_{q,w}$ such that for all τ-structures $\boldsymbol{A}$ with STD $(\boldsymbol{T}, r, (\bar{a}^t)_{t\ T})$ of width w we have

$$\begin{aligned}\mathit{type}(t) =& \Pi_{q,w}\big(\mathit{part}(t), \mathit{type}(u), \mathit{is}(u,t), \mathit{type}(u\ ), \mathit{is}(u\ ,t)\big)\\ &\text{for all parent nodes } t \in T \text{ with children } u, u\ ,\\ \mathit{type}(t) =& \Lambda_{q,w}\big(\mathit{part}(t)\big) \qquad \text{for all leaves } t \in T.\end{aligned}$$

The functions $\Pi_{q,w}$ and $\Lambda_{q,w}$ only depend on q and w, but not on the input structure $\boldsymbol{A}$. Furthermore, there is an algorithm that, given q and w, computes $\Pi_{q,w}$ and $\Lambda_{q,w}$ and stores them in look-up tables.

Finally, recall that $\langle B_r\rangle^{\boldsymbol{A}} = \boldsymbol{A}$ and $\big(\boldsymbol{A} \models \varphi \iff \varphi \in \mathrm{tp}_q(\bar{a}^r, \boldsymbol{A})\big)$ (because the quantifier rank of φ is q).

It is now easy to verify that the algorithm in Figure 1 solves the MSO-model-checking problem in time $f(\mathrm{tw}(\boldsymbol{G}), |\varphi|)|G|$ for a computable f. The statement of the theorem follows, because the problem of deciding whether the tree-width of a graph is w, parameterized by w, is in FPL (by Bodlaender's theorem).

```
ModelCheck(Structure A, MSO-sentence φ)

 1  Compute STD (T, r, (ā_t)_{t∈T}) of A of width w := tw(A)
 2  q := qr(φ)
 3  Compute Λ_{q,w} and Π_{q,w}
 4  t := Type(r)
 5  if φ ∈ t then accept else reject.

Type(Tree-Node t)

 6  Compute part(t)
 7  if t is leaf
 8      then return(Λ_{q,w}(part(t)))
 9      else
10          u := first child of t; u' := second child of t
11          Compute is(u,t) and is(u',t)
12          return(Π_{q,w}(part(t), Type(u), is(u,t), Type(u'), is(u',t))).
```

Figure 1. MSO-model-checking

□

Courcelle proved his theorem for graphs and hypergraphs. He uses a version of MSO in which one is allowed to quantify not only over sets of vertices of a graph, but also over sets of edges. This version is clearly more expressive. However, there is a natural way of including it into our framework: We encode

graphs as *incidence structures* of vocabulary $\{V, W, I\}$ with unary V, W and binary I. With a graph $\boldsymbol{G} = (G, E^{\boldsymbol{G}})$ we associate an $\{V, W, I\}$-structure $I(\boldsymbol{G})$ whose universe is the disjoint union of the vertex set V^G and the edge set W^G in the obvious way. Then a class $\mathcal{C}$ of graphs is definable in Courcelle's MSO if, and only if, the class $\{I(\boldsymbol{G}) \mid \boldsymbol{G} \in \mathcal{C}\}$ is definable in our MSO.

It is a nice exercise to prove that for each graph $\boldsymbol{G}$ we have $\mathrm{tw}(\boldsymbol{G}) = \mathrm{tw}(I(\boldsymbol{G}))$ (and not only the trivial $\mathrm{tw}(I(\boldsymbol{G})) \leq \mathrm{tw}(\boldsymbol{G}) + \binom{\mathrm{tw}(\boldsymbol{G})+1}{2}$). Thus Theorem 4 is valid for both versions of MSO.

Arnborg, Lagergren and Seese [ALS91] proved extensions of Courcelle's Theorem for MSO-definable counting and optimization problems. Courcelle, Makowsky, and Rotics [CMR98] consider another, more liberal parameter of graphs called *clique-width*. They prove that if a graph comes with a clique-decomposition (the analogue of a tree-decomposition for clique-width) of bounded width, then MSO-model-checking is still possible in polynomial time. The problem with this approach is that it is not known if such a decomposition can be computed in polynomial time even for fixed clique-width 4.

However, there is not much room for extensions of Courcelle's theorem to other natural classes of structures. 3-Colorability is already NP-complete on planar graphs of valence 4 [GJS76]. This implies that, unless P=NP, for every class $\mathcal{C}$ of graphs that contains all planar graphs of valence 4 the problem $\mathrm{MC}(\mathcal{C}, \mathrm{MSO})$ is not in FPT.

Let us turn to FO-model-checking. With each τ-structure $\boldsymbol{A}$ we associate a graph $G(\boldsymbol{A})$, called the *Gaifman graph* of $\boldsymbol{A}$. The universe of $G(\boldsymbol{A})$ is A, and there is an edge between two distinct elements $a, b \in A$ if there is a relation $R \in \tau$ and a tuple $\bar{c} \in R^A$ such that both a and b occur in $\bar{c}$. The *valence* of a structure $\boldsymbol{A}$, denoted by $\mathrm{val}(\boldsymbol{A})$, is defined to be the valence of its Gaifman graph, that is, $\mathrm{val}(\boldsymbol{A}) := \max\big\{|\{b \mid E^{G(\boldsymbol{A})}(a, b)\}| \quad a \in A\big\}$.

Using Hanf's Sphere Theorem [Han65], Seese proved the following:

Theorem 5 ([See96]). *For every vocabulary τ, the following parameterized problem is in* FPL*:*

Input:	τ-structure $\boldsymbol{A}$, FO-sentence φ
Parameter:	$(\mathrm{val}(\boldsymbol{A}), \lvert\varphi\rvert)$
Problem:	Decide if $\boldsymbol{A} \models \varphi$.

A more general approach is based on Gaifman's locality theorem [Gai82]. The *distance* $d^{\boldsymbol{A}}(a, b)$ between two elements a, b of a structure $\boldsymbol{A}$ is the length of the shortest path between a and b in $G(\boldsymbol{A})$. The *r-neighborhood* of a is the set $N_r^{\boldsymbol{A}}(a) := \{b \in A \mid d^{\boldsymbol{A}}(a, b) \leq r\}$.

Definition 4. The *local tree-width* of a structure $\boldsymbol{A}$ is the function $\mathrm{ltw}(\boldsymbol{A}) : \mathbb{N} \to \mathbb{N}$ defined by $\mathrm{ltw}(\boldsymbol{A})(r) := \max\big\{\mathrm{tw}(\langle N_r(a)\rangle^{\boldsymbol{A}}) \mid a \in A\big\}$.

A class $\mathcal{C}$ of structures has *bounded local tree-width* if there is a function $\lambda : \mathbb{N} \to \mathbb{N}$ such that $\mathrm{ltw}(\boldsymbol{A})(r) \leq \lambda(r)$ for all $\boldsymbol{A} \in \mathcal{C}$, $r \in \mathbb{N}$.

Surprisingly, there are many natural examples of classes of graphs of bounded local tree-width, among them all classes of bounded tree-width, bounded valence, and bounded genus.[5]

Theorem 6 ([FG99b]). *Let $\mathcal{C}$ be a class of graphs of bounded local tree-width. Then the problem* $\mathrm{MC}(\mathcal{C}, \mathrm{FO})$ *is solvable in time $f(|\varphi|)n^2$ for a suitable f and thus in* FPT*.*

A refinement of bounded local tree-width is the notion of a *locally tree-decomposable* class of graphs; all the examples of classes of bounded local tree-width that we mentioned above are also locally tree-decomposable. For such classes the FO-model-checking is in FPL. This yields both Seese's Theorem 5 and the following result. Its proof also uses a theorem of Mohar [Moh96] stating that for every surface S there is a linear time algorithm deciding whether a given graph can be embedded into the orientable surface of genus g.

Theorem 7 ([FG99b]). *The following parameterized problem is in* FPL*:*

Input:	Graph $\boldsymbol{G}$, FO-sentence φ
Parameter:	$(\mathrm{genus}(\boldsymbol{G}), \lvert\varphi\rvert)$
Problem:	Decide if $\boldsymbol{G} \models \varphi$.

The final result of this section is based on deep graph theoretic results due to Robertson and Seymour [RS95,RS]. A *minor* of a graph $\boldsymbol{G}$ is a graph that is obtained from a subgraph of $\boldsymbol{G}$ by *contracting* edges.

Theorem 8 ([FG99a]). *Let $\mathcal{C}$ be a class of graphs such that there exists a graph that is not a minor of a graph in $\mathcal{C}$. Then* $\mathrm{MC}(\mathcal{C}, \mathrm{FO})$ *is in* FPT*.*

Robertson and Seymour proved that a class $\mathcal{C}$ of graphs has bounded tree-width if, and only if, there is a planar graph that is not a minor of a graph in $\mathcal{C}$ [RS86b]. Putting things together, we obtain a nice corollary:

Corollary 2. *Let $\mathcal{C}$ be a class of graphs that is closed under taking minors.*

(1) Assume that $\mathrm{P} \neq \mathrm{NP}$*. Then MC($\mathcal{C}$,* MSO*) is in* FPT *if, and only if, $\mathcal{C}$ has bounded tree-width.*
(2) Assume that $\mathrm{FPT} \neq \mathrm{W}[1]$*. Then MC($\mathcal{C}$,* FO*) is in* FPT *if, and only if, $\mathcal{C}$ is not the class of all graphs.*

[5] The genus of a graph $\boldsymbol{G}$ is the least genus of an orientable surface in which $\boldsymbol{G}$ is embeddable. Thus the graphs of genus 0 are just the planar graphs. A class of graphs has *bounded* tree-width/valence/genus if there is a constant bounding the respective number for all graphs in the class.

Remark 1. The results of this section show that on many classes of graphs the model-checking problem for MSO and FO can be solved in time $f(|\varphi|)||\boldsymbol{A}||^c$ for some function f and a constant c (usually $c = 1$).

We never specified f. In all results except Theorem 5, f is a tower of 2s whose height is roughly the number of quantifier-alternations in the input formula. This is basically due to the enormous number of MSO_q types and FO_q types. Even in Theorem 5, f is at least doubly exponential.

This makes the algorithms that can be derived from these results useless for practical applications. The benefit of the results is to provide a simple way to recognize a property as being linear time computable on certain classes of graphs (by expressing it in MSO or FO). Analyzing the combinatorics of the specific property then, one may also find a practical algorithm.

4 Tractable Cases II: Simple Formulas

This final section is devoted to tractable cases of the model-checking problems obtained by restricting the second part of the input, the sentence φ. The complexity of model-checking is known to be intimately linked to the number of variables in φ [Var95]. We might try to parameterize the FO-model-checking problem by the number of variables instead of the formula-length, but since the formula-length is an upper bound for the number of variables this makes the model-checking problem only harder (and thus even "more intractable").

However, if we fix the number of variables in advance we obtain not only fixed-parameter tractability, but actually polynomial-time computability: Vardi proved that the problem $\mathrm{MC}(\mathcal{F}[\tau], \mathrm{FO}^s)$ is solvable in time $O(n^s)$ [Var95]. Here FO^s (for $s \geq 1$) denotes the fragment of FO consisting of all formulas with at most s variables. Similar results hold for the finite-variable fragments of other logics, for example least fixed-point logic [Var95].

The finite-variable fragments of least fixed-point logic can also be used to give a descriptive characterization of the class FPT in the style of the well-known Immerman-Vardi Theorem [Imm86,Var82]. To formulate this, it is convenient to consider a parameterized problem as a class $P \subseteq \mathcal{O}[\tau] \times \mathbb{N}$ for some vocabulary τ, where $\mathcal{O}[\tau]$ denotes the class of ordered τ-structures. Then P is in FPT if, and only if, there is an $s \geq 1$ and a computable sequence $(\varphi_k)_{k\geq 1}$ of least-fixed point formulas with at most s variables and at most one fixed-point operator such that for all $k \geq 1$, φ_k defines the class $\{\boldsymbol{A} \mid (\boldsymbol{A}, k) \in P\}$ [FG99a].

The rest of this section is devoted to various refinements of model-checking for Σ_1-formulas. Remember that Σ_1-formulas are EFO-formulas in prenex normal form. Recall the definition of the structure $\boldsymbol{A}_\varphi$ associated with a formula φ (cf. Page 15). The tree-width of a formula φ is defined to be the tree-width of $\boldsymbol{A}_\varphi$. Kolaitis and Vardi [KV98] observed that, for all $s \geq 1$, every Σ_1-sentence of tree-width at most s can be effectively transformed into an equivalent existential FO^{s+1}-sentence. This implies:

Theorem 9 ([FG99a]). *Model-checking for Σ_1-sentences of tree-width at most s is in* FPT.

Note that this does not extend to arbitrary EFO-sentences, because for every $k \geq 3$ the EFO-sentence $\exists x_1 \dots \exists x_k \bigwedge_{1 \leq i < j \leq k} \exists y \exists z (y = x_i \wedge z = x_j \wedge E(y,z))$, saying that a graph contains a k-clique, has tree-width 3.

The following extension of Theorem 9 is based on a theorem of Plehn and Voigt [PV90] stating that for every $w \geq 1$ the following restriction of the subgraph isomorphism problem can be solved in time $f(|H|)|G|^{w+1}$, for a suitable function f:

Input:	Graphs $\boldsymbol{G}, \boldsymbol{H}$ with $\mathrm{tw}(\boldsymbol{H}) \leq w$
Parameter:	$\lvert H \rvert$
Problem:	Decide if $\boldsymbol{H}$ is isomorphic to a subgraph of $\boldsymbol{G}$.

Recall that negation symbols in an EFO-formula only occur in front of atomic subformulas. For a formula φ, we let $\varphi_{-\neq}$ be the result of deleting all subformulas of the form $\neg x = y$ from φ. The *modified tree-width* of φ is defined to be $\mathrm{tw}(\varphi_{-\neq})$.

Theorem 10 ([FG99a]). *Let $w \geq 1$ and Φ the set of all Σ_1-sentences of modified tree-width at most w. Then* $\mathrm{MC}(\mathcal{F}, \Phi)$ *is in* FPT.

Papadimitriou and Yannakakis [PY97] proved a special case of this theorem for so-called acyclic queries. They also observed that the problem $\mathrm{MC}(\mathcal{F}, \Phi)$ is NP-complete even for the class Φ of all Σ_1-sentences with modified tree-width 1. To see this, note that the existence of a hamiltonian path is subsumed by this model-checking problem.

Proof (of Theorem 10): It clearly suffices to prove the result for formulas of the form $\exists \bar{x} \theta(\bar{x})$, where θ is a conjunction of atomic and negated atomic formulas, because every Σ_1-formula φ can be effectively transformed to an equivalent disjunction of formulas of this form that contains precisely the same atomic formulas as φ and thus has the same modified tree-width.

So we have to find an algorithm that, given a τ-structure $\boldsymbol{A}$ and a sentence $\varphi = \exists x_1 \dots \exists x_k \theta(x_1, \dots, x_k) \in \Phi$ of modified tree-width at most w, where θ is a conjunction of atoms or negated atoms, decides if $\boldsymbol{A} \models \varphi$.

Our algorithm is based on the so-called *color coding* technique introduced by Alon, Yuster and Zwick [AYZ95]. We only present a Monte Carlo algorithm solving the problem in time $f(|\varphi|)|A|^{w+1}$. This algorithm can be derandomized using a k-perfect family of hash-functions ([SS90,AYZ95]).

A *coloring* of φ is a mapping $\gamma : \{1, \dots, k\} \to \{1, \dots, k\}$ such that $\gamma(i) \neq \gamma(j)$ if $\neg x_i = x_j$ occurs in θ. For each coloring γ we let φ_γ be the formula obtained by deleting literals $\neg x_i = x_j$ in θ and adding atoms $C_{\gamma(i)} x_i$ for $1 \leq i \leq k$. Note that $\mathrm{tw}(\varphi_\gamma) = \mathrm{tw}(\varphi_{-\neq}) = w$.

A *coloring* of $\boldsymbol{A}$ is a partition of $\boldsymbol{A}$ into sets $C_1^{\boldsymbol{A}}, \dots, C_k^{\boldsymbol{A}}$.

Observe that $\boldsymbol{A} \models \varphi$ if, and only if, there is a coloring γ of φ and a coloring $C_1^{\boldsymbol{A}}, \dots, C_k^{\boldsymbol{A}}$ of $\boldsymbol{A}$ such that $(\boldsymbol{A}, C_1^{\boldsymbol{A}}, \dots, C_k^{\boldsymbol{A}}) \models \varphi_\gamma$.

For all colorings γ of φ we do the following: We randomly and independently choose a color for each $a \in A$. Let $C_1^{\boldsymbol{A}}, \dots, C_k^{\boldsymbol{A}}$ be the resulting coloring.

Then we check if $(\boldsymbol{A}, C_1^{\boldsymbol{A}}, \ldots, C_k^{\boldsymbol{A}}) \models \varphi_\gamma$, which is possible in time $O(|\varphi||A|^{w+1})$ by Theorem 9. Note that if there is no coloring $C_1^{\boldsymbol{A}}, \ldots, C_k^{\boldsymbol{A}}$ of $\boldsymbol{A}$ such that $(\boldsymbol{A}, C_1^{\boldsymbol{A}}, \ldots, C_k^{\boldsymbol{A}}) \models \varphi_\gamma$, this will certainly not be the case. On the other hand, if there is such a coloring it will be the case with probability $\geq \frac{1}{k^k}$.

Repeating this procedure $f(k) = \lceil \log(2)k^k \rceil$ times, we will get the correct answer with probability at least $\frac{1}{2}$. □

Remark 2. The running time of the last algorithm can be considerably improved by only coloring, with the least possible number of colors, the set of those variables that actually appear in an inequality.

5 Open Problems

(1) (cf. Page 21) Do the W-hierarchy and the A-hierarchy coincide, or is at least W[2] = A[2]?
(2) (cf. Theorem 6) Let $\mathcal{C}$ be a class of graphs of bounded local tree-width. Is the problem MC($\mathcal{C}$, FO) in FPL?
(3) (cf. Theorem 8) For $k \geq 1$, let $\mathcal{K}_k$ be the class of graphs that do not contain the complete graph K_k as a minor. Note that for every class $\mathcal{C}$ of graphs such that there exists a graph that is not a minor of a graph in $\mathcal{C}$ there exists a $k \geq 1$ such that $\mathcal{C} \subseteq \mathcal{K}_k$. Is the following problem in FPT or even FPL?

Input:	Graph $\boldsymbol{G}$, FO-sentence φ
Parameter:	$(\min\{k \mid G \in \mathcal{K}_k\}, \lvert\varphi\rvert)$
Problem:	Decide if $\boldsymbol{G}$ satisfies φ.

(4) Let $\mathcal{W}$ be the class of words. Is there a constant $c \geq 1$ such that MC($\mathcal{W}$, FO) is solvable in time $O(2^{|\varphi|}n^c)$? If this is the case, how about the class of trees, planar graphs, et cetera?
(5) Let $\boldsymbol{G}_k$ denote the $(k \times k)$-grid. Is the following problem in FPT?

Input:	Graph $\boldsymbol{G}$
Parameter:	$k \in \mathbb{N}$
Problem:	Decide if there is a homomorphism $h : \boldsymbol{G}_k \to \boldsymbol{G}$.

A negative answer to this question would imply the following: *Let $\mathcal{C}$ be a class of graphs that is closed under taking minors and let Φ be the class of all Σ_1-formulas φ such that $\boldsymbol{A}_\varphi \in \mathcal{C}$. Then* MC($\mathcal{F}, \Phi$) *is in* FPT *if, and only if, $\mathcal{C}$ has bounded tree-width (cf. [FG99a]).*

Acknowledgements

I would like to thank Jörg Flum, Markus Frick, Phokion Kolaitis, and Wolfgang Thomas for helpful comments on an earlier version of this paper.

References

[ACP87] S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8:277–284, 1987.

[ALS91] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12:308–340, 1991.

[AYZ95] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM*, 42:844–856, 1995.

[Bod96] H.L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.

[Bod97] H.L. Bodlaender. Treewidth: Algorithmic techniques and results. In I. Privara and P. Ruzicka, editors, *Proceedings 22nd International Symposium on Mathematical Foundations of Computer Science, MFCS'97*, volume 1295 of *Lecture Notes in Computer Science*, pages 29–36. Springer-Verlag, 1997.

[CM77] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th ACM Symposium on Theory of Computing*, pages 77–90, 1977.

[CMR98] B. Courcelle, J.A. Makowsky, and U. Rotics. Linear time solvable optimization problems on graphs of bounded clique width. In *Graph Theoretic Concepts in Computer Science, WG'98*, volume 1517 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1998.

[Cou90] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 2, pages 194–242. Elsevier Science Publishers, 1990.

[DF95] R.G. Downey and M.R. Fellows. Fixed-parameter tractability and completeness II: On completeness for $W[1]$. *Theoretical Computer Science*, 141:109–131, 1995.

[DF99] R.G. Downey and M.R. Fellows. *Parametrized Complexity*. Springer-Verlag, 1999.

[DFR98] R.G. Downey, M.R. Fellows, and K. Regan. Descriptive complexity and the W-hierarchy. In P. Beame and S. Buss, editors, *Proof Complexity and Feasible Arithmetic*, volume 39 of *AMS-DIMACS Volume Series*, pages 119–134. AMS, 1998.

[DFT96] R.G. Downey, M.R. Fellows, and U. Taylor. The parametrized complexity of relational database queries and an improved characterization of $W[1]$. In Bridges, Calude, Gibbons, Reeves, and Witten, editors, *Combinatorics, Complexity, and Logic – Proceedings of DMTCS '96*, pages 194–213. Springer-Verlag, 1996.

[Die97] R. Diestel. *Graph Theory*. Springer-Verlag, 1997.

[FG99a] J. Flum and M. Grohe. Fixed-parameter tractability and logic. In preparation.

[FG99b] M. Frick and M. Grohe. Deciding first-order properties of locally tree-decomposable graphs. In J. Wiedermann, P. van Emde Boas, M. Nielsen. editors, *Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 331-340. Springer-Verlag, 1999.

[FV93] T. Feder and M.Y. Vardi. Monotone monadic SNP and constraint satisfaction. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 612–622, 1993.

[Gai82] H. Gaifman. On local and non-local properties. In *Proceedings of the Herbrand Symposium, Logic Colloquium '81*. North Holland, 1982.

[GJS76] M.R. Garey, D.S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[GL] M. Grohe and K. Luosto. Private communication.

[Hal76] R. Halin. S-Functions for graphs. *Journal of Geometry*, 8:171–186, 1976.

[Han65] W. Hanf. Model-theoretic methods in the study of elementary logic. In J. Addison, L. Henkin, and A. Tarski, editors, *The Theory of Models*, pages 132–145. North Holland, 1965.

[Imm86] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

[Kar72] R.M. Karp. Reducibilities among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[KV98] Ph.G. Kolaitis and M.Y. Vardi. Conjunctive-query containment and constraint satisfaction. In *Proceedings of the 17th ACM Symposium on Principles of Database Systems*, pages 205–213, 1998.

[Moh96] B. Mohar. Embedding graphs in an arbitrary surface in linear time. In *Proceedings of the 28th ACM Symposium on Theory of Computing*, pages 392–397, 1996.

[PV90] J. Plehn and B. Voigt. Finiding minimally weighted subgraphs. In R. Möhring, editor, *Graph-Theoretic Concepts in Computer Science, WG '90*, volume 484 of *Lecture Notes in Computer Science*, pages 18–29. Springer-Verlag, 1990.

[PY97] C.H. Papadimitriou and M. Yannakakis. On the complexity of database queries. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems*, pages 12–19, 1997.

[RS86a] N. Robertson and P.D. Seymour. Graph minors II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986.

[RS86b] N. Robertson and P.D. Seymour. Graph minors V. Excluding a planar graph. *Journal of Combinatorial Theory, Series B*, 41:92–114, 1986.

[RS95] N. Robertson and P.D. Seymour. Graph minors XIII. The disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63:65–110, 1995.

[RS] N. Robertson and P.D. Seymour. Graph minors XVI. Excluding a non-planar graph. *Journal of Combinatorial Theory, Series B*. To appear.

[See96] D. Seese. Linear time computable problems and first-order descriptions. *Mathematical Structures in Computer Science*, 6:505–526, 1996.

[SM73] L.J. Stockmeyer and A.R. Meyer. Word problems requiring exponential time. In *Proceedings of the 5th ACM Symposium on Theory of Computing*, pages 1–9, 1973.

[SS90] J.P. Schmidt and A. Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM Journal on Computing*, 19:775–786, 1990.

[Var82] M. Y. Vardi. The complexity of relational query languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pages 137–146, 1982.

[Var95] M. Y. Vardi. On the complexity of bounded-variable queries. In *Proceedings of the 14th ACM Symposium on Principles of Database Systems*, pages 266–276, 1995.

[Yan95] M. Yannakakis. Perspectives on database theory. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science*, pages 224–246, 1995.

Logical Specification of Operational Semantics *

Peter D. Mosses

BRICS and Department of Computer Science, University of Aarhus
Ny Munkegade, bldg. 540, DK-8000 Aarhus C, Denmark
Home page: http://www.brics.dk/ pdm/

Abstract. Various logic-based frameworks have been proposed for specifying the operational semantics of programming languages and concurrent systems, including inference systems in the styles advocated by Plotkin and by Kahn, Horn logic, equational specifications, reduction systems for evaluation contexts, rewriting logic, and tile logic.
We consider the relationship between these frameworks, and assess their respective merits and drawbacks—especially with regard to the modularity of specifications, which is a crucial feature for scaling up to practical applications. We also report on recent work towards the use of the Maude system (which provides an efficient implementation of rewriting logic) as a meta-tool for operational semantics.

1 Introduction

The designers, implementors, and users of a programming language all need to acquire an intrinsically *operational* understanding of its semantics. Programming language reference manuals attempt to provide such an understanding using informal, natural language; but they are prone to ambiguity, inconsistency, and incompleteness, and totally unsuitable as a basis for sound reasoning about the effects of executing programs—especially when concurrency is involved.

Various mathematical frameworks have been proposed for giving formal descriptions of programming language semantics. Denotational semantics generally tries to avoid direct reference to operational notions, and its abstract domain-theoretic basis remains somewhat inaccessible to most programmers (although modelling programs as higher-order functions has certainly given useful insight to language designers and to theoreticians). Operational semantics, which directly aims to model the program execution process, is generally based on familiar first-order notions; it has become quite popular, and has been preferred to denotational semantics for defining programming languages [28] and process algebras [26].

Despite the relative popularity of operational semantics, there have been some "semantic engineering" problems with scaling up to descriptions of full practical programming languages. A significant feature that facilitates scaling-up is good modularity: the formulation of the description of one construct should

* Written while visiting SRI International and Stanford University.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 32–49, 1999.

not depend on the presence (or absence) of other constructs in the language. Recently, the author has proposed a solution to the modularity problem for the structural approach to operational semantics [31,32].

There are different ways of specifying operational semantics for a programming language: an *interpreter* for programs—written in some (other) programming language, or defined mathematically as an abstract machine—is an *algorithmic* specification, determining *how* to execute programs; a *logic* for inferring judgements about program executions is a *declarative* specification, determining *what* program executions are allowed, but leaving how to find them to logical inference. Following Plotkin's seminal work [37], much interest has focussed on logical specification of operational semantics.

In fact various kinds of logic have been found useful for specifying operational semantics: arbitrary inference systems, natural deduction systems, Horn logic, equational logic, rewriting logic, and tile logic, among others. Sections 2 and 3 review and consider the relationship between these applied logics, pointing out some of their merits and drawbacks—especially with regard to the modularity of specifications. The brief descriptions of the various logics are supplemented by illustrative examples of their use. It is hoped that the survey thus provided will be useful as an introduction to the main techniques available for logical specification of operational semantics.

The inference of a program execution in some logic is clearly not the same thing as the inferred execution itself. Nevertheless, a system implementing logical inference may be used to execute programs according to their operational semantics. Section 4 reports on recent work towards the use of the Maude system (which provides an efficient implementation of rewriting logic) as a meta-tool for operational semantics.

2 Varieties of Structural Operational Semantics

The structural style of operational semantics (SOS) is to specify inference rules for *steps* (or transitions) that may be made not only by whole programs but also by their constituent phrases: expressions, statements, declarations, etc. The steps allowed for a compound phrase are generally determined by the steps allowed for its component phrases, i.e., the steps are defined inductively according the (abstract) syntax of the described programming language. An atomic assertion of the specified logic (such as $\gamma \longrightarrow \gamma$) asserts the possibility of a step from one *configuration* γ to another γ . Some configurations are usually distinguished as *terminal*, and have no further steps, whereas initial and intermediate configurations have phrases that remain to be executed as components.

Small-step SOS: In so-called small-step SOS [37], a single step for an atomic phrase often gives rise to a single step for its enclosing phrase (and thus ultimately for the whole program). A complete program execution is modelled as a succession—possibly infinite—of these small steps. During such a program execution, phrases whose execution has terminated get replaced by the values that they have computed.

Suppose that an abstract syntax for expressions e includes also values v:

$$e ::= v \mid \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \ldots$$
$$v ::= \mathit{true} \mid \mathit{false} \mid \ldots$$

Here are a couple of typical examples of inference rules for small-step SOS, to illustrate the above points:

$$\frac{e_0 \longrightarrow e_0}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2} \quad (1)$$

$$\texttt{if}\ \mathit{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_1 \qquad \texttt{if}\ \mathit{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_2 \quad (2)$$

In the lack of further rules for $\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, it is easy to see that the intended operational semantics has been specified: the sub-expression e_0 must be executed first, and if that execution terminates with a truth-value, only one of e_1, e_2 will then be executed.

Big-step SOS: In big-step SOS [17], a step for a phrase always corresponds to its entire (terminating) execution, so no iteration of steps is needed. A step for a compound phrase thus depends on steps for all those component phrases that have to be executed. (Big-step SOS has been dubbed *Natural Semantics* since the inference rules may resemble those of Natural Deduction proof systems [17].) Here is an example, where the notation $e \Downarrow v$ asserts the possibility of the evaluation (i.e., execution) of e terminating with value v:

$$\frac{e_0 \Downarrow \mathit{true} \qquad e_1 \Downarrow v_1}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v_1} \qquad \frac{e_0 \Downarrow \mathit{false} \qquad e_2 \Downarrow v_2}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v_2} \quad (3)$$

The intended operational semantics, where e_0 is supposed to be executed before e_1 or e_2, is not so evident here as it is in the small-step SOS rules. In other examples, however, explicit data dependencies may indicate the flow of control more clearly.

Big-step SOS cannot express the possibility of non-terminating executions, and thus it appears ill-suited to the description of reactive systems. However, the possibility of non-termination may be specified separately [8].

Note that small- and big-step SOS may be used together in the same description, e.g. big-step for modelling expression evaluation and small-step for modelling statement execution. Moreover, the transitive closure of the small-step relation (restricted to appropriate types of arguments) provides the big-step relation between phrases and their computed values.

Substitution: Binding constructs of programming languages, such as declarations and formal parameters, give rise to open phrases with free variables; however, these phrases do not get executed until the values to be bound to the free variables have actually been determined. Thus one possibility is to replace the free

variables by their values, producing a closed phrase, using a *substitution* operation (here written $[v/x]e$). However, the definition of substitution itself can be somewhat tedious—in practice, it is often left to the reader's imagination (as here):

$$\frac{e_1 \longrightarrow e_1}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow \texttt{let}\ x = e_1\ \texttt{in}\ e_2} \tag{4}$$

$$\texttt{let}\ x = v_1\ \texttt{in}\ e_2 \longrightarrow [v_1/x]e_2 \tag{5}$$

There is obviously no need to give a rule for evaluating a variable x to its value when using substitution.

Environments: An alternative approach, inspired by the treatment of binding constructs in denotational semantics and in Landin's work [18], is to use *environments* ρ: a judgement then has the form $\rho \vdash \gamma \longrightarrow \gamma$. In effect, the environment keeps track of the relevant substitutions that could have been made; the combination (often referred to as a *closure*) of an open phrase and an appropriate environment is obviously equivalent to a closed phrase. Environments are particularly simple to use in big-step SOS, but in small-step SOS, auxiliary syntax for explicit closures may have to be added to the described language (Plotkin managed to avoid adding auxiliary syntax in [37] only because the example language that he described already had a form of local declaration that was general enough to express closures). Here is the same example as described above, but now using environments instead of substitution:

$$\frac{\rho \vdash e_1 \longrightarrow e_1}{\rho \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow \texttt{let}\ x = e_1\ \texttt{in}\ e_2} \tag{6}$$

$$\frac{\rho[x \mapsto v] \vdash e_2 \longrightarrow e_2}{\rho \vdash \texttt{let}\ x = v_1\ \texttt{in}\ e_2 \longrightarrow \texttt{let}\ x = v_1\ \texttt{in}\ e_2} \tag{7}$$

$$\rho \vdash \texttt{let}\ x = v_1\ \texttt{in}\ v_2 \longrightarrow v_2 \tag{8}$$

Here, in contrast to when using substitution, a rule is needed for evaluating the use of a variable x occurring in an expression e:

$$\frac{\rho(x) = v}{\rho \vdash x \longrightarrow v} \tag{9}$$

The equation $\rho(x) = v$ above is formally regarded as a *side-condition* on the inference rule, although for notational convenience it is written as an antecedent of the rule. It restricts the conclusion of the rule to the case that the environment ρ does indeed provide a value v for x. Note that proofs of steps do not explicitly involve proofs of side-conditions.

Stores: For describing imperative programming languages, where the values last assigned to variables have to be kept for future reference, configurations are usually pairs of phrases and *stores*. Thus a judgement might have the form $\rho \vdash e, s \longrightarrow e\,, s\,$. Stores themselves are simply (finite) maps from locations to stored values. Unfortunately, adding stores to configurations invalidates all our previous rules, which should now be reformulated before being extended with rules for imperative phrases. For instance:

$$\frac{\rho \vdash e_1, s \longrightarrow e_1, s}{\rho \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2, s \longrightarrow \texttt{let}\ x = e_1\ \texttt{in}\ e_2, s} \tag{10}$$

$$\frac{\rho[x \mapsto v] \vdash e_2, s \longrightarrow e_2, s}{\rho \vdash \texttt{let}\ x = v_1\ \texttt{in}\ e_2, s \longrightarrow \texttt{let}\ x = v_1\ \texttt{in}\ e_2, s} \tag{11}$$

$$\rho \vdash \texttt{let}\ x = v_1\ \texttt{in}\ v_2, s \longrightarrow v_2, s \tag{12}$$

$$\frac{\rho(x) = v}{\rho \vdash x, s \longrightarrow v, s} \tag{13}$$

The need for this kind of reformulation reflects the poor inherent modularity of SOS. Later in this section, however, we shall see how the modularity of SOS can be significantly improved.

The following rules illustrate the SOS description of variable allocation, assignment, and dereferencing (assuming that locations l are not values v):

$$\frac{l \notin \mathrm{dom}(s)}{\rho \vdash \texttt{ref}\ v, s \longrightarrow l, s[l \mapsto v]} \tag{14}$$

$$\frac{l \in \mathrm{dom}(s)}{\rho \vdash l\ \texttt{:=}\ v, s \longrightarrow (\,), s[l \mapsto v]} \tag{15}$$

$$\frac{\rho(x) = l \qquad s(l) = v}{\rho \vdash x, s \longrightarrow v, s} \tag{16}$$

Conventions: A major example of an operational semantics of a programming language is the definition of Standard ML (SML) [28]. It is a big-step SOS, using environments and stores. A couple of "conventions" have been introduced to abbreviate the rules: one of them allows the store to be elided from configurations, relying on the flow of control to sequence assignments to variables; the other caters for raised exceptions preempting the normal sequence of evaluation of expressions. Although these conventions achieve a reasonable degree of conciseness, the need for them perhaps indicates that the big-step style of SOS has some pragmatic problems with scaling up to languages such as SML. Moreover, they make it difficult to exploit the definition of SML directly for verification or prototyping.

Recently, an alternative definition of SML has been proposed [15], without the need for the kind of conventions used in the original definition. SML is first

translated into an "internal language", which is itself defined by a (small-step) reduction semantics, see Sect. 3. (The translation of SML to the internal language is itself specified using a big-step SOS, but that aspect of the approach seems to be inessential.) A similar technique is used in the action semantics framework [29,30], where programs are mapped to an action notation that has already been defined using a (small-step) SOS.

Process Calculi: Small-step SOS is a particularly popular framework for the semantic description of calculi for concurrent processes, such as CCS. There, steps are generally *labelled*, and judgements have the form $\gamma \xrightarrow{\alpha} \gamma$. For CCS, labels α range over atomic "actions", and for each action l there is a complementary action $\bar{l}$ for synchronization; there is also an unobservable label τ, representing an internal synchronization. Here are some of the usual rules for CCS [26]:

$$\alpha.p \xrightarrow{\alpha} p \tag{17}$$

$$\frac{p_1 \xrightarrow{\alpha} p_1}{p_1 \mid p_2 \xrightarrow{\alpha} p_1 \mid p_2} \qquad \frac{p_2 \xrightarrow{\alpha} p_2}{p_1 \mid p_2 \xrightarrow{\alpha} p_1 \mid p_2} \tag{18}$$

$$\frac{p_1 \xrightarrow{l} p_1 \qquad p_2 \xrightarrow{\bar{l}} p_2}{p_1 \mid p_2 \xrightarrow{\tau} p_1 \mid p_2} \tag{19}$$

Also programming languages with constructs for concurrency, for instance Concurrent ML [40,39], can be described using small-step SOS. Unfortunately, the SOS description proposed for ML with concurrency primitives in [2] is not inductive in the syntax of the language, and the need to reformulate inference rules previously given for the purely functional part of the language is again a sign of the poor inherent modularity of the SOS framework. Also the more conventional SOS descriptions given in [12,16] have undesirably complex rules for the functional constructs.

Syntactic Congruence: When using SOS to describe process calculi, it is common practice to exploit a *syntactic congruence* on phrases, i.e., the syntax becomes a set of equivalence classes. For instance, the processes $p_1 \mid p_2$ and $p_2 \mid p_1$ might be identified, removing the need for one of the symmetric rules given in (18) above.

Evaluation to Committed Form: It is possible to describe the operational semantics of CCS and other concurrent calculi without labelling steps [35]. The idea is to give a big-step SOS for the evaluation of a process to its "committed forms" where the possible actions are apparent (cf. reduction to "head normal form" in the lambda-calculus). For example:

$$\frac{p_1 \Downarrow l.p_1 \qquad p_2 \Downarrow \bar{l}.p_2 \qquad p_1 \mid p_2 \Downarrow k}{p_1 \mid p_2 \Downarrow k} \tag{20}$$

The technique relies heavily on a syntactic congruence between processes.

Enhanced Operational Semantics: By labelling steps with their *proofs*, information about features such as causality and locality can be provided. This idea has been further developed and applied in the "enhanced" SOS style [9,38], where models taking account of different features of concurrent systems can be obtained by applying relabelling functions (extracting the relevant details from the proofs).

For a simple CCS-like process calculus, proofs may be constructed using tags $|_1$, $|_2$ to record the use of the rules that let processes act alone, and pairs $\langle |_1\, \theta_1, |_2\, \theta_2 \rangle$ to record synchronization. An auxiliary function l is used to extract actions from proofs. The following rules illustrate the form of judgements:

$$\frac{p_1 \xrightarrow{\theta} p_1}{p_1 \mid p_2 \xrightarrow{|_1\theta} p_1 \mid p_2} \qquad \frac{p_2 \xrightarrow{\theta} p_2}{p_1 \mid p_2 \xrightarrow{|_2\theta} p_1 \mid p_2} \tag{21}$$

$$\frac{p_1 \xrightarrow{\theta_1} p_1 \qquad p_2 \xrightarrow{\theta_2} p_2 \qquad l(\theta_1) = \overline{l(\theta_2)}}{p_1 \mid p_2 \xrightarrow{\langle |_1\theta_1, |_2\theta_2 \rangle} p_1 \mid p_2} \tag{22}$$

Despite its somewhat intricate notation, enhanced operational semantics provides a welcome uniformity and modularity for models of concurrent systems. By using substitution, the need for explicit environments can be avoided—but if one wanted to add stores, it seems that a major reformulation of the inference rules for steps would still be required. Or could labels be used also to record *changes* to stored values? The next variety of SOS considered here suggests that they can indeed.

Modular SOS: Recently, the author has proposed a solution to the SOS modularity problem [31,32]. In Modular SOS (MSOS) the transition rules for each construct are completely independent of the presence or absence of other constructs. When one extends or changes the described language, the description can be extended or changed accordingly, without reformulation—even though new kinds of information processing may be required.

The basic idea of MSOS is to incorporate *all* semantic entities as components of labels. Thus configurations are restricted to syntax and computed values, and judgements are *always* of the form $\gamma \xrightarrow{\alpha} \gamma$.

In fact the labels in MSOS are regarded as the arrows of a *category*, and the labels on adjacent steps have to be composable in that category. The labels are no longer the simple atomic actions often used in studies of process algebra, but usually have semantic entities—e.g. environments and stores—as components; so do the objects of the label category, which correspond to the states of the processed information.

Some basic label transformers for defining appropriate categories (starting from the trivial category) are available; they correspond to some of the simpler monad transformers used to obtain modularity in denotational semantics. Each label transformer adds a fresh indexed component to labels, and provides notation for setting and getting that component—independently of the presence or

absence of other components. By using variables α ranging over arbitrary labels, and ι ranging over arbitrary *identity* labels that remain in the same state, rules can be expressed independently of the presence or absence of irrelevant components of labels. For example:

$$\frac{e_0 \xrightarrow{\alpha} e_0}{\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \xrightarrow{\alpha} \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2} \tag{23}$$

$$\texttt{if } \mathit{true} \texttt{ then } e_1 \texttt{ else } e_2 \xrightarrow{\iota} e_1 \qquad \texttt{if } \mathit{false} \texttt{ then } e_1 \texttt{ else } e_2 \xrightarrow{\iota} e_2 \tag{24}$$

The above rules remain both valid and appropriate when the category of labels gets enriched with (e.g.) environment components, allowing the rules for binding constructs to be added:

$$\frac{e_1 \xrightarrow{\alpha} e_1}{\texttt{let } x = e_1 \texttt{ in } e_2 \xrightarrow{\alpha} \texttt{let } x = e_1 \texttt{ in } e_2} \tag{25}$$

$$\frac{\rho = \mathit{get}(\alpha, \mathit{env}) \qquad \alpha = \mathit{set}(\alpha, \mathit{env}, \rho[x \mapsto v]) \qquad e_2 \xrightarrow{\alpha'} e_2}{\texttt{let } x = v_1 \texttt{ in } e_2 \xrightarrow{\alpha} \texttt{let } x = v_1 \texttt{ in } e_2} \tag{26}$$

$$\texttt{let } x = v_1 \texttt{ in } v_2 \xrightarrow{\iota} v_2 \tag{27}$$

$$\frac{\rho = \mathit{get}(\iota, \mathit{env}) \qquad \rho(x) = v}{x \xrightarrow{\iota} v} \tag{28}$$

The use of ι rather than α above excludes the possibility of any change of state.

Axiomatic Specifications: For proof-theoretic reasoning about SOS descriptions—especially when establishing bisimulation and other forms of equivalence—it is convenient that steps can only occur when proved by just the *specified* inference rules. For other purposes, however, it may be an advantage to reformulate the inference rules of SOS as ordinary conditional formulae, i.e., *Horn clauses*, and use the familiar inference rules for deduction, such as *Modus Ponens*. The close correspondence between inference rules and Horn clauses has been used in the implementation of big-step SOS by compilation to Prolog [17].

The axiomatic reformulation of SOS requires side-conditions on rules to be treated as ordinary conditions, along with judgements about possible steps. It has been adopted in the SMoLCS framework [1], which combines SOS with algebraic specifications. It has also been exploited in the modular SOS of action notation [34], where CASL, the Common Algebraic Specification Language [7], is used throughout (CASL allows the declaration of total and partial functions, relations, and subsorts, which is just what is needed in the side-conditions of SOS descriptions).

3 Varieties of Reduction Semantics

Many of the inference rules specified in the structural approach to operational semantics merely express that execution steps of particular components give rise to execution steps of the enclosing phrases. The notion of *reduction* in term rewriting systems enjoys a similar property, except that there is *a priori* no restriction on the order in which component phrases are to be reduced. Thus for any term constructor f, the following inference rule may be specified for the reduction relation $t \longrightarrow t$:

$$\frac{t_i \longrightarrow t_i}{f(t_1, \ldots, t_i, \ldots, t_n) \longrightarrow f(t_1, \ldots, t_i, \ldots, t_n)} \tag{29}$$

The above rule is subsumed by the following somewhat more elegant rule, where C ranges over arbitrary one-hole term *contexts*, and $C[t]$ is the term obtained by filling the unique hole in the context C with the term t:

$$\frac{t \longrightarrow t}{C[t] \longrightarrow C[t\,]} \tag{30}$$

It is straightforward to define the arbitrary one-hole contexts for any ranked alphabet of (constant and) function symbols; similarly for many-sorted and order-sorted signatures—introducing a different sort of context for each *pair* of argument and result sorts.

Several frameworks for operational semantics are based on variations of the basic notion of reduction, and are reviewed below.

Reduction Strategies: The problem with using ordinary reduction to specify operational semantics is the lack of control concerning the order of reduction steps: the entire sequence of reductions might be applied to a part of the program that in fact should not be executed at all.

For instance, consider the λ-expressions with constants, which may be regarded as a simple functional programming language:

$$\begin{aligned} e &::= v \mid e_1\ e_2 \\ v &::= b \mid f \mid x \mid \lambda x.e \end{aligned}$$

where the basic constants b and function constants f are left unspecified. The execution steps for evaluating λ-expressions are δ-reductions, concerned with applications of the form $f\ b$, and β-reductions:

$$(\lambda x.e)(e\ ) \longrightarrow [e\ /x]e \tag{31}$$

where the substitution of expressions e for variables x, written $[e\ /x]e$, is assumed to avoid capture of free variables. An expression such as

$$(\lambda y.b)((\lambda x.xx)(\lambda x.xx))$$

has both terminating and non-terminating reduction sequences: the one that takes the leftmost, outermost β-reduction corresponds to "call-by-name" semantics for λ-expressions; that which always applies β-reduction to $(\lambda x.xx)(\lambda x.xx)$ corresponds to "call-by-value" semantics.

Standard reduction sequences are those which always make the leftmost outermost reduction at each step. For λ-expressions, restricting reductions to standard δ- and β-reductions ensures call-by-name operational semantics. Remarkably, also call-by-value semantics can be ensured by restricting to standard reductions—provided that the β-reduction rule is itself restricted to the case where the argument of the application is already a value v [36,11]:

$$(\lambda x.e)(v) \longrightarrow [v/x]e \tag{32}$$

Standard reduction sequences with this restricted notion of β-reduction correspond to an operational semantics for λ-expressions defined by an SECD machine [36].

By adopting the restriction to standard reductions, it might be possible to give reduction semantics for other programming languages. However, the following technique not only subsumes this approach, but also has the advantage of admitting an explanation in terms of inference systems.

Evaluation Contexts: An alternative way of controlling the applicability of reductions is to require them to occur in *evaluation contexts* [10]. It is convenient to specify evaluation contexts E in the same way as the abstract syntax of programs, using context-free grammars. The symbol [ ] represents the hole of the context; the grammar must ensure that exactly one hole occurs in any evaluation context.

The restriction to evaluation contexts corresponds to simply replacing the general context rule for reduction (30) above by:

$$\frac{t \longrightarrow t}{E[t] \longrightarrow E[t\,]} \tag{33}$$

For example, to obtain the call-by-value semantics of λ-expressions, let evaluation contexts E be defined by the grammar:

$$E ::= [\,] \mid v\ E \mid E\ e$$

It is easy to see that when an expression is of the form $E[e_1\ e_2]$, a standard reduction step can only reduce $e_1\ e_2$ or some sub-expression of it. Similarly, it appears that call-by-name semantics can be obtained by letting evaluation contexts be defined as follows:

$$E ::= [\,] \mid f\ E \mid E\ e$$

(where f ranges over function constants).

The following specification of evaluation contexts would be appropriate for the intended operational semantics of the illustrative language constructs considered in Sect. 2:

$$E ::= [\,] \mid \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ x = E\ \texttt{in}\ e_2$$

(where the grammar for expressions e and values v is as before). Notice the close correspondence between the productions of the above grammar and the previously-given small-step SOS rules (1) and (4). The above grammar is clearly more concise than the inference rules for expressing the allowed order of execution; this economy of specification may account for at least some of the popularity of the evaluation context approach.

Assuming that reductions are restricted to occur only in evaluation contexts, the following rules may now be given:

$$\texttt{if}\ \mathit{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_1 \qquad \texttt{if}\ \mathit{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_2 \tag{34}$$

$$\texttt{let}\ x = v_1\ \texttt{in}\ e_2 \longrightarrow [v_1/x]e_2 \tag{35}$$

where $[v/x]e$ is substitution, as before. As the reader may have noticed, these are exactly the same as the small-step SOS rules (2) and (5).

An alternative technique with evaluation contexts is to combine (30) above with the reduction rules themselves—now insisting that reductions are always applied to the entire program. With the same definition of evaluation contexts, the above reduction rules would then be written:

$$E[\texttt{if}\ \mathit{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2] \longrightarrow E[e_1] \tag{36}$$

$$E[\texttt{if}\ \mathit{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2] \longrightarrow E[e_2] \tag{37}$$

$$E[\texttt{let}\ x = v_1\ \texttt{in}\ e_2] \longrightarrow E[[v_1/x]e_2] \tag{38}$$

This seemingly innocent reformulation in fact provides a significant new possibility, which is perhaps the *forte* of the evaluation context approach: reductions may depend on and/or change the structure of the context itself. For example, we may easily add a construct that is intended to stop the execution of the entire program, and specify the reduction:

$$E[\texttt{stop}] \longrightarrow \texttt{stop} \tag{39}$$

To specify the same semantics in small-step semantics would require giving an explicit basic rule for the propagation of `stop` out of each evaluation context construct:

$$\texttt{if stop then}\ e_1\ \texttt{else}\ e_2 \longrightarrow \texttt{stop} \tag{40}$$

$$\texttt{let}\ x = \texttt{stop in}\ e_2 \longrightarrow \texttt{stop} \tag{41}$$

In a big-step semantics, one would have to provide extra inference rules, e.g.:

$$\frac{e_0 \Downarrow \texttt{stop}}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow \texttt{stop}} \tag{42}$$

Moreover, evaluation contexts can be used to specify the operational semantics of advanced control constructs, such as those manipulating *continuations* [11]. Although it may be possible to specify continuations in SOS, the appropriateness of the use of evaluation contexts here cannot be denied.

An evaluation context may contain more than just the syntactic control context: for instance, it may also contain a store, recording the values assigned to variables. A store s is represented syntactically as a sequence of pairs of locations l and values v, with no location occurring more than once. It is quite straightforward to give reduction rules for variable allocation, assignment, and dereferencing [4]:

$$s\ E[\texttt{ref}\ v] \longrightarrow s, (l, v)\ E[l] \quad \text{if } l \text{ is not used in } s \tag{43}$$

$$s, (l, v), s\ \ E[l\ \texttt{:=}\ v\ ] \longrightarrow s, (l, v\ ), s\ \ E[(\ )] \tag{44}$$

$$s, (l, v), s\ \ E[l] \longrightarrow s, (l, v), s\ \ E[v] \tag{45}$$

The previously given rules for functional constructs using explicit evaluation contexts (e.g. (36)) remain valid, so the modularity of the approach appears to be good—also when adding concurrency primitives to a functional language, as illustrated in [39,40]. However, it appears that it would not be so straightforward to add explicit environments to evaluation contexts, and the reliance on syntactic substitution may complicate the description of languages with "dynamic" scope rules.

One significant potential problem when using evaluation contexts for modelling the operational semantics of concurrent languages is how to define and prove equivalence of processes. In particular cases "barbed" bisimulation can be defined [27,41]; also, a rather general technique for extracting labelled transition systems (and hence bisimulations) from evaluation context semantics has been proposed [42].

Rewriting Logic: The framework of Rewriting Logic (RL) [21] generalizes conventional term rewriting in two main directions:

- rewriting may be *modulo* a set of equations between terms (i.e., it applies to arbitrary equationally-specified data structures); and
- rewriting may be *concurrent* (i.e., non-overlapping sub-terms may be rewritten simultaneously).

Moreover, no assumptions about confluence or termination are made: the rules are understood not as equations, but as transitions.

The inference rules for RL are as follows, where rewriting from between equivalence classes of terms is written $[t] \longrightarrow [t\ ]$. Rewriting is taken to be reflexive:

$$[t] \longrightarrow [t] \tag{46}$$

which allows one or more of the arguments to remain the same in concurrent rewriting:

$$\frac{[t_1] \longrightarrow [t_1] \quad \ldots \quad [t_n] \longrightarrow [t_n]}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t_1, \ldots, t_n)]} \tag{47}$$

The following inference rule combines replacement of variables $x_1, \ldots, x_m$ by terms $t_1, \ldots, t_m$ in a specified rule $r : [t(x_1, \ldots, x_m)] \longrightarrow [t\ (x_1, \ldots, x_m)]$ with the possibility of rewriting the terms in the same step:

$$\frac{[t_1] \longrightarrow [t_1] \quad \ldots \quad [t_n] \longrightarrow [t_n]}{[t(t_1/x_1, \ldots, t_n/x_n)] \longrightarrow [t\ (t_1/x_1, \ldots, t_n/x_n)]} \tag{48}$$

Finally, rewriting is taken to be transitive:

$$\frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]} \tag{49}$$

Specified rewriting rules are also allowed to be conditional, which requires a further inference rule for discharging conditions.

RL has been used as a unifying model for concurrency [23] and as a logical framework [20]. It has also been proposed as a *semantic* framework, as an alternative to frameworks such as SOS [20]. RL has been efficiently implemented in the Maude system [5], which makes its use as a semantic framework particularly attractive in connection with the possibilities for prototyping semantic descriptions (see Sect. 4).

Two techniques for expressing SOS descriptions in RL have been proposed [20]. The first is a special case of a general technique for representing sequent systems in unconditional RL, with the rewriting relation corresponding to provability; the second is more specific to SOS, and uses conditional rewriting rules. Let us illustrate both techniques with the same example: the SOS rules (17)–(19) for concurrency in CCS (Sect. 2).

To start with, term constructors for the abstract syntax of processes and labels are needed; we shall only make use of the binary process constructor $p \mid p$, which is now specified to be both associative and commutative (corresponding to a syntactic congruence in SOS).

For the first technique, we also introduce a term constructor $S(p, \alpha, p\ )$ representing the assertion of an SOS step from process p to process p with label α; and an infix term constructor $s_1 \& s_2$ representing the conjunction of such assertions, specified to be associative, commutative, with unit $\bot$.

The SOS rules are then expressed in RL as follows:

$$[\bot] \longrightarrow [S(\alpha.p, \alpha, p)] \tag{50}$$

$$[S(p_1, \alpha, p_1)] \longrightarrow [S(p_1 \mid p_2, \alpha, p_1 \mid p_2)] \tag{51}$$

$$[S(p_1, l, p_1) \& S(p_2, \bar{l}, p_2)] \longrightarrow [S(p_1 \mid p_2, \tau, p_1 \mid p_2)] \tag{52}$$

The relationship between the SOS steps and the rewriting relation is that $p \xrightarrow{\alpha} p$ in the SOS iff $[\bot] \longrightarrow [S(p, \alpha, p\ )]$ is provable in RL. Note that the rewriting relation is highly non-deterministic, and in practice a goal-directed strategy would be needed in order to use the Maude implementation of RL for proving $[\bot] \longrightarrow [S(p, \alpha, p\ )]$ for some particular process p.

For the second technique, we introduce only a term constructor $\alpha; p$ that combines the label α with the process p, representing the "result" of some SOS step. The result sort of $\alpha; p$ is regarded as a supersort of the sort of processes, and rewriting is always of the form $[p] \longrightarrow [\alpha; p\,]$, i.e. it is sort-increasing. The SOS rules are then expressed in RL as follows:

$$[\alpha.p] \longrightarrow [\alpha; p] \tag{53}$$

$$[p_1 \mid p_2] \longrightarrow [\alpha; p_1 \mid p_2] \;\text{ if }\; [p_1] \longrightarrow [\alpha; p_1] \tag{54}$$

$$[p_1] \mid p_2] \longrightarrow [\tau; p_1 \mid p_2] \;\text{ if }\; [p_1] \longrightarrow [l; p_1] \wedge [p_2] \longrightarrow [\bar{l}; p_2] \tag{55}$$

The relationship between the SOS steps and the rewriting relation is now that given a process p, there are processes $p_1, \ldots, p_{n-1}$ and labels $\alpha_1, \ldots, \alpha_n$ such that $p \xrightarrow{\alpha_1} p_1 \xrightarrow{\alpha_1} \ldots p_{n-1} \xrightarrow{\alpha_n} p$ in SOS iff $[p] \longrightarrow [\alpha_1; \ldots; \alpha_n; p\,]$ in RL.

Tile Logic: Although Tile Logic (TL) is listed here together with other frameworks for reduction semantics, due to its close relationship with Rewriting Logic (translations both ways between the two frameworks have been provided [25,3]), it could just as well have been classified as a structural framework. In fact it is a development of so-called *context systems* [19] where steps are specified much as in SOS, except that phrases may be contexts with multiple holes, and the actions that label the steps (the *effects*) may depend on actions to be provided by the holes (the *triggers*). A context may be thought of as an m-tuple of terms in n variables; there are operations, familiar from Lawvere's algebraic theories, for composing contexts sequentially (plugging the terms of one context into the holes of another) and in parallel (concatenating tuples of terms over the same variables), together with units and projections.

In TL, steps may affect the interfaces of contexts, and the steps themselves have a rich algebraic structure; see [13,14] for the details. Here we shall merely introduce the notation of TL, and illustrate its use to express the operational semantics of a familiar fragment of CCS.

The conventional algebraic notation for a tile is $s \xrightarrow[b]{a} t$ (ignoring the label of the tile, for simplicity), where $s \longrightarrow t$ is a context rewrite step, a is the trigger of the step, and b is its effect. The tile requires that the variables of s are rewritten with a cumulative effect a.

For appropriate arguments, sequential composition of contexts is written $s; t$, with unit id, and parallel composition is written as $s \otimes t$. Duplicators are written ∇, and dischargers (or sinks) as ! (permuters are also provided). The operations satisfy all the axioms that one might expect.

Two tiles can be composed in parallel (using $\otimes$), vertically (using $\cdot$), or horizontally (using $*$), provided that their components have the appropriate types.

Finally, here are the tiles for some CCS constructs (where the variables x_1, x_2 are actually redundant, and are usually omitted):

$$\alpha.x_1 \xrightarrow[\alpha]{id} x_1 \tag{56}$$

The above rule may be read operationally as: the context prefixes the hole x_1 with the action α, and may become just the hole, emitting α as effect, without any trigger.

$$x_1 \mid x_2 \overset{\alpha \otimes id}{\underset{\alpha}{\longrightarrow}} x_1 \mid x_2 \tag{57}$$

$$x_1 \mid x_2 \overset{id \otimes \alpha}{\underset{\alpha}{\longrightarrow}} x_1 \mid x_2 \tag{58}$$

$$x_1 \mid x_2 \overset{\alpha \otimes \bar{\alpha}}{\underset{\tau}{\longrightarrow}} x_1 \mid x_2 \tag{59}$$

Of course, this simple kind of SOS example does not nearly exploit the full generality of the Tile Logic framework, which encompasses graph rewriting as well as rewriting logic and context systems.

4 Prototyping

The Maude implementation of Rewriting Logic (RL) [5] has several features that make it particularly attractive to use for prototyping operational semantics of programming languages. For instance, it provides meta-level functions for parsing, controlled rewriting, and pretty-printing; moreover, the Maude rewriting engine is highly efficient. Maude also supports Membership Algebra [24], which is an expressive framework for order-sorted algebraic specification.

Together with Christiano Braga at SRI International, the author has recently been developing a representation of Modular SOS (MSOS) [31,32] in RL and implementing it in Maude; this involved first extending Maude with a new kind of conditional rule, using the Maude meta-level. (Presently, MSOS rules are translated manually to Maude rules, but later the translation is itself to be implemented using Maude meta-level facilities.)

The translation process transforms an MSOS specification into an SOS-like one [32]. MSOS rules are translated into Maude rules over configurations that have a syntactic and a semantic component. Label formulae are translated into equations dealing with the associated states. The MSOS of Action Notation [34] is being prototyped this way—when completely implemented, together with further meta-level functions for processing descriptions formulated in Action Semantics [29,30], it should enable the prototyping of action-semantic descriptions of programming languages.

5 Conclusion

It is hoped that this survey of frameworks for the logical specification of operational semantics has provided a useful overview of much of the work in this area (apologies to those whose favourite frameworks have been omitted). It would be unwise to try to draw any definite conclusions on the basis of the remarks made here about the various frameworks: both the SOS and the reduction semantics approaches have their strengths, and are currently active areas of research—as is Tile Logic, which is strongly related to the structural approach, as well as to

Rewriting Logic. However, it is clear that logic has been found to be a particularly useful tool for specifying operational semantics, and appears to be preferred in practice to approaches based on abstract machines and interpreters.

Acknowledgements: Thanks to Christiano Braga, José Meseguer, and Carolyn Talcott for comments on a draft of this paper, and to José Meseguer for helpful discussions and support throughout the author's visit to SRI International. During the preparation of this paper the author was supported by BRICS (Centre for Basic Research in Computer Science), established by the Danish National Research Foundation in collaboration with the Universities of Aarhus and Aalborg, Denmark; by an International Fellowship from SRI International; and by DARPA-ITO through NASA-Ames contract NAS2-98073.

References

[1] E. Astesiano and G. Reggio. SMoLCS driven concurrent calculi. In *TAPSOFT'87, Proc. Int. Joint Conf. on Theory and Practice of Software Development, Pisa*, volume 249 of *LNCS*. Springer-Verlag, 1987.

[2] D. Berry, R. Milner, and D. N. Turner. A semantics for ML concurrency primitives. In *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, pages 119–129. ACM, 1992.

[3] R. Bruni, J. Meseguer, and U. Montanari. Process and term tile logic. Technical Report SRI-CSL-98-06, Computer Science Lab., SRI International, July 1998.

[4] R. Cartwright and M. Felleisen. Extensible denotational language specifications. In M. Hagiya and J. C. Mitchell, editors, *TACS'94, Symposium on Theoretical Aspects of Computer Software*, volume 789 of *LNCS*, pages 244–272, Sendai, Japan, 1994. Springer-Verlag.

[5] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [22], pages 65–89.

[6] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible by WWW[1] and FTP[2].

[7] CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [6], Oct. 1998.

[8] P. Cousot and R. Cousot. Inductive definitions, semantics, and abstract interpretation. In *Proc. POPL'92*, pages 83–94. ACM, 1992.

[9] P. Degano and C. Priami. Enhanced operational semantics. *ACM Computing Surveys*, 28(2):352–354, June 1996.

[10] M. Felleisen and D. P. Friedman. Control operators, the SECD machine, and the λ-calculus. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 193–217. North-Holland, 1987.

[11] M. Felleisen and R. Hieb. The reveised report on the syntactic theories of sequential control and state. *Theoretical Comput. Sci.*, 102:235–271, 1992.

[12] W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisumulation for core CML. *J. Functional Programming*, 8(5):447–451, 1998.

[1] http://www.brics.dk/Projects/CoFI

[2] ftp://ftp.brics.dk/Projects/CoFI

[13] F. Gadducci and U. Montanari. Tiles, rewriting rules, and CCS. In Meseguer [22].

[14] F. Gadducci and U. Montanari. The tile model. In *Proof, Language, and Interaction*. The MIT Press, 1999. To appear.

[15] R. Harper and C. Stone. A type-theoretic interpretation of Standard ML. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Robin Milner Festschrifft*. MIT Press, 1998. To appear.

[16] A. S. A. Jeffrey. Semantics for core concurrent ML using computation types. In A. D. Gordon and A. J. Pitts, editors, *Higher Order Operational Techniques in Semantics, Proceedings*. Cambridge University Press, 1997.

[17] G. Kahn. Natural semantics. In *STACS'87, Proc. Symp. on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.

[18] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[19] K. G. Larsen and L. Xinxin. Compositionality through operational semantics of contexts. In *Proc. ICALP'90*, volume 443 of *LNCS*, pages 526–539. Springer-Verlag, 1990.

[20] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay, editor, *Handbook of Philosophical Logic*, volume 6. Kluwer Academic Publishers, 1998. 'Also Technical Report SRI-CSL-93-05, SRI International, August 1993.

[21] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[22] J. Meseguer, editor. *First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996.

[23] J. Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In U. Montanari and V. Sassone, editors, *Proc. CONCUR'96*, volume 1119 of *LNCS*, pages 331–372. Springer-Verlag, 1996.

[24] J. Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *LNCS*, pages 18–61. Springer-Verlag, 1998.

[25] J. Meseguer and U. Montanari. Mapping tile logic into rewriting logic. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *LNCS*. Springer-Verlag, 1998.

[26] R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume B, chapter 19. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[27] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. ICALP'92*, volume 623 of *LNCS*, pages 685–695. Springer-Verlag, 1992.

[28] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[29] P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[30] P. D. Mosses. Theory and practice of action semantics. In *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, volume 1113 of *LNCS*, pages 37–61. Springer-Verlag, 1996.

[31] P. D. Mosses. Semantics, modularity, and rewriting logic. In C. Kirchner and H. Kirchner, editors, *WRLA'98, Proc. 2nd Int. Workshop on Rewriting Logic and its Applications, Pont-à-Mousson, France*, volume 15 of *Electronic Notes in Theoretical Computer Science*, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[32] P. D. Mosses. Foundations of Modular SOS (extended abstract). In *MFCS'99, Proc. 24th Intl. Symp. on Mathematical Foundations of Computer Science, Szklarska-Poreba, Poland*, LNCS. Springer-Verlag, 1999. To appear.

[33] P. D. Mosses. A modular SOS for Action Notation. Research series, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. To appear. Draft available from `http://www.brics.dk/ {}pdm/`.

[34] P. D. Mosses. A modular SOS for Action Notation (extended abstract). Number NS-99-3 in Notes Series, BRICS, Dept. of Computer Science, Univ. of Aarhus, May 1999. Full version available [33].

[35] A. M. Pitts and J. R. X. Ross. Process calculus based upon evaluation to committed form. In U. Montanari and V. Sassone, editors, *Proc. CONCUR'96*, volume 1119 of *LNCS*. Springer-Verlag, 1996.

[36] G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Comput. Sci.*, 1:125–159, 1975.

[37] G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Dept. of Computer Science, Univ. of Aarhus, 1981.

[38] C. Priami. *Enhanced Operational Semantics for Concurrency*. PhD thesis, Dipartimento di Informatica, Università degli Studi di Pisa, 1996.

[39] J. H. Reppy. CML: A higher-order concurrent language. In *Proc. SIGPLAN'91, Conf. on Prog. Lang. Design and Impl.*, pages 293–305. ACM, 1991.

[40] J. H. Reppy. *Higher-Order Concurrency*. PhD thesis, Computer Science Dept., Cornell Univ., 1992. Tech. Rep. TR 92-1285.

[41] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Dept. of Computer Science, Univ. of Edinburgh, 1992.

[42] P. Sewell. From rewrite to bisimulation congruences. In *Proc. CONCUR'98*, volume 1466 of *LNCS*, pages 269–284. Springer-Verlag, 1998.

Constraint-Based Analysis of Broadcast Protocols

Giorgio Delzanno[1], Javier Esparza[2], and Andreas Podelski[1]

[1] Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany
e-mail: {delzanno | podelski}@mpi-sb.mpg.de
[2] Technische Universität Munchen, Arcisstr. 21, 80290 München, Germany
esparza@informatik.tu-muenchen.de

Abstract. Broadcast protocols are systems composed of a finite but arbitrarily large number of processes that communicate by rendezvous (two processes exchange a message) or by broadcast (a process sends a message to all other processes). The paper describes an optimized algorithm for the automatic verification of safety properties in broadcast protocols. The algorithm checks whether a property holds for any number of processes.

1 Introduction

Broadcast protocols [EN98] are systems composed of a finite but arbitrarily large number of processes that communicate by rendezvous (two processes exchange a message) or by broadcast (a process sends a message to all other processes). They are a natural model for problems involving readers and writers, such as cache-coherence problems.

From a mathematical point of view, broadcast protocols can be regarded as an extension of vector addition systems or Petri nets. Their operational semantics is a transition system whose states are tuples of integers. Moves between transitions are determined by a finite set of affine transformations with guards. Vector Addition Systems correspond to the particular case in which the matrix of the affine transformation is the identity matrix.

In [EFM99], Esparza, Finkel and Mayr show that the problem of deciding whether a broadcast protocol satisfies a safety property can be reduced to a special reachability problem, and using results by Abdulla *et al.*, [ACJ$^+$96] (see also [FS98]), they prove that this problem is decidable. They propose an abstract algorithm working on infinite sets of states. The algorithm starts with the set of states to be reached, and repeatedly adds to it the set of its immediate predecessors until a fixpoint is reached.

As shown e.g. in [Kin99,DP99], linear arithmetic constraints can be used to finitely represent infinite sets of states in integer valued systems. Symbolic model checking algorithms can be defined using the 'satisfiability' and the 'entailment' test to symbolically compute the transitive closure of the *predecessor* relation defined over sets of states. However, in order to obtain an efficient algorithm it is crucial to choose the right format for the constraints.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 50–66, 1999.

In this paper we discuss different classes of constraints, and propose *linear constraints with disjoint variables* as a very suitable class for broadcast protocols. We show that the operations of computing the immediate predecessors and checking if the fixpoint has been reached can both be efficiently implemented. We also propose a compact data structure for these constraints.

We have implemented a specialized checker based on our ideas, and used it to define a symbolic model checking procedure for broadcast protocols. As expected, the solver leads to a significant speed-up with respect to procedures using general purpose constraint solvers (HyTech [HHW97] and Bultan, Gerber and Pugh's model checker based on the Omega library [BGP97]). We present some experimental results for both broadcast protocols and weighted Petri Nets.

2 Broadcast Protocols: Syntax and Semantics

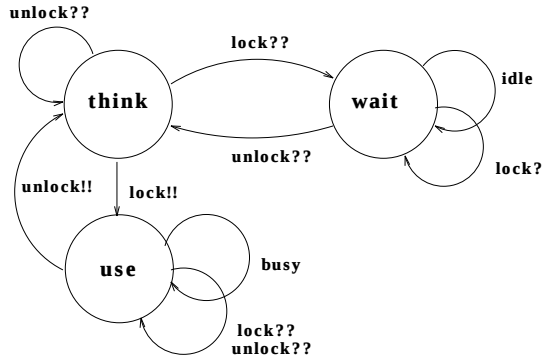
2.1 Syntax

A *broadcast protocol* is a triple (S, L, R) where

- S is a finite set of *states*.
- L is a set of *labels*, composed of a set Σ_l of *local* labels, two sets $\Sigma_r \times \{?\}$ and $\Sigma_r \times \{!\}$ of *input* and *output rendez-vous* labels, and two sets $\Sigma_b \times \{??\}$ and $\Sigma_b \times \{!!\}$ of *input* and *output broadcast* labels, where $\Sigma_l, \Sigma_r, \Sigma_b$ are disjoint finite sets. The elements of $\Sigma = \Sigma_l \cup \Sigma_r \cup \Sigma_b$ are called *actions*.
- $R \subseteq S \times L \times S$ is a set of *transitions* satisfying the following property: for every $a \in \Sigma_b$ and every state $s \in S$, there exists a state $s' \in S$ such that $s \xrightarrow{a??} s'$. Intuitively, this condition guarantees that a process is always willing to receive a broadcasted message.

We denote $(s, l, s) \in R$ by $s \xrightarrow{l} s'$. The letters $a, b, c, \ldots$ denote actions. Rendezvous and broadcast labels like $(a, ?)$ or $(b, !!)$ are shortened to $a?$ and $b!!$. We restrict our attention to broadcast protocols satisfying the following additional conditions: (i) for each state s and each broadcast label $a??$ there is exactly one state s' such that $s \xrightarrow{a??} s'$ (determinism); (ii) each label of the form a, $a!$, $a?$ and $a!!$ appears in exactly one transition.

Consider the following example:



The finite-state automata in the figure models the behaviour of a system of identical processes that race for using a shared resource. Initially, all processes are in the state **think**. Before accessing its own critical section, a process broadcasts the request `lock!!`. In reply to the broadcast (`lock??`) the remaining processes are forced to move to the state **wait** (an abstraction of a queue). After using the resource, the process in the critical section broadcasts the message `unlock!!` in order to restore the initial configuration. The key point here is that the description of the protocol is independent of the number of processes in the network.

2.2 Semantics

Let $B = (S, L, R)$ be a broadcast protocol, and let $S = \{s_1, \ldots, s_n\}$. A *configuration* is a vector $\mathbf{c} = \langle c_1, \ldots, c_n \rangle$ where c_i denotes the number of processes in state s_i for $i : 1, \ldots, n$.

Moves between configurations are either local (a process moves in isolation to a new state), rendezvous (two processes exchange a message and move to new states), or broadcasts (a process sends a message to all other processes; all processes move to new states). Formally, the possible moves are the smallest subset of $\mathbb{N}^n \times \Sigma \times \mathbb{N}^n$ satisfying the three conditions below, where u_i denotes the configuration such that $u_i(s_i) = 1$ and $u_i(s_j) = 0$ for $j \neq i$, and where $\mathbf{c} \xrightarrow{a} \mathbf{c}'$ denotes $(\mathbf{c}, a, \mathbf{c}') \in R$.

- If $s_i \xrightarrow{a} s_j$, then $\mathbf{c} \xrightarrow{a} \mathbf{c}'$ for every $\mathbf{c}$, $\mathbf{c}'$ such that $\mathbf{c}(s_i) \geq 1$ and $\mathbf{c}' = \mathbf{c} - \mathbf{u}_i + \mathbf{u}_j$. I.e. one process is removed from s_i, and one process is added to s_j.
- If $s_i \xrightarrow{a!} s_j$ and $s_k \xrightarrow{a?} s_l$, then $\mathbf{c} \xrightarrow{a} \mathbf{c}'$ for every $\mathbf{c}$, $\mathbf{c}'$ such that $\mathbf{c}(s_i) \geq 1$, $\mathbf{c}(s_k) \geq 1$ and $\mathbf{c}' = \mathbf{c} - \mathbf{u}_i - \mathbf{u}_k + \mathbf{u}_j + \mathbf{u}_l$.
  I.e. one process is removed from s_i and s_k, and one process is added to s_j and s_l.
- If $s_i \xrightarrow{a!!} s_j$, then $\mathbf{c} \xrightarrow{a} \mathbf{c}'$ for every $\mathbf{c}$, $\mathbf{c}'$ such that $\mathbf{c}(s_i) \geq 1$ and $\mathbf{c}'$ can be computed from $\mathbf{c}$ in the following three steps:

$$\mathbf{c}_1 = \mathbf{c} - \mathbf{u}_i \tag{1}$$

$$\mathbf{c}_2(s_k) = \sum_{\{s_l \mid s_l \xrightarrow{a??} s_k\}} \mathbf{c}_1(s_l) \tag{2}$$

$$\mathbf{c}' = \mathbf{c}_2 + \mathbf{u}_j \tag{3}$$

  I.e. the sending process leaves s_i (1), all other processes receive the broadcast and move to their destinations (2), and the sending process reaches s_j (3).

Thanks to the conditions (i) and (ii) of Section 2.1, the configuration $\mathbf{c}'$ is completely determined by $\mathbf{c}$ and the action a.

We denote by $\preceq$ the pointwise order between configurations, i.e. $\mathbf{c} \preceq \mathbf{c}'$ if and only if $\mathbf{c}(s_i) \leq \mathbf{c}'(s_i)$ for every $i : 1, \ldots, n$. A *parameterized configuration* is a *partial* function $\mathbf{p}\colon S \to \mathbb{N}$. Loosely speaking, $\mathbf{p}(s) = \perp$ denotes that the number of processes on state s is arbitrary. Formally, a parameterised configuration denotes a set of configurations, namely those extending $\mathbf{p}$ to a total function.

2.3 Checking Safety Properties

In this paper we study the *reachability* problem for broadcast protocols, defined as follows:

> Given a broadcast protocol B, a parameterized initial configuration $\mathbf{p}_0$ and a set of configurations C, can a configuration $\mathbf{c} \in C$ be reached from one of the configurations of $\mathbf{p}_0$?

In [EFM99] this problem is shown to be decidable for upwards-closed sets C.[1] A set C is *upwards-closed* if $\mathbf{c} \in C$ and $\mathbf{c}' \succeq \mathbf{c}$ implies $\mathbf{c}' \in C$. The mutual exclusion property of the example in the introduction can be checked by showing that no configuration satisfying $Use \geq 2$ (an upwards-closed set) is reachable from an initial configuration satisfying $Wait = 0, Use = 0$. It is shown in [EFM99] that the model-checking problem for safety properties can be reduced to the reachability problem for upwards-closed sets. (Here we follow the automata-theoretic approach to model-checking [VW86], in which a safety property is modelled as a regular set of dangerous sequences of actions the protocol should *not* engage in.)

The algorithm of [EFM99] for the reachability problem in the upwards-closed case is an "instantiation" of a general backwards reachability algorithm presented in [ACJ+96] (see also [FS98]). Define the *predecessor* operator as follows:

$$pre(C) = \{\mathbf{c} \mid \mathbf{c} \xrightarrow{a} \mathbf{c}',\ \mathbf{c}' \in C\}.$$

I.e., *pre* takes a set of configurations C_0, and delivers its set of *immediate* predecessors. The algorithm repeatedly applies the predecessor operator until a fixpoint is reached, corresponding to the set of all predecessors of C_0. If this set contains some initial configurations, then C_0 is reachable.

Proc Reach(C_0 : upwards-closed set of configurations)
  $C := C_0$;
  repeat
    $old\_C := C$;
    $C := old\_C \ \cup \ pre(old\_C)$;
  until $C = old\_C$;
  return C

The algorithm works because of the following properties: (i) if C is upwards-closed, then so is $pre(C)$; (ii) the set of minimal elements of an upwards-closed set with respect to the pointwise order is finite (see also Section 4); (iii) the repeat loop terminates. To prove property (i), we observe that we can associate to each label $a \in \Sigma$ [EFM99]:

- The set of configurations Occ_a from which a can occur.
  In the case of local moves and broadcasts there is a state s_i such that $Occ_a = \{\mathbf{c} \mid \mathbf{c}(s_i) \geq 1\}$. In the case of rendezvous there are states s_i, s_j such that $Occ_a = \{\mathbf{c} \mid \mathbf{c}(s_i) \geq 1 \text{ and } \mathbf{c}(s_j) \geq 1\}$.

[1] On the other hand, the problem is undecidable for singleton sets!.

– An affine transformation $\mathbf{T_a}(\mathbf{x}) = \mathbf{M_a} \cdot \mathbf{x} + \mathbf{b}_a$ such that if $\mathbf{c} \xrightarrow{a} \mathbf{c}'$, then $\mathbf{c}' = \mathbf{T_a}(\mathbf{c})$.
$\mathbf{M_a}$ is a matrix whose columns are unit vectors, and $\mathbf{b}$ is a vector of integers. (Actually, the components of $\mathbf{b}$ belong to $\{-1, 0, 1\}$, but our results can be extended without changes to the case in which they are arbitrary integer numbers. An example is discussed in Section 8.)

It follows that $pre(C)$ can be computed by the equation

$$pre(C) = \bigcup_{a \in \Sigma} (Occ_a \cap \mathbf{T_a^{-1}}(C)) \tag{4}$$

Hence if C is upwards-closed then so is $pre(C)$. Properties (ii) and (iii) are an immediate consequence of the well-known

Lemma 1 (Dickson's Lemma). *Let $\mathbf{v}_1, \mathbf{v}_2, \ldots$ be an infinite sequence of elements of $\mathbb{N}^k$. There exists $i < j$ such that $\mathbf{v}_i \preceq \mathbf{v}_j$ (pointwise order).*

The only known upper-bound for the number of iterations until termination is non-primitive recursive [McA84]. However, despite this result, the algorithm can still be applied to small but interesting examples.

3 Symbolic Representation via Constraints

A *linear arithmetic constraint* (or *constraint* for short) is a (finite) first-order formula $\phi_1 \wedge \ldots \wedge \phi_n$. with free variables (implicitly existentially quantified), and such that each ϕ_i is an atomic formula (constraint) built over the predicates $=, \geq, \leq, >, <$ and over arithmetic expressions (without multiplication between variables) built over $+, -, *, 0, 1$, etc.

The solutions (assignments of values to the free variables that make the formula true) of a constraint ϕ over the domain $\mathcal{D}$ are denoted by $[\![\phi]\!]_{\mathcal{D}}$. In the sequel we always take $\mathcal{D} = \mathbb{Z}$, and abbreviate $[\![\phi]\!]_{\mathbb{Z}}$ to $[\![\phi]\!]$. We often represent the *disjunction* of constraints $\phi_1 \vee \ldots \vee \phi_n$ as the *set* $\{\phi_1, \ldots, \phi_n\}$.

Constraints can be used to symbolically represent sets of configurations of a broadcast protocol. Given a protocol with states $\{s_1, \ldots, s_n\}$, let $\mathbf{x} = x_1, \ldots, x_n$ be a vector of variables, where x_i is intended to stand for the number of processes currently in state s_i. We assume that variables range over positive values (i.e., each variable x_i comes with an implicit constraint $x_i \geq 0$). A configuration $\mathbf{c} = \langle c_1, \ldots, c_n \rangle$ is simply represented as the constraint $\bigwedge_{i=1}^{n} x_i = c_i$. A parametric configuration $\mathbf{p} = \langle p_1, \ldots, p_n \rangle$ is represented as the constraint $\bigwedge_{i=1}^{n} \phi_i$ where: if $p_i \in \mathbb{N}$ then ϕ_i is the atomic constraint $x_i = c_i$, and if $p_i = \perp$ then ϕ_i is the atomic constraint $x_i \geq 0$.

As an example, the flow of processes caused by the *lock* broadcast in the protocol of the introduction is described by the inequality below (where, for clarity, we use *Think*, *Wait*, *Use* instead of x_1, x_2, x_3 and we omit the equalities of the form $x_i' = x_i$).

$$Think \geq 1 \wedge Think' = 0 \wedge Wait' = Think + Wait - 1 \wedge Use' = Use + 1$$

Let $\mathcal{C}$ be a class of constraints denoting exactly the upwards-closed sets, i.e., if a set S is upwards-closed then there is a set of constraints $\Phi \subseteq \mathcal{C}$ such that $[\![\Phi]\!] = S$, and viceversa. We can use any such class $\mathcal{C}$ to derive a symbolic version Symb-Reach$_\mathcal{C}$ of the procedure Reach:

Proc Symb-Reach$_\mathcal{C}$(Φ_0 : set of constraints of $\mathcal{C}$)
  $\Phi := \Phi_0$;
  repeat
    $old\_\Phi := \Phi$;
    $\Phi := old\_\Phi \ \cup \ \mathbf{pre}_\mathcal{C}(old\_\Phi)$;
  until Entail$_\mathcal{C}(\Phi, old_\Phi)$;
  return Φ

where (a) $\mathcal{C}$ is closed under application of $\mathbf{pre}_\mathcal{C}$, (b) $[\![\mathbf{pre}_\mathcal{C}(\Phi)]\!] = pre([\![\Phi]\!])$, and (c) Entail$_\mathcal{C}(\Phi, \Psi) = true$ if and only if $[\![\Phi]\!] \subseteq [\![\Psi]\!]$.

Condition (b) on $\mathbf{pre}_\mathcal{C}$ can be reformulated in syntactic terms. Let Φ be a set of constraints, and for each action a let G_a be a constraint such that $[\![G_a]\!] = Occ_a$ (we call G_a the *guard* of the action a). We have $\mathbf{T_a^{-1}}([\![\Phi]\!]) = [\![\Phi[\mathbf{x}/\mathbf{T_a}(\mathbf{x})]]\!]$. By equation (4) we obtain

$$\mathbf{pre}_\mathcal{C}(\Phi) \equiv \bigvee_{a \in \Sigma, \phi \in \Phi} G_a \wedge \phi[\mathbf{x}/\mathbf{T_a}(\mathbf{x})] \tag{5}$$

where $\equiv$ denotes logical equivalence of constraints.

In the next sections we investigate which classes of constraints are suitable for Symb-Reach$_\mathcal{C}$. We consider only classes $\mathcal{C}$ denoting exactly the upwards-closed sets. In this way, the termination of Symb-Reach$_\mathcal{C}$ follows directly from the termination of Reach, under the proviso that there exist procedures for computing $\mathbf{pre}_\mathcal{C}(\Phi)$ and for deciding Entail$_\mathcal{C}(\Phi, \Psi)$.

The suitability of a class $\mathcal{C}$ is measured with respect to the following parameters:

(1) The computational complexity of deciding Entail$_\mathcal{C}(\Phi, \Psi)$.
(2) The size of the set $\mathbf{pre}_\mathcal{C}(\Phi)$ as a function of the size of Φ.

A note about terminology. Given two sets of constraints Φ, Ψ, we refer to the *containment problem* as the decision problem Entail$(\Phi, \Psi) = true$ for two sets of constraints Φ, Ψ, whereas we refer to the *entailment problem* as the decision problem Entail$(\{\phi\}, \{\psi\}) = true$ for constraints ϕ and ψ.

4 NA-Constraints: No Addition

A NA-constraint is a conjunction of atomic constraints of the form $x_i \geq k$, where $x_i \in \{x_1, \ldots, x_n\}$ and k is a positive integer.

The class of NA-constraints denotes exactly the upwards closed sets. If Φ is a set of NA-constraints then $[\![\Phi]\!]$ is clearly upwards-closed. For the other direction, observe first that an upwards-closed set C is completely characterised by its set of

minimal elements M, where minimality is taken with respect the pointwise order $\prec$. More precisely, we have $C = \cup_{\mathbf{m}\in M} Up(\mathbf{m})$, where $Up(\mathbf{m}) = \{\mathbf{c} \mid \mathbf{c} \succeq \mathbf{m}\}$. The set M is finite by Dickson's lemma, and $Up(\mathbf{m})$ can be represented by the constraint $x_1 \geq \mathbf{m}(s_1) \wedge \ldots \wedge x_n \geq \mathbf{m}(s_n)$. So the set C can be represented by a set of NA-constraints.

4.1 Complexity of the Containment Problem in NA

The containment problem can be solved in polynomial time. In fact, the following properties hold. Let Φ, Ψ be sets of NA-constraints. Then,

- Φ entails Ψ if and only if for every constraint $\phi \in \Phi$ there is a constraint $\psi \in \Psi$ such that ϕ entails ψ.
- $\bigwedge_{i=1}^{n} x_i \geq k_i$ entails $\bigwedge_{i=1}^{n} x_i \geq l_i$ if and only if $k_i \geq l_i$ for $i : 1, \ldots, m$.

Thus, the worst-case complexity of the test 'Φ entails Ψ' is $O(|\Phi| * |\Psi| * n)$, where n is the number of variables in Φ and Ψ.

4.2 Size of the Set $\mathbf{pre}_{\mathrm{NA}}(\Phi)$

Let Φ be a set of NA-constraints. By equation (5), $\mathbf{pre}_{\mathrm{NA}}(\Phi)$ must be equivalent to the set $\bigvee_{a\in\Sigma,\phi\in\Phi} G_a \wedge \phi[\mathbf{x}/\mathbf{T_a}(\mathbf{x})]$. Unfortunately, we cannot choose $\mathbf{pre}_{\mathrm{NA}}(\Phi)$ *equal* to this set, because it may contain constraints of the form $x_{i_1} + \ldots + x_{i_m} \geq k$. However, when evaluating variables on positive integers, a constraint of the form $x_{i_1} + \ldots + x_{i_m} \geq k$ is equivalent to the following set (disjunction) of NA-constraints:

$$\bigvee_{\langle k_1,\ldots,k_m\rangle} x_{i_1} \geq k_1 \wedge \ldots \wedge x_{i_m} \geq k_m,$$

where each tuple of positive integers $\langle k_1, \ldots, k_m\rangle$ represents an ordered partition of k, i.e. $k_1 + \ldots + k_m = k$. (Moreover, it is easy to see that this is the smallest representation of $x_{i_1} + \ldots + x_{i_m} \geq k$ with NA-constraints.) We define the operator $\mathbf{pre}_{\mathrm{NA}}$ as the result of decomposing all constraints with additions of (5) into NA-constraints.

The cardinality of $\mathbf{pre}_{\mathrm{NA}}(\Phi)$ depends on the number of ordered partitions of the constants appearing in constraints with additions. For $x_1 + \ldots + x_m \geq k$, this number, denoted by $\rho(m, k)$, is equal to the number of subsets of $\{1, 2, \ldots, k + m - 1\}$ containing $m - 1$ elements, i.e.,

$$\rho(m,k) = \binom{k+m-1}{n-1} = \binom{k+m-1}{k}.$$

If c is the biggest constant occurring in constraints of Φ, and n, a are the number of states and actions of the broadcast protocol, we get $|\mathbf{pre}_{\mathrm{NA}}(\Phi)| \in O(|\Phi| * a * \rho(n, c))$. This makes NA-constraints inadequate for cases in which the constants $c \approx n$, initially or during the iteration of algorithm Symb-Reach$_{\mathrm{NA}}$. In this case we get $\rho(n, c) \approx \frac{4^n}{\sqrt{\pi n}}$, which leads to an exponential blow-up.

4.3 Conclusion

NA-constraints have an efficient entailment algorithm, but they are inadequate as data structure for Symb-Reach. Whenever the constants in the constraints reach values similar to the number of states, the number of constraints grows exponentially.

The blow-up is due to the decomposition of constraints with additions into NA-constraints. In the following section we investigate whether constraints with additions are a better data structure.

5 AD-Constraints: With Addition

An AD-constraint is a conjunction of atomic constraints $x_{i_1} + \ldots + x_{i_m} \geq k$ where $x_{i_1}, \ldots, x_{i_m}$ are *distinct* variables of $\{x_1, \ldots, x_n\}$, and k is a positive integer. A constraint in AD can be characterized as the system of inequalities $\mathbf{A} \cdot \mathbf{x} \geq \mathbf{b}$ where $\mathbf{A}$ is a 0-1 matrix.

It is easy to see that AD-constraints denote exactly the upwards-closed sets. Since AD-constraints are equivalent to disjunctions of NA-constraints, they only denote upwards-closed sets, and since they are more general than NA-constraints, they denote them all.

5.1 Complexity of the Containment Problem in AD

The following result shows that even the entailment test between two AD-constraints is difficult to decide.

Proposition 1 (Entailment in AD is co-NP complete). *Given two AD-constraints ϕ and ψ, the problem 'ϕ entails ψ' is co-NP complete.*

Proof. By reduction from HITTING SET [GJ78]. An instance of HITTING SET consists of a finite set $S = \{s_1, \ldots, s_n\}$, a finite family $S_1, \ldots, S_m$ of subsets of S, and a constant $k \leq n$. The problem is to find $T \subseteq S$ of cardinality at most k that *hits* all the S_i, i.e., such that $S_i \cap T \neq \emptyset$.

Take a collection of variables $X = \{x_1, \ldots, x_n\}$. Let ϕ be a conjunction of atomic constraints ϕ_i, one for each set S_i, given by: If $S_i = \{s_{i_1}, \ldots, s_{i_{n_i}}\}$, then $\phi_i = x_{i_1} + \ldots + x_{i_{n_i}} \geq 1$. Let $\psi = x_1 + \ldots + x_n \geq k + 1$.

If ϕ does not entail ψ, then there is a valuation $V: X \to I\!N$ that satisfies ϕ but not ψ. Let T be the set given by: $s_i \in T$ if and only $V(x_i) > 0$. Since V satisfies ϕ, T is a hitting set. Since V does not satisfy ψ, it contains at most k elements.

If T is a hitting set with at most k elements, then the valuation $V: X \to I\!N$ given by $V(x_i) = 1$ if $s_i \in T$, and 0 otherwise, satisfies ϕ but not ψ.

This implies that entailment of AD-constraints is co-NP-hard. Completeness follows by noting that the containment problem for sets of linear arithmetics constraints is co-NP complete [Sri92]. □

The following corollary immediately follows.

Corollary 1 (Containment in AD is co-NP complete). *Given two sets of AD-constraints Φ and Ψ, the problem 'Φ entails Ψ' is co-NP complete.*

5.2 Size of the Set $\mathbf{pre_{AD}}(\Phi)$

We can define

$$\mathbf{pre}_{\mathrm{AD}}(\Phi) = \bigvee_{a \in \Sigma, \phi \in \Phi} G_a \wedge \phi[\mathbf{x}/\mathbf{T_a}(\mathbf{x})]$$

since the right hand side is a set of AD-constraints whenever Φ is. If a is the number of actions of the broadcast protocol, then $|\mathbf{pre}_{\mathrm{AD}}(\Phi)| \in O(|\Phi| * a)$.

5.3 Conclusion

AD-constraints are not a good data structure for Symb-Reach either, due to the high computational cost of checking containment and entailment. This result suggests to look for a class of constraints between NA and AD.

6 DV-Constraints: With Distinct Variables

DV-constraints are AD-constraints of the form

$$x_{1,1} + \ldots + x_{1,n_1} \geq k_1 \quad \wedge \ldots \wedge \quad x_{m,1} + \ldots + x_{m,n_m} \geq k_m \,,$$

where $x_{i,j}$ and $x_{i',j'}$ are distinct variables (DV) for all i, j, i', j'. In other words, a DV-constraint can be represented as $\mathbf{A} \cdot \mathbf{x} \geq \mathbf{b}$ where $\mathbf{A}$ is a 0-1 matrix with unit vectors as columns.

Since DV-constraints are more general than NA-constraints, but a particular case of AD-constraints, they denote exactly the upwards-closed sets.

6.1 Complexity of the Containment Problem in DV

Entailment between sets of DV-constraints can still be very expensive, as shown by the following result.

Proposition 2 (Containment in DV is co-NP complete). *Given two sets of DV-constraints Φ and Ψ, the problem 'Φ entails Ψ' is co-NP complete.*

Proof. By reduction from INDEPENDENT SET [GJ78]. An instance of INDEPENDENT SET consists of a finite graph $G = (V, E)$ and a constant $k \leq |V|$. The problem is to find $I \subseteq V$ of cardinality at most k such that for every $u, v \in I$ there is no edge between u and v.

Assume $V = \{v_1, \ldots, v_n\}$. Take a collection of variables $X = \{x_1, \ldots, x_n\}$. The set Φ contains a constraint $x_i \leq 1$ for $i : 1 \ldots n$, and $x_i + x_j \leq 1$ for every edge $(v_i, v_j) \in E$. The set Ψ is the singleton $\{\psi\}$, where $\psi = x_1 + \ldots + x_n \geq k+1$.

If Φ does not entail ψ, then there is a valuation $V : X \to I\!N$ that satisfies Φ but not ψ. Let I be the set given by: $s_i \in I$ if and only $V(x_i) > 0$. Since V

satisfies Φ, I is an independent set. Since V does not satisfy ψ, it contains at most k elements.

If I is an independent set with at most k elements, then the valuation $V: X \to \mathbb{N}$ given by $V(x_i) = 1$ if $s_i \in I$, and 0 otherwise, satisfies ϕ but not ψ. □

However, and differently from the AD-case, checking entailment between two AD-constraints can be done in polynomial time. Let $Var(\phi)$ denote the set of free variables occurring in the constraint ϕ, and let $Cons(\gamma)$ denote the constant occurring in the *atomic* constraint γ. We have the following result:

Proposition 3. *Let ϕ and γ be an arbitrary and an atomic* DV*-constraint, respectively. Let Δ be the largest set of atomic constraints δ in ϕ such that $Var(\delta) \subseteq Var(\gamma)$. Then, ϕ entails γ if and only if $\Sigma_{\delta \in \Delta} Cons(\delta) \geq Cons(\gamma)$.*

Proof. ($\Rightarrow$): Assume $\Sigma_{\delta \in \Delta} Cons(\delta) < Cons(\gamma)$. Then, any valuation that assigns $Cons(\delta)$ to one variable in δ and 0 to the others, and 0 to the remaining variables of $Var(\gamma)$, satisfies ϕ but not γ.
($\Leftarrow$): Clearly ϕ entails Δ. Since ϕ is a DV-constraint, Δ entails the constraint $\sum_{x_i \in Var(\delta)} x_i \geq \Sigma_{\delta \in \Delta} Cons(\delta)$. Since $Var(\delta) \subseteq Var(\gamma)$ and $\sum_{\delta \in \Delta} Cons(\delta) \geq Cons(\gamma)$, it also entails $\sum_{x_i \in Var(\gamma)} x_i \geq Cons(\gamma)$, which is the constraint γ. □

For instance, we have that $x_1 + x_2 \geq a \wedge x_3 \geq b$ entails $x_1 + x_2 + x_3 + x_4 \geq c$ if and only if $a + b \geq c$.

Since ϕ entails ψ if and only if ϕ entails each atomic constraint of ψ, we get the following

Corollary 2 (Entailment in DV **is in P).** *Given two* DV*-constraints ϕ and ψ, it can be checked in polynomial time whether ϕ entails ψ.*

Since the symbolic procedure for the reachability problem requires to check containment, and not entailment, Corollary 2 does not seem to be of much use at first sight. However, it allows to define a new reachability procedure by replacing the $\text{Entail}_{\mathcal{C}}(\Phi, old\_\Phi)$ test in Symb-Reach by the *local* containment test:

$$\textbf{forall } \phi \in \Phi \textbf{ exists } \psi \in old\_\Phi$$

Clearly, the local containment test implies the containment test, and so the new procedure is partially correct. The risk of weakening the fixpoint test is that we may end up with a non-terminating algorithm. Fortunately, this turns out not to be the case, as shown by the following proposition.

Proposition 4. *The procedure Symb-Reach$_{\text{DV}}$ terminates.*

Proof. Let X be a set of variables. Given $Y \subseteq X$, let $Y \geq k$ denote the constraint $\sum_{x_i \in Y} x_i \geq k$.

Let ϕ be a DV-constraint on X. We define the function f_ϕ which assigns to $Y \subseteq X$ a natural number as follows:

$$f_\phi(Y) = \begin{cases} k & \text{if } \Phi \text{ contains the constraint } Y \geq k \\ 0 & \text{otherwise} \end{cases}$$

Observe that f_ϕ is well defined because ϕ is a DV-constraint. Define the pointwise ordering $\preceq$ on these functions, given by $f_\phi \preceq f_\psi$ if $f_\phi(Y) \leq f_\psi(Y)$ for every subset Y of X. We prove that the local containment test corresponds exactly to the pointwise ordering. I.e., for DV-constraints, ϕ entails ψ if and only if $f_\phi(Y) \geq f_\psi(Y)$.

- If $f_\phi \geq f_\psi$, then ϕ entails ψ.
  Let $Y \geq k$ be an atomic constraint of ψ. It follows from $f_\phi(Y) \geq f_\psi(Y)$ that ϕ contains a constraint $Y \geq k'$ such that $k' \geq k$. So every solution of ϕ is a solution of $Y \geq k$.
- If ϕ entails ψ, then $f_\phi \geq f_\psi$.
  We prove the contraposition. Let $Y \subseteq X$ such that $f_\phi(Y) < f_\psi(Y)$. Then ψ contains a constraint $Y \geq k$, and ϕ contains a constraint $Y \geq k'$ such that $k' < k$ (if ϕ contains no constraint $Y \geq k'$ we can assume that it contains the constraint $Y \geq 0$). Since ϕ is a DV-constraint, it has a solution X_0 such that $Y_0 = k'$. So X_0 does not satisfy $Y \geq k$, and so ϕ does not entail ψ.

Assume now that Symb-Reach$_{\mathrm{DV}}$ does not terminate. Then, the i-th iteration of the repeat loop generates at least one constraint ϕ_i such that ϕ_i does not entail ϕ_j for any $i > j$. By the result above, the sequence of functions f_{ϕ_i} satisfies $f_{\phi_i} \not\preceq f_{\phi_j}$ for any $i > j$. This contradicts Dickson's lemma (consider a function f_ϕ as a vector of $\mathbb{N}^{2^{|X|}}$). □

6.2 Size of the Set $\mathbf{pre}_{\mathrm{DV}}(\Phi)$

If Φ is a set of DV-constraints, then the set of constraints (5) may contain AD-constraints with shared variables. However, each constraint in set (5) is either a DV-constraint or has one of the two following forms: $\phi \wedge x_i \geq 1$ or $\phi \wedge x_i \geq 1 \wedge x_j \geq 1$, where ϕ is a DV-constraint with at most one occurrence of x_i and x_j. The constraints of the form $x_i \geq 1$ correspond to the 'guards' of the transition rules of the protocol. Thus, in order to maintain constraints in DV-form, all we have to do is to merge the 'guards' and the remaining DV-constraint (i.e. ϕ). The operator $\mathbf{pre}_{\mathrm{DV}}$ is defined as the result of applying the following normalization: Given a constraint $x \geq 1 \ \wedge \ x + y_1 + \ldots + y_m \geq k \wedge \phi$ where, by hypothesis, x does not occur in ϕ, replace it by the equivalent set of constraints

$$\bigvee_{i=0}^{k-1} (x \geq k - i \ \wedge \ y_1 + \ldots + y_m \geq i \wedge \phi) \, .$$

In the worst case, it is necessary to reduce each new constraint with respect to two guards, possibly generating $O(k^2)$ new constraints. Thus, if a is the number of actions of the protocol and c is the maximum constant occurring in the set Φ of DV-constraints, we have $|\mathbf{pre}_{\mathrm{DV}}(\Phi)| \in O(|\Phi| * a * c^2)$.

6.3 Conclusion

DV-constraints are a good compromise between AD and NA-constraints. The application of $\mathbf{pre}_{\mathrm{DV}}$ does not cause an exponential blow up as in the case of NA-constraints. Furthermore, though the containment test is co-NP complete, it can be relaxed to an entailment of low polynomial complexity, unlike the case of AD-constraints. Moreover, as shown in the next section, sets of DV-constraints can be compactly represented.

7 Efficient Representation of Sets of Constraints

DV-constraints can be manipulated using very efficient data-structures and operations. We consider constraints over the variables $\{x_1, \ldots, x_n\}$.

Each *atomic* DV-constraint $\Sigma_{x_i \in Y} x_i \geq k$ can be represented as a pair $\langle \mathbf{b}, k \rangle$, where $\mathbf{b}$ is a *bit-vector*, i.e., $\mathbf{b} = \langle b_1, \ldots, b_n \rangle$ and $b_i = 1$ if $x_i \in Y$, and 0 otherwise. Thus, a DV-constraint can be represented as a set of pairs. Based on this encoding, the decision procedure of Corollary 2 can be defined using bitvector operations *not* and *or*. (**1** denotes the bitvector containing only 1's.)

Proc Entails(*cstr1*, *cstr2*: codings of DV-constraints)
var s : *integer*
for all pairs $\langle \mathbf{b}_2, k_2 \rangle$ in *cstr2*
$s := 0$;
for all pairs $\langle \mathbf{b}_1, k_1 \rangle$ in *cstr1*
if $(not(\mathbf{b}_1)\ or\ \mathbf{b}_2) = \mathbf{1}$ **then** $s := s + k_1$ **endif**
endfor
if $s < k_2$ **then return false endif**
endfor;
return true

8 Examples

In this section we present and discuss some experimental results. We first show some examples of systems and properties that we were able to verify automatically, and then we compare the execution times obtained by using different constraint systems.

The protocol shown in Fig. 1 models a network of processes accessing two shared files (called 'a' and 'b') under the last-in first-served policy. When a process wants to write on one of the files all processes reading it are redirect in the initial state **I**. In the state **I** a process must send a broadcast before starting reading a file: in this case all writers are sent back to the state **I** (last-in first-served). Note that processes operating on 'b' simply skip the broadcast concerning operations on 'a' and vice versa. The protocol must ensure mutual

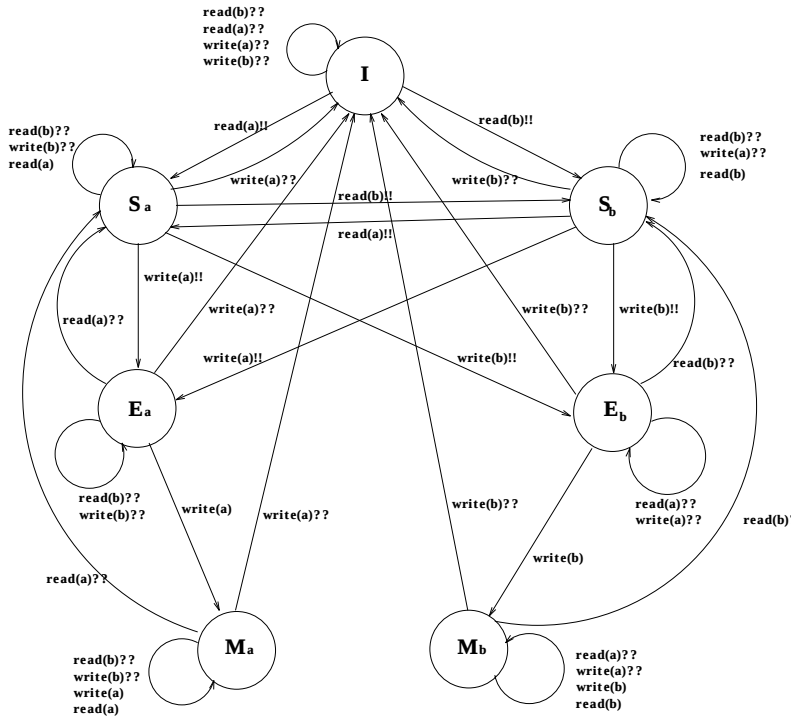


Fig. 1. Last-in first-served access to two resources.

exclusion between readers and writers. The initial parameterized configuration of the protocol is

$$I \geq 1, S_a = 0, S_b = 0, E_a = 0, E_b = 0, M_a = 0, M_b = 0\,.$$

We prove that the unsafe configurations $Sa \geq 1, Ma \geq 1$ are not reachable.

In Fig. 2, we describe a central server model [ABC+95]. Processes in state **think** represent thinking clients that submit jobs to the CPU. A number of processes may accumulate in state $\mathbf{wait_{cpu}}$. The first job requesting the CPU finds it idle and starts using it. A job that completes its service proceeds to a selection point where it continues requesting the I/O subsytem or leaves the central system. No specific policy is specified for the queues of waiting jobs. In the initial state of the broadcast protocol in Fig. 2 an arbitrary number of processes are in state **think**, whereas one process is respectively in state $\mathbf{idle_{cpu}}$, $\mathbf{idle_{disk}}$, $\mathbf{no_{int}}$. The protocol must ensure that only one job at a time can use the CPU and the I/O subsytem. The flow of processes is represented by a collection of rules over 17 variables (one for each state). The initial parameterized configuration of the protocol is

$$\mathit{Think} \geq 1,\ \mathit{Idle}_{cpu} = 1,\ \mathit{Idle}_{disk} = 1,\ \mathit{No\text{-}int} = 1,$$

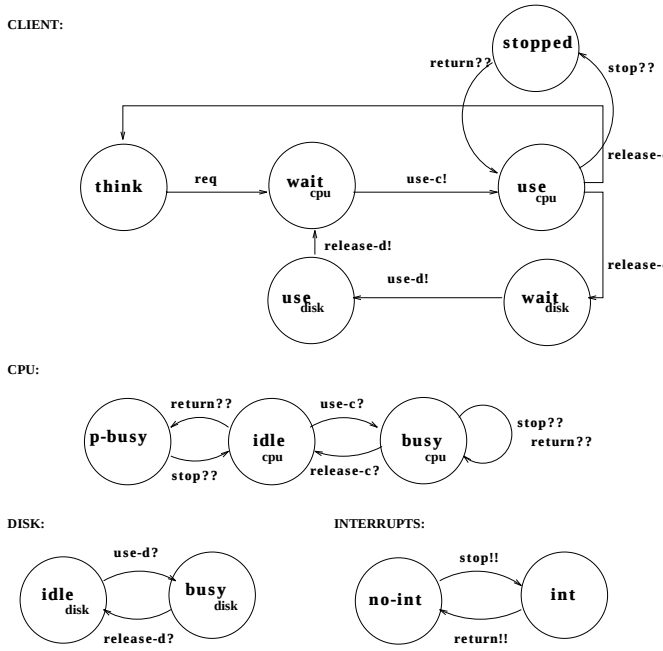


Fig. 2. Central Server System.

with all other variables equal to zero. We prove that the unsafe configurations $Use_{cpu} \geq 2$ is not reachable.

Petri Nets can be seen as a special case of broadcast protocols where the constraints generated during the analysis are in NA-form. Consider the Petri net of [Ter94] shown in Fig. 3, which describes a system for manufacturing tables (for instance, transition t_4 assembles a table by taking a board from the place p_6 and four legs from the place p_5). The constraint-based representation introduces a variable for each place and for each transition. The variables corresponding to transitions count the number of times a transition is fired during the execution. There is a rule for each transition. For instance, the rule corresponding to transition t_4 is

$$P_6 \geq 1,\ P_5 \geq 4,\ P_6' = P_6 - 1,\ P_5' = P_5 - 4,\ P_7' = P_7 + 1,\ T_4' = T_4 + 1$$

In [Ter94] it is shown that an initial marking of this is deadlock-free (i.e., no sequence of transition occurrences can lead to a deadlock) if and only if it enables a sequence of transition occurrences containing t_1 at least three times and all other transitions at least twice. Based on this preliminary result we can then compute all deadlock-free initial states. They are exactly the predecessors states of the states

$$T_1 \geq 3,\ T_2 \geq 2,\ T_3 \geq 2,\ T_4 \geq 2,\ T_5 \geq 2,\ T_6 \geq 2$$

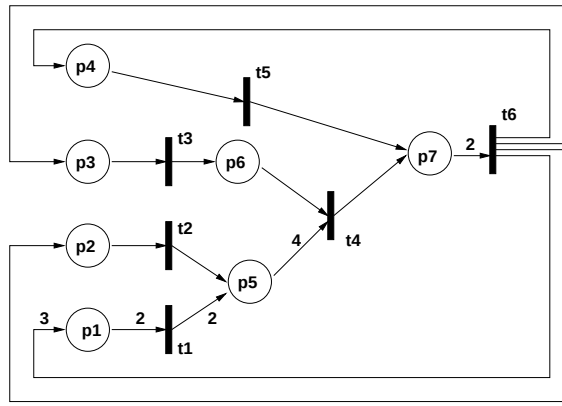


Fig. 3. Manufacturing System modeled as a Choice-free Petri Net.

intersected with the initial states of the system, i.e., those such that $T_i = 0$ for all i and $P_5 = P_6 = P_7 = 0$. The result of the fixpoint computation is given by the following set of constraints

$P_1 \geq 10, P_2 \geq 1, P_3 \geq 2$ $\quad$ $P_1 \geq 8, P_2 \geq 3$ $\quad$ $P_1 \geq 12, P_3 \geq 2$
$P_1 \geq 6, P_2 \geq 5, P_3 \geq 2$ $\quad$ $P_1 \geq 8, P_3 \geq 1, P_4 \geq 1$ $\quad$ $P_1 \geq 6, P_4 \geq 2$
$P_1 \geq 6, P_2 \geq 1, P_3 \geq 1, P_4 \geq 1$

8.1 Comparison of Execution Times

We have tested the previous examples on HyTech (polyhedra representation of sets of configurations, full entailment test), on Bultan, Gerber and Pugh's model checker based on the Omega library for Presburger arithmetic [BGP97], and on the specialized model checker we have introduced in the paper (DV-constraint representation of sets of states, local entailment test). HyTech works on real arithmetic, i.e., it employs efficient constraint solving for dealing with linear constraints. The results are shown in the following table, where 'Presb' refers to the model checker of [BGP97], and 'BitVector' to our checker.

Fig	Rules	Unsafe States	Steps	BitVector[1]	HyTech[1]	Presb[2]
1	21	$S_a \geq 1, M_a \geq 1$	2	<1s	<1s	not tested
2	9	$Use_{cpu} \geq 2$	7	<1s	5.5s	40s
		$Use_{cpu} \geq 3$	10	<1s	16s	290s
		$Use_{cpu} \geq 4$	13	<1s	40s	1558s
		$Use_{cpu} \geq 8$	25	15s	578s	not tested
		$Use_{cpu} \geq 10$	31	76s	1738s	not tested
3	6	$T_1 \geq 3, \wedge_{i>1} T_i \geq 2$	24	1090s	>6h	19h50m

[1] On a Sun Sparc 5.6. [2] On a Sun Ultra Sparc.

9 Related Work

The first algorithm for testing safety properties of broadcast protocols was proposed by Emerson and Namjoshi in [EN98]. Their approach is based on an extension of the Karp and Miller's cover graph construction (used for Petri Nets) [KM69]. In [EFM99], Esparza, Finkel and Mayr show that the algorithm may not terminate and propose a backwards-reachability procedure. The correctness of the procedure follows from general results on the decidability of infinite state systems by Abdulla *et al.* [ACJ$^+$96]. In [Kin99], Kindahl uses constraints as symbolic representation of upwards-closed sets for Petri Nets and lossy channel systems, but does not discuss the issue of finding adequate classes of constraints. Finally, Delzanno and Podelski [DP99], and Bérard and Fribourg [BF99] have recently applied real-arithmetics to model checking of integer systems.

10 Conclusion

We have proposed linear constraints with disjoint variables as a good symbolic representation for upwards-closed sets of configurations of broadcast protocols. Experimental results shown that even a prototype implementation can beat tools for more general constraints.

Acknowledgements We thank Tevfik Bultan for the experiments using his model checker based on Presburger Arithmetics [BGP97].

References

[ACJ$^+$96] P. A. Abdulla, K. Cerāns, B. Jonsson and Y.-K. Tsay. General Decidability Theorems for Infinite-State Systems. In *Proc. 10th IEEE Int. Symp. on Logic in Computer Science*, pp. 313–321, 1996.

[ABC$^+$95] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* Series in Parallel Computing. John Wiley & Sons, 1995.

[BF99] B. Bérard, and L. Fribourg. Reachability analysis of (timed) Petri nets using real arithmetic. In *Proc. .10th Int. Conf. on Concurrency Theory (CONCUR'99)*, Eindhoven, The Netherlands, August, 1999. To appear.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic Model Checking of Infinite-state Systems using Presburger Arithmetics. In Orna Grumberg, editor, *Proc. 9th Conf. on Computer Aided Verification (CAV'97)*, LNCS 1254, pp. 400–411. Springer-Verlag, 1997.

[DP99] G. Delzanno and A. Podelski, Model Checking in CLP. In W. R. Cleaveland, editor, *Proc. 5th Int. Con. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99).*, LNCS 1579, pp. 223–239. Springer-Verlag, 1999.

[EN98] E. A. Emerson and K. S. Namjoshi. On Model Checking for Non-Deterministic Infinite-State Systems. In *Proc. 13th IEEE Int. Symp. on Logic in Computer Science*, 1998.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In *Proc. 14th IEEE Int. Symp. on Logic in Computer Science*, 1998.

[FS98] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! Technical Report LSV-98-4, Laboratoire Spécification et Vérification, Ecole Normale Supérieure de Cachan. April 1998. To appear in *Theoretical Computer Science*, 1999.

[GJ78] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, 1978.

[HHW97] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a Model Checker for Hybrid Systems. In Orna Grumberg, editor, *Proc. 9th Conf. on Computer Aided Verification (CAV'97)*, LNCS 1254, pp. 460–463. Springer-Verlag, 1997.

[Kin99] M. Kindahl. *Verification of Infinite State Systems: Decision Problems and Efficient Algorithms*. Ph.D. thesis, Department of Computer Systems, Uppsala University, June 1999. Available as report DoCS 110.

[KM69] R. M. Karp and R. E. Miller. Parallel Program Schemata. *Journal of Computer and System Sciences*, 3, pp. 147-195, 1969.

[McA84] K. McAloon. Petri Nets and Large Finite Sets. *Theoretical Computer Science* 32, pp. 173–183, 1984.

[Sri92] D. Srivistava. Subsumption and indexing in constraint query languages with linear arithmetic constraints. In *2nd Int. Symp. on Artificial Intelligence and Mathematics, Fort Lauderdale*, 1992.

[Ter94] E. Teruel. *Structure Theory of Weighted Place/Transition Net Systems: The Equal Conflict Hiatus*. Ph.D. Thesis, University of Zaragoza, 1994.

[VW86] M. Y. Vardi and P. Wolper. Automata-Theoretic Techniques for Modal Logics of Programs. *Journal of Computer and System Sciences*. 32(2), pp. 183–221, April, 1986.

Descriptive Complexity Theory for Constraint Databases

Erich Grädel and Stephan Kreutzer

Lehrgebiet Mathematische Grundlagen der Informatik,
RWTH Aachen, D-52056 Aachen,
{graedel, kreutzer}@informatik.rwth-aachen.de

Abstract. We consider the data complexity of various logics on two important classes of constraint databases: dense order and linear constraint databases. For dense order databases, we present a general result allowing us to lift results on logics capturing complexity classes from the class of finite ordered databases to dense order constraint databases. Considering linear constraints, we show that there is a significant gap between the data complexity of first-order queries on linear constraint databases over the real and the natural numbers. This is done by proving that for arbitrary high levels of the Presburger arithmetic there are complete first-order queries on databases over $(\mathbb{N}, <, +)$. The proof of the theorem demonstrates a simple argument for translating complexity results for prefix classes in logical theories to results on the complexity of query evaluation in constraint databases.

1 Introduction

Descriptive complexity theory studies the relationship between logical definability and computational complexity. In particular one looks for results saying that, on a certain class $\mathcal{K}$ of structures, a logic L (like first-order logic or least fixed point logic) *captures* a complexity class $\mathcal{C}$. This means that (1) for every fixed sentence $\psi \in L$, the complexity of evaluating ψ on structures from $\mathcal{K}$ is a problem in the complexity class $\mathcal{C}$, and (2) every property of structures in $\mathcal{K}$ that can be decided with complexity $\mathcal{C}$ is definable in the logic L. Two important examples of such results are Fagin's Theorem, saying that existential second-order logic captures NP on the class of all finite structures, and the Immerman-Vardi Theorem, saying that least fixed point logic captures PTIME on the class of all ordered finite structures. Indeed, on *ordered* finite structures, logical characterizations of this kind are known for all major complexity classes. On the other hand it is not known, and one of the major open problems in the area, whether PTIME can be captured by any logic, if no ordering is present. We refer to [1, 10] for background on descriptive complexity.

Up to now, descriptive complexity has been considered almost exclusively on finite structures. But the research program of descriptive complexity makes sense also for classes of infinite structures, provided that they admit a finite presentation. There have been a few studies of descriptive complexity theory on

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 67–81, 1999.

infinite structures concerning, for instance, metafinite structures and complexity theory over the reals [3, 4], recursive structures [8] and, as we do in the present paper, constraint databases (see e.g. [12, 6, 5] and the references there).

Constraint databases are a modern database model admitting infinite relations that are finitely presented by quantifier-free formulae (constraints) over some fixed background structure. For example, to store geometrical data, it is useful to have not just a finite set as the domain of the database, but to include all real numbers 'in the background'. Also the presence of interpreted functions, like addition and multiplication, is desirable. The constraint database framework introduced by Kanellakis, Kuper and Revesz [12] meets both requirements. Formally, a constraint database consists of a *context structure* $\mathfrak{A}$, like $(\mathbb{R}, <, +, \cdot)$, and a set $\{\varphi_1, \ldots, \varphi_m\}$ of quantifier-free formulae defining the database relations. We give the precise definition in the next section.

When studying the data complexity of constraint query languages, it soon became clear that allowing recursion in query languages leads to non-closed or undecidable query languages even for rather simple context structures. On the other hand there are promising results for non-recursive languages in many interesting contexts. For the context structure $(\mathbb{R}, <)$ a LOGSPACE data complexity for first-order logic has been established by Kanellakis, Kuper, and Revesz which was later improved to AC^0 by Kanellakis and Goldin [11]. In [12] it has also been shown that first-order logic still has data complexity NC if the context structure is extended by addition and multiplication. Thus first-order logic is well-suited as a query language for spatial databases where the context structure is the field of reals.

In this paper we will consider the complexity of query evaluation in two important cases: (1) linear constraint databases, where the context structure is $(\mathbb{R}, <, +)$ or $(\mathbb{N}, <, +)$, and (2) constraint databases over dense linear orders.

It turns out that the data complexity of first-order query on linear constraint databases depends heavily on the universe. The data complexity of first-order queries on databases over $(\mathbb{R}, <, +)$ is known to be in NC. It has been conjectured by Grumbach and Su [6] that this is also true for the context structure $(\mathbb{N}, <, +)$. We refute this conjecture here by showing that we find complete first-order queries for each level of the polynomial time hierarchy.

As stated above, allowing recursion in query languages tends to result in undecidable languages. For instance, we will observe that this is the case for linear constraint queries over $(\mathbb{R}, <, +)$. An exception are *dense order constraint databases*, where the context structure is $(\mathbb{R}, <)$ (or any other dense linear order without endpoints). There we can incorporate recursion and still end up in decidable and closed languages. For instance, it has been shown in [12, 5] that inflationary DATALOG with negation has PTIME data complexity and, in fact, that it captures PTIME on dense order constraint databases. We continue this line of research and present a general technique that allows to lift capturing results from the class of ordered finite structures to constraint databases over $(\mathbb{R}, <)$. This is done by associating with every constraint database over $(\mathbb{R}, <)$ a finite ordered structure, called the *invariant* of the database which carries all the

information stored in the infinite database. The finite database can be defined by first-order formulae and therefore with very low data complexity. A query on the constraint database can be evaluated in the invariant in such a way, that the result of the original query can be regained from the answer on the finite database with very low data complexity. Indeed the invariant is first-order interpretable in the original database, and this allows to translate any formula that represents a query on the invariant into an equivalent formula over the original database. In this way capturing results are lifted from ordered finite structures to dense order constraint databases.

2 Constraint Databases

The basic idea in the definition of constraint databases is to allow infinite relations which have a finite representation by a quantifier-free formula. Let $\mathfrak{A}$ be a τ-structure, called the *context structure*, and $\varphi(x_1, \ldots, x_n)$ be a quantifier-free formula of vocabulary τ that may contain elements from A as parameters. Let $\sigma := \{R_1, \ldots, R_k\}$ be a relational signature disjoint from τ.

We say that an n-ary relation $R \subseteq A^n$ is *represented by* $\varphi(x_1, \ldots, x_n)$ *over* $\mathfrak{A}$, if $R = \{(a_1, \ldots, a_n) : \mathfrak{A} \models \varphi(a_1, \ldots, a_n)\}$. A *$\sigma$-constraint database* over the context structure $\mathfrak{A}$ is an expansion $\mathfrak{B} = (\mathfrak{A}, R_1, \ldots, R_k)$ of $\mathfrak{A}$ where all σ-relations R_i are finitely represented by formulae φ_{R_i} over $\mathfrak{A}$. The set $\Phi := \{\varphi_{R_1}, \ldots, \varphi_{R_k}\}$ is called a *finite representation of* $\mathfrak{B}$. The set of finitely representable relations over $\mathfrak{A}$ is denoted by $\mathrm{Rel}_{fr}(\mathfrak{A})$ and the set of all constraint databases over $\mathfrak{A}$ is denoted by $\mathrm{Exp}_{fr}(\mathfrak{A})$. The signature τ is called the *context signature* whereas σ is called the *database signature*.

By definition, constraint databases are expansions of a context structure by finitely representable database relations. Note that the same relation can be represented in different ways, e.g. $\varphi_1 := x < 10 \wedge x > 0$ and $\varphi_2 := (0 < x \wedge x < 6) \vee (6 < x \wedge x < 10) \vee x = 6$ are different formulae but define the same relation. Two representations Φ and Φ' are $\mathfrak{A}$-*equivalent*, if they represent the same database over $\mathfrak{A}$.

To measure the complexity of algorithms taking constraint databases as inputs we have to define the size of a constraint database. Unlike finite databases, the size of constraint databases cannot be given in terms of the number of elements stored in them but has to be based on a representation of the database. Note that equivalent representations of a database need not to be of the same size. Thus the size of a constraint database cannot be defined independent of a particular representation. In the following, whenever we speak of a constraint database $\mathfrak{B}$, we have a particular representation Φ of $\mathfrak{B}$ in mind. The size $|\mathfrak{B}|$ of $\mathfrak{B}$ then is defined as the sum of the length of the formulae in Φ. This corresponds to the standard encoding of constraint databases by the formulae of their representation.

Constraint queries. Let $\mathfrak{A}$ be a τ-structure and σ a relational signature. A *constraint query* $Q : Exp_{fr}(\mathfrak{A}) \to Rel_{fr}(\mathfrak{A})$ is a mapping from σ-constraint databases over $\mathfrak{A}$ to finitely representable relations over $\mathfrak{A}$. In the sequel we are

interested only in queries defined by formulae of a given logic $\mathfrak{L}$. In order to define queries by $\mathfrak{L}$-formulae, we require the context structures to admit quantifier elimination for $\mathfrak{L}$. This means that every $\mathfrak{L}$-formula φ is equivalent in $\mathfrak{A}$ to a quantifier-free formula. If $\mathfrak{A}$ admits quantifier elimination for $\mathfrak{L}$, then every formula $\psi(x_1, \ldots, x_k) \in \mathfrak{L}[\tau \cup \sigma]$ defines a query Q_φ taking a σ-constraint database $\mathfrak{B}$ over $\mathfrak{A}$ to the set $\{\bar{a} \in A^k \ : \ \mathfrak{B} \models \psi(\bar{a})\}$, and the result of the query is itself finitely representable.

Typical questions that arise when dealing with constraint query languages are the complexity of query evaluation for a certain constraint query language and the definability of a query in a given language. For a fixed query formula $\varphi \in \mathcal{L}$, the *data complexity* of the query Q_φ is defined as the amount of resources (e.g. time, space, or number of processors) needed to evaluate the function that takes a representation Φ of a database $\mathfrak{B}$ to a representation of the answer relation $Q_\varphi(\mathfrak{B})$.

3 Linear Constraints

In this section we consider linear constraint databases, that is, databases defined over the context structures $(\mathbb{R}, <, +)$, $(\mathbb{Q}, <, +)$ or $(\mathbb{N}, <, +)$. The data complexity of linear constraint queries in the context of $(\mathbb{R}, <, +)$ and $(\mathbb{Q}, <, +)$ has been studied by Grumbach, Su, and Tollu in [5, 7]. In [5] it is claimed that "first-order queries on linear constraint databases have a NC_1 data complexity."

First, we briefly discuss the possibility whether more powerful query languages than first-order logic can be effectively evaluated on linear constraint databases. However, a simple argument shows that adding a recursion mechanism to first-order logic leads to non-closed or undecidable languages. For example, the (FO+DTC)-formula $nat(x) := [\mathrm{DTC}_{x,y}(x + 1 = y)](0, x)$ defines the natural numbers, and multiplication of natural numbers can be defined by the (FO+DTC)-formula $mult(x, y, z) := [\mathrm{DTC}_{uv,u'v'}(u{+}1 = u' \wedge v{+}x = v')](00, yz)$.

It follows that Hilbert's 10th problem (or the existential theory of arithmetic) can be reduced to the evaluation of existential FO+DTC-queries on linear constraint databases.

Theorem 1. *Every query language over the context structure* $(\mathbb{R}, <, +)$ *which is at least as expressive as existential* FO+DTC *is undecidable.*

Thus the result by Grumbach and Su cannot be extended to query languages allowing recursion. We now show that the result does also not generalize to linear constraint queries over the natural numbers.

Presburger arithmetic (PrA), the theory of the structure $(\mathbb{N}, <, +)$, is well known to be decidable. Strictly speaking, we have to expand $(\mathbb{N}, <, +)$ by divisibility relations $a \mid x$ (for all parameters $a \in \mathbb{N}$), because otherwise the theory would not admit quantifier elimination and hence non-Boolean queries could not be evaluated in closed form. Note that $a \mid x$ is of course definable in $(\mathbb{N}, <, +)$ but not by a quantifier-free formula. However, we will show that even the evaluation

of boolean first-order queries is much more complex in the context of the Presburger arithmetic than on $(\mathbb{R}, <, +)$. This result relies on complexity results for fragments of PrA with bounded quantifier prefixes. Let $\mathcal{Q} := Q_1 \cdots Q_k$ be a word in $\{\exists, \forall\}^*$. Then $[\mathcal{Q}] \cap \mathrm{PrA}$ is the set of sentences of the form $\psi := Q_1 x_1 \cdots Q_k x_k \varphi$ such that $(\mathbb{N}, <, +) \models \psi$ and φ is quantifier-free. It has been shown [2, 15] that the complexity of such fragments of Presburger arithmetic may reside on arbitrary high levels of the polynomial-time hierarchy. Essentially the evaluation of formulae with $m+1$ quantifier blocks of bounded length is in the m-th level of the hierarchy.

Theorem 2 (Grädel, Schöning). *Let $m \geq 1, r_1, \ldots, r_m \geq 1$ and $r_{m+1} \geq 3$. Then, for odd m, $[\exists^{r_1} \forall^{r_2} \cdots \exists^{r_m} \forall^{r_{m+1}}] \cap \mathrm{PrA}$ is Σ_m^p-complete, and $[\forall^{r_1} \exists^{r_2} \cdots \forall^{r_m} \exists^{r_{m+1}}] \cap \mathrm{PrA}$ is Π_m^p-complete. For even m, $[\exists^{r_1} \forall^{r_2} \cdots \forall^{r_m} \exists^{r_{m+1}}] \cap \mathrm{PrA}$ is Σ_m^p-complete and $[\forall^{r_1} \exists^{r_2} \cdots \exists^{r_m} \forall^{r_{m+1}}] \cap \mathrm{PrA}$ is Π_m^p-complete.*

The proof of the following theorem exhibits a simple argument for translating such complexity results for prefix classes in logical theories to results on the complexity of query evaluation in constraint databases.

Theorem 3. *Let ψ be a first-order boolean query on constraint databases over $(\mathbb{N}, <, +)$. Then the data complexity of ψ is in the polynomial-time hierarchy. Conversely, for each class Σ_k^p, resp. Π_k^p of the polynomial time hierarchy there is a fixed query ψ whose data complexity is Σ_k^p-complete, resp. Π_k^p-complete.*

Proof. We can assume that $\psi = Q_1 x_1 \cdots Q_k x_k \varphi$ with φ quantifier-free and with database relations $R_1, \ldots, R_m$. Given a database $\mathfrak{B} = (\mathbb{N}, <, +, R_1, \ldots, R_m)$ where the database relations are represented by $\beta_1, \ldots, \beta_m$ over $(\mathbb{N}, <, +)$, let $\psi' := \mathit{unfold}(\psi, \mathfrak{B})$ be the unfolded query, obtained by replacing in ψ all occurrences of $R_1, \ldots, R_m$ by the defining formulae $\beta_1, \ldots, \beta_m$. Since the β_i are quantifier-free ψ' has the same prefix as ψ and length bounded by $O(|\mathfrak{B}|)$ (given that ψ is considered fixed). Obviously $\mathfrak{B} \models \psi$ if and only if $\psi' \in [Q_1 \ldots Q_k] \cap \mathrm{PrA}$. Hence the data complexity of ψ is in the polynomial-time hierarchy (and actually, we can read off the level of the hierarchy directly from the prefix of ψ).

For the second assertion of the theorem, consider any quantifier prefix $\mathcal{Q} = Q_1 \ldots Q_m$. Let R be an m-ary relation symbol and let $\psi_\mathcal{Q}$ be the query $Q_1 x_1 \cdots Q_m x_m R x_1 \ldots x_m$. The decision problem for $[\mathcal{Q}] \cap \mathrm{PrA}$ reduces to the evaluation problem of $\psi_\mathcal{Q}$ on constraint databases over $(\mathbb{N}, +, <)$. Indeed, for every sentence $\varphi = Q_1 x_1 \cdots Q_m x_m \varphi'(x_1, \ldots, x_m)$ in $\mathrm{FO}(<, +)$, let $\mathfrak{B}_\varphi$ be the $\{R\}$-database over $(\mathbb{N}, <, +)$ such that $R^{\mathfrak{B}_\varphi}$ is represented by φ'. The size of $\mathfrak{B}_\varphi$ is bounded by the length of φ. Clearly, φ is true in $(\mathbb{N}, <, +)$ if and only if $\mathfrak{B}_\varphi \models \psi_\mathcal{Q}$.

Hence, by choosing $\mathcal{Q}$ as indicated by Theorem 2, the evaluation problem for $\psi_\mathcal{Q}$ is Σ_k^p-complete, resp. Π_k-complete. □

We have seen that first-order logic can express quite complex queries. We now consider sub-classes of first-order logic which can still be efficiently evaluated. It has been shown by Lenstra and Scarpellini (see references in [2]) that for all fixed dimensions $t \in \mathbb{N}$, $[\exists^t] \cap \mathrm{PrA}$ and $[\forall^t] \cap \mathrm{PrA}$ are in PTIME. Thus, by an argument similar to the one above, we can show the following theorem.

Theorem 4. *Existential and universal boolean queries on constraint databases over* $(\mathbb{N}, <, +)$ *have* PTIME *data complexity.*

However, as soon as we admit queries with alternation depth two, the evaluation problem is NP- or Co-NP-hard. This follows from a result by Schöning [15] who proved that $[\exists\forall] \cap PrA$ is NP-complete, strengthening a result in [2].

4 Dense Linear Orders

We now consider the complexity of query evaluation in the context of dense linear orders. We prove a general result which allows us to give precise complexity bounds for the data complexity of various logics such as transitive closure or fixed-point logic and to extend results on logics capturing complexity classes from the realm of finite ordered structures to constraint databases over dense linear orders. Given a fixed query, its evaluation in a database can be done by (1) transforming the database into a finite structure, called its invariant, (2) evaluating a slightly modified version of the query on the invariant, and (3) transforming the result of the evaluation to an answer of the original query.

We fix the context structure $\mathfrak{A} := (\mathbb{R}, <)$ and a query ψ of vocabulary $\{<\} \cup \sigma$ with database signature $\sigma = \{R_1, \ldots, R_k\}$. Let $P^\psi \subseteq \mathbb{R}$ be the (finite) set parameters that occur in ψ. The query has to be transformed so that it can be evaluated in the invariant. This transformation is independent of a particular database and can be seen as a compilation or preprocessing step. To set up the evaluation method outlined above, we define two mappings. The first, *inv*, maps databases to their corresponding invariants; the second, π, maps the answer of the query on the invariant to the answer of the original query.

4.1 The invariant of a constraint database

Definition 5. Let $\sigma := \{R_1, \ldots, R_k\}$ be a signature, $\mathfrak{B}$ be a σ-database over $(\mathbb{R}, <)$, $P \subset \mathbb{R}$ a set of elements, and $\bar{b}$ a tuple of real numbers.

- The *complete atomic type of* $\bar{b}$ *over* P *with respect to* $\mathfrak{B}$, written as $atp_P^{\mathfrak{B}}(\bar{b})$, is the set of all atomic and negated atomic formulae $\varphi(\bar{x})$ over the signature $\{<, R_1, \ldots, R_k\}$ using parameters from P such that $\mathfrak{B} \models \varphi(\bar{b})$. We omit the index P if P is empty and denote by $otp^{\mathfrak{B}}(\bar{b})$, resp. $otp_P^{\mathfrak{B}}(\bar{b})$, the complete atomic type of $\bar{b}$ (over P) with respect to $\mathfrak{B}$ over the signature $\{<\}$.
- A maximally consistent set of atomic and negated atomic $\sigma \cup \{<\}$-formulae $\varphi(\bar{x})$ is a *complete atomic type (over* P*) in the variables* $\bar{x}$, if it is a complete atomic type (over P) of a tuple $\bar{b}$ with respect to a σ-expansion of $\mathfrak{A}$. We write $atp^{\mathfrak{B}}(\bar{x})$, resp. $atp_P^{\mathfrak{B}}(\bar{x})$, for a complete atomic type (over P) in the variables $\bar{x}$ over the database signature σ of $\mathfrak{B}$.

A type is an n-type if it has n free variables. We omit $\mathfrak{B}$ if it is clear from the context. When speaking about types we always mean complete atomic types throughout this chapter.

We call complete atomic types over $\sigma \cup \{<\}$ also *complete database types*. Database types are of special interest here because the database type of a tuple $\bar{b}$ determines everything we can say about $\bar{b}$ in terms of the database, especially in which database relations $\bar{b}$ stands.

Suppose $\mathfrak{B}$ is a database and $P^{\mathfrak{B}}$ the set of parameters used in its definition. Recall from the introduction that there are different ways to represent the database $\mathfrak{B}$. The set of parameters used in these representations will generally differ from $P^{\mathfrak{B}}$. We define a set of parameters, called the canonical parameters, which can be extracted from $\mathfrak{B}$ independent of its representation.

Definition 6. Suppose $\mathfrak{B} = (\mathbb{R}, <, R_1^{\mathfrak{B}}, \ldots, R_k^{\mathfrak{B}})$ is a database. The set $cp(\mathfrak{B}) \subset \mathbb{R}$ of *canonical parameters of* $\mathfrak{B}$ is the set of elements p satisfying the following condition.
For at least one n-ary relation $R \in \{R_1^{\mathfrak{B}}, \ldots, R_k^{\mathfrak{B}}\}$ there are $a_1, \ldots, a_n \in \mathbb{R}$, an $\epsilon \in \mathbb{R}, \epsilon > 0$, and an ϵ-neighbourhood $\delta = (p - \epsilon, p + \epsilon)$ of p such that one of the following holds.

- For all $q \in \delta, q < p$ and for no $q \in \delta, q > p$ we have $R\bar{a}[p/q]$. Here $R\bar{a}[p/q]$ means that all components $a_i = p$ are replaced by q.
- For all $q \in \delta, q > p$ and for no $q \in \delta, q < p$ we have $R\bar{a}[p/q]$.
- $R\bar{a}[p/q]$ holds for all $q \in \delta \backslash \{p\}$ but not for $q = p$.
- $R\bar{a}[p/q]$ holds for $q = p$ but not for any $q \in \delta \backslash \{p\}$.

Lemma 7. *All canonical parameters of* $\mathfrak{B}$ *occur explicitly in all representations of* $\mathfrak{B}$.

In particular this implies that the cardinality of $cp(\mathfrak{B})$ is bounded by the size of any representation of $\mathfrak{B}$.

We show in the next lemma that an atomic order type over $cp(\mathfrak{B})$ uniquely determines a complete database type. It follows that every two tuples realizing the same atomic order type over $cp(\mathfrak{B})$ occur in the same database relations. Thus the parameter set $cp(\mathfrak{B})$ is sufficient to define a representation of $\mathfrak{B}$.

Lemma 8. *Suppose* $\mathfrak{B}$ *is a database and* $\bar{a}, \bar{b} \in \mathbb{R}^k$ *are two* k*-tuples.*

(i) If $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\bar{a}) = otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\bar{b})$, *then* $atp^{\mathfrak{B}}(\bar{a}) = atp^{\mathfrak{B}}(\bar{b})$.
(ii) If $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(a_i) = otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(b_i)$ *for all* $1 \leq i \leq k$ *and* $otp^{\mathfrak{B}}(\bar{a}) = otp^{\mathfrak{B}}(\bar{b})$, *then* $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\bar{b}) = otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\bar{a})$.
(iii) If $P \supseteq cp(\mathfrak{B})$ *is a superset of* $cp(\mathfrak{B})$, *then* $otp^{\mathfrak{B}}_{P}(\bar{b}) = otp^{\mathfrak{B}}_{P}(\bar{a})$ *implies* $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\bar{a}) = otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\bar{b})$.

Proof. The proof of the second and third part are straightforward. To prove the first part suppose for the sake of contradiction that $atp^{\mathfrak{B}}(\bar{b})$ and $atp^{\mathfrak{B}}(\bar{a})$ differ. Then there is an atomic or negated atomic formula φ such that $\mathfrak{B} \models \varphi(\bar{a})$ but $\mathfrak{B} \not\models \varphi(\bar{b})$. If φ is of the form $x_i < x_j$, then $a_i < a_j$ but not $b_i < b_j$, which contradicts the assumption that $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\bar{b}) = otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\bar{a})$.
Now suppose φ is of the form $Rx_1 \cdots x_r$, where $r := ar(R)$. Let $\mathcal{C} := (\bar{c}_0, \bar{c}_1, \ldots, \bar{c}_k)$

be a sequence of points in $\mathbb{R}^k$, such that for $c_{ij} := b_j$ for all $j \leq i$ and $c_{ij} := a_j$ for all $j > i$. Thus $\bar{c}_0 = \bar{a}$, $\bar{c}_k = \bar{b}$, $\bar{c}_1 = (b_1, a_2, \ldots, a_k)$, $\bar{c}_2 = (b_1, b_2, a_3, \ldots, a_k)$, and so on. Further, let $L := (l_1, \ldots, l_k)$ be a sequence of lines such that the endpoints of l_i are c_{i-1} and c_i. As $\mathfrak{B} \models \varphi(\bar{a})$ but $\mathfrak{B} \not\models \varphi(\bar{b})$, there is an l_j that intersects both $R^{\mathfrak{B}}$ and $\mathbb{R}^k \backslash R^{\mathfrak{B}}$. Assume w.l.o.g. that $a_j < b_j$. Let $\bar{q} := \bar{c}_{j-1}$. Then there is a $p \in \mathbb{R}$ with $a_j < p \leq b_j$ such that $R^{\mathfrak{B}}\bar{q}$ but not $R^{\mathfrak{B}}q_1, \ldots, q_{j-1}, p, q_{j+1}, \ldots, q_k$. We claim that there is at least one canonical parameter d with $a_j \leq d \leq p$. To prove this claim, let $A := \{a \in \mathbb{R} : a_j \leq a$ and $R^{\mathfrak{B}}q_1, \ldots, q_{j-1}a'q_{j+1}, \ldots, q_k$ for all $a_j \leq a' \leq a\}$. Let d be the supremum of A. Then, by Definition 6, c is a canonical parameter and $a_j \leq d \leq p$. This proves the claim. Thus $\bar{a}$ and $\bar{b}$ do not satisfy the same complete order type over $cp(\mathfrak{B})$ which contradicts the assumption. □

One implication of the lemma is the following. Suppose we want to decide if $R\bar{a}$ holds for a tuple $\bar{a} := a_1, \ldots, a_k$ and a k-ary database relation R. The question can be answered if we know whether $R\bar{b}$ holds for a tuple $\bar{b} := b_1, \ldots, b_k$ such that $\bar{a}$ and $\bar{b}$ realize the same order type and each b_i realizes the same 1-order type over $cp(\mathfrak{B})$ as a_i. This will be the central idea in the definition of the invariant.

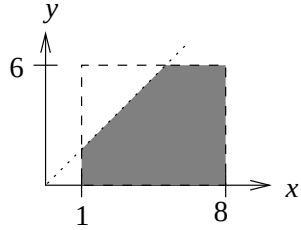
The relevant set of parameters that we need for the evaluation of ψ on a database $\mathfrak{B}$ is $P^{\mathfrak{B},\psi} := \{0, 1\} \cup cp(\mathfrak{B}) \cup P^{\psi}$. The constants 0 and 1 are included because they will be needed in the definition of the invariant.

Since P^{ψ} is finite and Definition 6 of the set of canonical parameters can obviously be formalized in first-order logic, it follows that for any fixed ψ, the set $P^{\mathfrak{B},\psi}$ is uniformly first-order definable over $(\mathfrak{R}, <, 0, 1, P^{\psi})$.

Lemma 9. *There exists a first-order formula $\delta(x)$ of vocabulary $\{<, 0, 1, P^{\psi}\} \cup \sigma$ such that for every σ-database $\mathfrak{B} = (\mathbb{R}, <, R_1^{\mathfrak{B}}, \ldots, R_k^{\mathfrak{B}})$, $P^{\mathfrak{B},\psi} = \{a \in \mathbb{R} : \mathfrak{B} \models \delta(a)\}$.*

We are now ready to define the invariant. Given a database $\mathfrak{B}$, define an equivalence relation $\sim$ on $\mathbb{R}$ such that two elements a and b are $\sim$-equivalent if and only if they realize the same 1-order type over $P^{\mathfrak{B},\psi}$. As $P^{\mathfrak{B},\psi}$ is first-order definable the equivalence relation $\sim$ is first-order definable as well. The set of equivalence classes $\mathbb{R}_{\sim}$ serves as the universe of the invariant. To complete the definition we have to specify the database relations.

Before we give the detailed definition of the relations in the invariant, we illustrate the idea by an example. Consider a database $\mathfrak{B}$ with a single binary relation S represented by $\varphi_S(x, y) := x > 1 \wedge x < 8 \wedge y > 0 \wedge y < 6 \wedge y < x$. The relation is shown in the following figure.



As explained above, the invariant depends not only on the database $\mathfrak{B}$ but also on the parameters used in the query ψ. To simplify the example let ψ be an arbitrary query with no extra parameters. Thus the set $P^{\mathfrak{B},\psi}$ consists of the four elements $0, 1, 6, 8$ and there are nine different $\sim$-equivalence classes, namely the intervals $(-\infty, 0), \{0\}, (0, 1), \{1\}, (1, 6), \{6\}, (6, 8), \{8\}$, and $(8, \infty)$. Recall that these equivalence classes form the universe of the invariant. Thus the relation S has somehow to be defined in terms of these classes. Obviously it is not enough to factorize S by $\sim$, because as $5 \sim 5.1$, the equivalence classes [5] and [5.1] are equal, but $([5.1], [5]) \in S$ and $([5], [5]) \notin S$. Thus $S_{/\sim}$ would not be well-defined.

Instead of simply factorizing a m-ary relation R by $\sim$ we consider the set C_R of $(m+1)$-tuples $([a_1], \ldots, [a_k], \rho)$, where $[a_i] \in \mathbb{R}_{/\sim}, 1 \leq i \leq m$ and ρ denotes an m-order type, such that $([a_1], \ldots, [a_m], \rho) \in C_R$ if and only if there is a $\bar{b} \in \mathbb{R}^m$ realizing ρ such that $R\bar{b}$ holds and $a_i \sim b_i$ for all $1 \leq i \leq m$. In the example above, the set C_S consists of the set of all triples $([a_1], [a_2], \rho)$ such that $[a_1] \times [a_2]$ is in the rectangle marked by the dashed line in the figure and ρ is the order type $x < y$.

The idea behind the definition of the relation in the invariant is to use the set C_R as a finite relation carrying all the information necessary to restore the original database relation R.

Note that the set $ord(m)$ of different m-order types is finite for all m. Thus we can assign to each order type $p \in ord(m)$ a binary word $\xi_m(p) \in \{0,1\}^{\ell(m)}$ where $\ell(m) := \min\{\ell : 2^\ell \geq |ord(m)|\}$. For $m = 2$ we define ξ_2 to be the encoding taking $x < y$ to 00, $x = y$ to 01, and $y < x$ to 10. Once such an encoding ξ_m is fixed, the set C_R can be represented by a set $C'_R := \{([a_1], \ldots, [a_m], \bar{t}) : ([a_1], \ldots, [a_m], \rho) \in C_R \text{ and } \xi(\rho) = \bar{t}\}$. This gives the definition of the relations in the invariant.

Definition 10. Let $\sigma := \{R_1, \ldots, R_k\}$ and $\mathfrak{B}$ be a σ-database over $(\mathbb{R}, <)$. The *invariant* $\mathfrak{B}'$ of $\mathfrak{B}$ is a finite structure with universe U over the signature $\{<, R'_1, \ldots, R'_k\}$, where

- $U := \mathbb{R}_{/\sim}$,
- $[x] < [y]$ if and only if $x < y$ and $x \not\sim y$, and
- If $R \in \sigma$ has arity m, then the corresponding relation R' has arity $m + \ell(m)$ and $R'[a_1] \ldots [a_m] t_1 \ldots t_{\ell(m)}$ holds in $\mathfrak{B}'$ iff there are $b_1, \ldots, b_m \in \mathbb{R}$ with $\xi_m(otp(\bar{b})) = t_1 \cdots t_{\ell(m)}$ so that $R_i^{\mathfrak{B}} b_1 \ldots b_m$ and $[a_i] = [b_i]$ for $1 \leq i \leq m$.

The mapping inv is defined as the function taking databases to their invariants.

We also need a function taking the finite encoding of relations back to their representation.

Definition 11. Let S be a $(m + \ell(m))$-ary relation of the form indicated by Definition 10. The function

$$\hat{\pi} : S \mapsto \varphi_S(x_1, \ldots, x_m) := \bigvee_{\bar{a}\bar{t} \in S} (\sigma_m(\bar{x}, \bar{t}) \wedge \bigwedge_{j=1}^{m} (x_j \sim a_j)),$$

maps S to a formula φ_S representing the corresponding relation on the original database. Here $\sigma_m(\bar{x}, \bar{i})$ is a formula stating that $\bar{x}$ satisfies the order type specified by $\bar{i}$. The corresponding function mapping relations on the invariant to finitely representable relations over the database is $\pi : S \mapsto \{\bar{a} \ : \ \mathfrak{A} \models \hat{\pi}(S)[\bar{a}]\}$.

Lemma 12. *The invariant $inv(\mathfrak{B})$ is an ordered finite structure whose cardinality is linearly bounded in the size of any representation of $\mathfrak{B}$.*

Proof. For any set P, the number of 1-order types over P is $2|P| + 1$. The cardinality of $inv(\mathfrak{B})$ is the number of 1-order types over $P^{\mathfrak{B},\psi}$. Recall that $|P^{\mathfrak{B},\psi}| = |cp(\mathfrak{B})| + O(1)$ (since ψ is considered fixed) and that the size of $cp(\mathfrak{B})$ is bounded by the size of any representation of $\mathfrak{B}$. □

Corollary 13. *The functions inv and $\hat{\pi}$ can be computed in* LOGSPACE.

Proof. The LOGSPACE-computability of *inv* is a direct implication of the previous lemma and a result by Kanellakis, Kuper and, Revesz stating that first-order queries can be evaluated in LOGSPACE. For $\hat{\pi}$, let S be a $(m + \ell(m))$-ary answer of a query on an invariant. As an implication of the previous lemma, the size of S is polynomially bounded in the size of any representation of $\mathfrak{B}$. All the algorithm to calculate $\hat{\pi}(S)$ has to do is to output the disjunction of the formulae $(\sigma_m(\bar{x}, \bar{i}) \wedge \bigwedge_{j=1}^{k}(x_j \sim a_j))$ for every tuple $\bar{a}\bar{i} \in S$. Clearly, this can be done in LOGSPACE. □

4.2 The transformation of the query

Having defined the invariant of a database, we have to explain how the query has to be transformed for evaluation in the invariant. This translation of the formulae follows the same ideas described above, namely to increase the arity of the relations to store the order type. While translating a formula with free variables $\{x_1, \ldots, x_m\}$ we introduce new free variables $\bar{i}$ to hold the order type.

It will be necessary to compare order types over a different number of variables. Suppose that ρ_1, ρ_2 are order types in the variables $x_1, \ldots, x_m$ and $x_1, \ldots, x_n$, respectively, where $m \leq n$. We say that ρ_2 *extends* ρ_1, if $\rho_1 \subseteq \rho_2$. This means that the order type ρ_2 behaves on $x_1, \ldots, x_m$ in the same way as ρ_1. In the query transformation we need a formula $extends_{mn}(\bar{i}, \bar{j})$ stating that $\bar{i} := i_1, \ldots, i_{\ell(m)}$ codes some m-order type ρ_1, $\bar{j} := j_1, \ldots, j_{\ell(n)}$ codes a n-order type ρ_2, and ρ_2 extends ρ_1. The formula is defined as

$$extends_{mn}(\bar{i}, \bar{j}) := \bigvee_{\rho_2 \in ord(n)} (\xi_n(\rho_2) = \bar{j} \rightarrow \bigvee_{\substack{\rho_1 \in ord(m) \\ \rho_2 \text{ extends } \rho_1}} \xi_m(\rho_1) = \bar{i}).$$

Definition 14. Suppose σ is a database schema and τ the signature of the invariants corresponding to σ-databases. Further, let $\mathcal{L}$ be a logic from {FO, FO+DTC, FO+TC, FO+LFP, FO+PFP}. $f : \mathcal{L}[\sigma] \rightarrow \mathcal{L}[\tau]$ is defined inductively as follows.

- Let $\psi(x,y) := x < y$. Then $(f\psi)(x,y,i_1,i_2) := x \leq y \wedge i_1 = 0 \wedge i_2 = 0$.
- Let $\psi(x) := x < c$. Then $(f\psi)(x,i) := x < [c] \wedge i = 0$.
- An equality $\psi(x,y) := x = y$ is translated to $(f\psi)(x,y,i_1,i_2) := x = y \wedge i_1 = 0 \wedge i_2 = 1$.
- An equality $\psi(x) := x = c$ corresponds to $(f\psi)(x,i) := x = [c] \wedge i = 0$.
- Let $\psi(x_1,\ldots,x_j) := R_i u_1 \ldots u_m$ where the u_i are either constants or variables from $\{x_1,\ldots,x_j\}$ and all x_i occur in $\{u_1,\ldots,u_m\}$. Then $(f\psi)(x_1,\ldots,x_j,i_1,\ldots,i_{\ell(j)}) := R_i' v_1 \ldots v_m \bar{i}$, where $v_r := \begin{cases} x_s & \text{if } u_r = x_s, \\ [c] & \text{if } u_r = c. \end{cases}$
- Let $\psi(x_1,\ldots,x_m) := \psi_1(y_1,\ldots,y_{m_1}) \wedge \psi_2(z_1,\ldots,z_{m_2})$, where all y_i and z_i occur in $\bar{x}$. Let $\bar{i} := i_1,\ldots,i_{\ell(m)}$, $\bar{j} := j_1,\ldots,j_{\ell(m_1)}$, and $\bar{j}' := j_1',\ldots,j_{\ell(m_2)}'$. Then $(f\psi)(\bar{x},\bar{i}) := \exists\bar{j}\exists\bar{j}'\ extends_{m_1 m}(\bar{j},\bar{i}) \wedge extends_{m_2 m}(\bar{j}',\bar{i}) \wedge (f\psi_1)(\bar{y},\bar{j}) \wedge (f\psi_2)(\bar{z},\bar{j}')$.
- For $\psi := \neg\varphi$, set $(f\psi) := \neg(f\varphi)$.
- Let $\psi(x_1,\ldots,x_m) := \exists y\ \varphi(\bar{x},y)$. Then $(f\psi)(x_1,\ldots,x_m,\bar{i}) := \exists y \exists j_1,\ldots, \exists j_{\ell(m+1)} extend_{m(m+1)}(\bar{i},\bar{j}) \wedge (f\varphi)(\bar{x},y,\bar{j})$.
- Let $\psi(\bar{u},\bar{v}) := [\mathrm{DTC}_{\bar{x},\bar{y}}\varphi(\bar{x},\bar{y})](\bar{u},\bar{v})$.
  Then $(f\psi)(\bar{u},\bar{v},\bar{i}) := [DTC_{\bar{x},\bar{y},\bar{j}}(f\varphi)(\bar{x},\bar{y},\bar{j})](\bar{u},\bar{v},\bar{i})$.
- Let $\psi(\bar{u}) := [\mathrm{LFP}_{R,\bar{x}}\varphi(R,\bar{x})](\bar{u})$.
  Then $(f\psi)(\bar{u},\bar{i}) := [\mathrm{LFP}_{R',\bar{x},\bar{j}}(f\varphi)(R',\bar{x},\bar{j})](\bar{u},\bar{i})$.
- The rules for the TC, IFP- and PFP-operators are defined analogously.

All parts of the evaluation algorithm have now been defined. The next theorem proves its correctness.

Theorem 15. *Let $\psi \in \mathcal{L}$, where $\mathcal{L}$ is one of the logics in Definition 14, be a query, $\mathfrak{B}$ be a database over $(\mathbb{R},<)$ and $\mathfrak{B}' := inv(\mathfrak{B})$ be the invariant corresponding to $\mathfrak{B}$. Then $\psi^{\mathfrak{B}} = \pi((f\psi)^{\mathfrak{B}'})$.*

Proof. The proof is by induction on the structure of the query. The argument for the boolean operations is straightforward and therefore omitted. Also, we only give the argument for the LFP-operator and omit the cases of formulae built by DTC, TC, and PFP-operators which are treated in precisely the same way.

• For $\psi(x,y) := x < y$, the set $\psi^{\mathfrak{B}}$ contains the pairs $(a,b) \in \mathbb{R}^2$ such that $a < b$. By definition, $f(\psi)$ is $x \leq y \wedge i_1 = 0 \wedge i_2 = 0$. Evaluating $(f\psi)$ on $\mathfrak{B}'$ results in the set $C := \{(a,b,i_1,i_2) \ : \ a \leq b, i_1 = 0, i_2 = 0\}$. Transforming this set with the mapping $\hat{\pi}$ yields the formula $\varphi_C(x,y) := \bigvee_{(a,b,i_1,i_2)\in C}(\sigma_2(x,y,i_1,i_2) \wedge x \sim a \wedge y \sim b)$. As i_1 and i_2 are 0 for all tuples $(a,b,i_1,i_2) \in C$, $\sigma_2(x,y,i_1,i_2)$ reduces to $x < y$ and thus $\pi(C)$ equals $\{(a,b) \in \mathbb{R}^2 \ : \ a < b\}$.

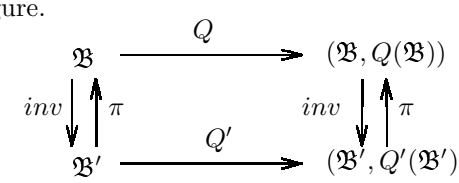
• Let $\psi(x) := x = c$. Then $(f\psi)(x,i) := x = [c] \wedge i = 0$ and $(f\psi)$ evaluates on $\mathfrak{B}'$ to the set $C := \{([c],0)\}$. Thus $\hat{\pi}(C)$ results in the formula $\varphi(x) := \sigma_1(x,0) \wedge x \sim c$. This formula is satisfied only by c because $c \in P$ and therefore the only member of $[c]$ is c itself. We get $\pi(C) := \{c\} = \psi^{\mathfrak{B}}$.

• Let $\psi(x_1,\ldots,x_j) := R_s u_1 \ldots u_m$ as in Definition 14. We assume w.l.o.g. that the first arguments of the relation are the variables and the parameters come thereafter, that is $u_1 = x_1,\ldots,u_j = x_j$ and $u_{l+1} = c_1,\ldots,u_m = c_{m-j}$. The

transformed query is $(f\psi)(x_1,\ldots,x_j,\bar{i}) := R'_s x_1 \ldots x_j[c_1]\ldots[c_{m-j}]\bar{i}$. Evaluating $f(\psi)$ on $\mathfrak{B}'$ yields the set $C := \{([a_1],\ldots,[a_j],[c_1],\ldots,[c_{m-j}],\bar{i}) \in R_s'^{\mathfrak{B}'}\}$. Now we have to show that $\pi(C) = \psi^{\mathfrak{B}}$. Suppose that $(a_1,\ldots,a_m) \in \pi(C)$. Then there is a disjunct $\varphi := \sigma_m(x_1,\ldots,x_m,\bar{i}) \wedge \bigwedge_r (x_r \sim b_r)$ in $\hat{\pi}(C)$ with $(\bar{b},\bar{i}) \in C$ and $\mathfrak{B} \models \varphi(\bar{a})$. As $(\bar{b},\bar{i}) \in R'^{\mathfrak{B}'}$ and therefore, by Definition 10, $(a_1,\ldots,a_m) \in R^{\mathfrak{B}}$ we get $\bar{a} \in \psi^{\mathfrak{B}}$. Conversely, suppose that $(a_1,\ldots,a_m) \in R^{\mathfrak{B}}$. Then $([a_1],\ldots,[a_m],\bar{i})$ is in $R'^{\mathfrak{B}'}$, where $\xi_m(otp(\bar{a})) = \bar{i}$, and $\sigma_m(\bar{x},\bar{i}) \wedge \bigwedge_r a_r \sim x_r$ occurs as a disjunct in $\hat{\pi}(C)$. Obviously this formula is satisfied by $\bar{a}$ and therefore $\bar{a} \in \pi(C)$.

- Let $\psi(x_1,\ldots,x_m) := \exists y\, \varphi(\bar{x},y)$. The transformed formula is $(f\psi)(\bar{x},\bar{i}) := \exists y \exists j_1,\ldots,j_{\ell(m+1)} extend_{m(m+1)}(\bar{i},\bar{j}) \wedge (f\varphi)(\bar{x},y,\bar{j})$. Suppose that $(a_1,\ldots,a_k) \in \psi^{\mathfrak{B}}$. This is the case if and only in there is an a_{m+1} with $(a_1,\ldots,a_m,a_{m+1}) \in \varphi^{\mathfrak{B}}$. By induction $\varphi^{\mathfrak{B}} = \pi((f\varphi)^{\mathfrak{B}'})$. Thus there is a tuple $([a_1],\ldots,[a_{m+1}],\bar{j}) \in (f\varphi)^{\mathfrak{B}'}$ and $(a_1,\ldots,a_{m+1})$ satisfies the $(m+1)$-order type ρ denoted by $\bar{j}$. This is the case if and only if there is a tuple $([a_1],\ldots,[a_m],\bar{i}) \in (f\psi)^{\mathfrak{B}'}$ such that ρ extends the order type denoted by $\bar{i}$. Thus we get that $(a_1,\ldots,a_m) \in \psi^{\mathfrak{B}}$ if and only if $([a_1],\ldots,[a_m],\bar{i}) \in (f\psi)^{\mathfrak{B}'}$, where $(a_1,\ldots,a_m)$ satisfies the order type denoted by $\bar{i}$. This implies that $\psi^{\mathfrak{B}} = \pi((f\psi)^{\mathfrak{B}'})$.
- Finally, let $\psi(\bar{u}) := [\mathrm{LFP}_{R,\bar{x}}\varphi(R,\bar{x})](\bar{u})$. straightforward. We can assume that φ does not contain an LFP-operator. The proof then is straightforward. □

Now all parts of the evaluation method are defined. We illustrate the method in the following figure.



To evaluate the query Q (considered as being fixed) in the database $\mathfrak{B}$, the invariant $\mathfrak{B}' := inv(\mathfrak{B})$ is constructed, the transformed query $Q' := f(Q)$ is evaluated in $\mathfrak{B}'$, and the result is transformed back via the map $\hat{\pi}$. By Corollary 13 the mappings inv and $\hat{\pi}$ are LOGSPACE-computable. Thus we get the following theorem.

Theorem 16. *Suppose $\mathcal{L} \in \{\mathrm{FO}, \mathrm{FO{+}DTC}, \mathrm{FO{+}TC}, \mathrm{FO{+}LFP}, \mathrm{FO{+}IFP}, \mathrm{FO{+}PFP}\}$ is a logic and $\mathcal{C}$ a complexity class so that the evaluation problem for $\mathcal{L}$ on finite databases is in $\mathcal{C}$. Then the evaluation problem for $\mathcal{L}$ on dense linear order databases is also in $\mathcal{C}$.*

4.3 Capturing complexity classes

We now use the invariant to lift the capturing results of descriptive complexity theory from finite ordered structures to dense linear order databases. The crucial observation is that $inv(\mathfrak{B})$ is interpretable in $\mathfrak{B}$. In particular, this will give us a transformation from formulae over the invariant to formulae over the database. See [9] for background on interpretations.

Definition 17. Let $\mathfrak{B} := (\mathbb{R}, <, R_1^{\mathfrak{B}}, \ldots, R_k^{\mathfrak{B}})$ a database with signature σ over $(\mathbb{R}, <)$, let $\mathfrak{B}' = inv(\mathfrak{B})$ its invariant, and τ be the signature of the invariant. The interpretation Γ interpreting $\mathfrak{B}'$ in $\mathfrak{B}$ is given by

(1) a surjective function $f_\Gamma : \mathbb{R} \to U$ defined as $f_\Gamma(x) := [x]$, and
(2) for each atomic τ-formula $\psi(x_1, \ldots, x_m)$ a formula $\psi_\Gamma(x_1, \ldots, x_m) \in \mathrm{FO}[\sigma]$ such that for all tuples $\bar{a} \in \mathbb{R}^m$: $\mathfrak{B}' \models \psi(f_\Gamma(\bar{a}))$ if and only if $\mathfrak{B} \models \psi_\Gamma(\bar{a})$.

An equality $u = v \in \mathrm{FO}[\tau]$ corresponds to $u \sim v$, where u, v denote either variables or parameters from $P^{\mathfrak{B},\psi}$ (recall that $\sim$ is first-order definable). The translations for all other atomic formulae is given according to Definition 10. That is, a formula $u < v \in \mathrm{FO}[\tau]$ corresponds to $u < v \wedge \neg u \sim v$ and $R'_s \bar{x}\bar{i}$ to $\exists \bar{y}\, R_s \bar{y} \wedge \sigma_{ar(R_s)}(\bar{y}, \bar{i}) \wedge \bigwedge_j (x_j \sim y_j)$. (Recall the definition of σ_k from Definition 11).

We can now replace in any formula ψ of vocabulary τ in first-order logic, transitive closure logic or fixed point logic the atomic formula by their corresponding formulae and obtain a σ-formula ψ_Γ. The equivalence between ψ and ψ_Γ in part (2) of the definition thus extends to arbitrary formula in these logics.

We are now ready to lift the capturing results from finite ordered structures to dense linear order databases. Clearly, every, say, FO+LFP-query ψ is invariant under automorphisms on $\mathfrak{A}$ that preserve the constants in ψ. Thus we can only hope to capture those PTIME-queries which are invariant under such automorphisms. This is made precise in the following definition.

Definition 18. A complexity class $\mathcal{C}$ is captured by a logic $\mathcal{L}$ on the class of dense order databases, if for all queries Q in $\mathcal{C}$ for which we can choose a finite set $S \subset \mathbb{R}$ such that Q commutes with every automorphism on $(\mathbb{R}, <, S)$, there is a formula ψ in $\mathcal{L}$ satisfying the following property: For all dense order databases $\mathfrak{B}$ we have that $Q(\mathfrak{B})$ is true iff $\mathfrak{B} \models \psi$.

Theorem 19. *Let $\mathcal{L}$ be a logic as in Theorem 16 and $\mathcal{C}$ be a complexity class such that $\mathcal{L}$ captures $\mathcal{C}$ on the class of finite ordered structures. Then $\mathcal{L}$ captures $\mathcal{C}$ on the class of dense order databases.*

Proof. We give the proof explicitly only for FO+LFP. The other cases can be proven analogously. We have already shown that FO+LFP $\subseteq$ PTIME. For the other direction, suppose that Q a polynomial-time computable query on dense order constraint databases of signature σ. We show that there is an FO+LFP $[\sigma]$-formula ψ_Q defining Q.

Again let τ denote the signature of the corresponding invariants. Let Q' be the query that takes invariants $inv(\mathfrak{B})$ of databases $\mathfrak{B}$ as inputs and returns as output the set $Q'(\mathfrak{B}') := \{f_\Gamma(\bar{a}) : \bar{a} \in Q(\mathfrak{B})\}$. Clearly Q' can be computed in polynomial time, since a representation of the database $\mathfrak{B}$ whose invariant is given as the input can be computed in LOGSPACE and since Q is a PTIME-query. (Note that in contrast to the algorithm of the previous section this algorithm constructs the database from the invariant and evaluates the query in the database,

whereas the algorithm in the previous section constructs the invariant from the database and then operates on the invariant.)

Since Q' is a PTIME-query on finite ordered structures, there exists by the Theorem of Immerman and Vardi (see [1, 10]) an FO+LFP $[\tau]$-formula φ that defines Q'. By the remarks above, there exists a formula $\varphi_\Gamma \in \text{FO+LFP}\,[\sigma]$ such that for all $\bar{a} \in \mathbb{R}^m$, $inv(\mathfrak{B}) \models \varphi(f_\Gamma(\bar{a}))$ iff $\mathfrak{B} \models \varphi_\Gamma(\bar{a})$. Thus $\mathfrak{B} \models \varphi_\Gamma(\bar{a})$ if and only if $\bar{a} \in Q(\mathfrak{B})$. This proves the theorem. □

The following table summarizes the relations between logics and complexity classes in the context of dense linear orders.

Logics and complexity classes in the context of dense linear orders.

$$\begin{aligned} \text{FO+DTC} &= \textsc{Logspace} \\ \text{FO+TC} &= \textsc{NLogspace} \\ \text{FO+LFP} &= \textsc{Ptime} \\ \text{FO+PFP} &= \textsc{Pspace} \end{aligned}$$

5 Summary and Further Results

In the main result of this paper we presented a general method to prove complexity bounds for query languages over dense order databases. The idea was to code the finitely represented database as a finite database and then use the evaluation algorithms available for the query language on finite databases. It turned out that this encoding can be defined by first-order formulae using only the order predicate and some very limited kind of arithmetic. It can therefore be done with very low data complexity. This method enabled us to evaluate queries for various query languages within the same complexity classes as for finite databases.

This method also works for databases defined by inequality constraints over a countable infinite set. By a simple argument based on Ehrenfeucht-Fraïssé games we can also prove that the various fixed-point logics considered before are too weak to express all LOGSPACE-computable queries.

Unfortunately the good results for dense order databases cannot be extended to linear constraint databases over the reals. As soon as we admit recursion in the query language the arithmetic over $\mathbb{N}$ becomes definable and thus the query language undecidable.

The situation changes drastically if structures with a discrete order as universe are considered. It is known that positive DATALOG-queries on discrete order databases can be evaluated in closed form (see [14]) but the data complexity is still unknown. For first-order queries a better result can be shown.

Theorem 20. *First-order queries on discrete order databases can be evaluated in* LOGSPACE.

See [13] for a proof of the theorem. In Section 3 we have shown that the data complexity of first-order queries over $(\mathbb{N}, <, +)$ is in the polynomial time hierarchy and that there are complete first-order queries for all levels of PH.

As in the case of the context structure $(\mathbb{R}, <, +)$, adding recursion to the query language leads to undecidable query languages. Of course, even first-order queries are undecidable if we also add multiplication to the context structure.

The following table summarizes the results. The NC bound for first-order queries on databases over the field of reals comes from [12]. Note that only in the case of $(\mathbb{R}, <)$ we have precise capturing results. The other cases are just complexity bounds.

	inequality	$(\mathbb{R}, <)$	$(\mathbb{R}, <, +)$	$(\mathbb{R}, <, +, \cdot)$	$(\mathbb{N}, <_c)$	$(\mathbb{N}, <, +)$	$(\mathbb{N}, <, +, \cdot)$
FO	AC^0	AC^0	NC	NC	LOGSPACE	PH	n.d.
FO+DTC	LOGSPACE	LOGSPACE	n.d.	n.d.	n.d.	n.d.	n.d.
FO+TC	NLOGSPACE	NLOGSPACE	n.d.	n.d.	n.d.	n.d.	n.d.
FO+LFP	PTIME	PTIME	n.d.	n.d.	n.d.	n.d.	n.d.
FO+PFP	PSPACE	PSPACE	n.d.	n.d.	n.d.	n.d.	n.d.

n.d. = not decidable

References

[1] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer, 1995. 67, 80

[2] E. Grädel. Subclasses of Presburger arithmetic and the polynomial-time hierarchy. *Theoretical Computer Science*, 56:289–301, 1988. 71, 71, 72

[3] E. Grädel and Y. Gurevich. Metafinite model theory. *Information and Computation*, 140:26–81, 1998. 68

[4] E. Grädel and K. Meer. Descriptive complexity theory over the real numbers. In *Mathematics of Numerical Analysis: Real Number Algorithms*, volume 32 of *AMS Lectures in Applied Mathematics*, pages 381–403. 1996. 68

[5] S. Grumbach and J. Su. Finitely representable databases. *Journal of Computer and System Sciences*, 55:273–298, 1997. 68, 68, 70, 70

[6] S. Grumbach and J. Su. Queries with arithmetical constraints. *Theoretical Computer Science*, 173:151–181, 1997. 68, 68

[7] S. Grumbach, J. Su, and C. Tollu. Linear constraint query languages: Expressive power and complexity. *Lecture Notes in Computer Science*, 960:426–446, 1995. 70

[8] D. Harel. Towards a theory of recursive structures. volume 1450 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 1998. 68

[9] W. Hodges. *Model Theory*. Cambridge University Press, 1993. 78

[10] N. Immerman. *Descriptive complexity*. Graduate Texts in Computer Science. Springer, 1998. 67, 80

[11] P. Kanellakis and D. Goldin. Constraint programming and database query languages. volume 789 of *Lecture Notes in Computer Science*, pages 96–120. Springer, 1994. 68

[12] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *Journal of Computer and Systems Sciences*, 51:26–52, 1995. 68, 68, 68, 68, 81

[13] S. Kreutzer. Descriptive complexity theory for constraint databases. Diplomarbeit, RWTH Aachen, 1999. 80

[14] P. Revesz. A closed form evaluation for datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 116:117–149, 1993. 80

[15] U. Schöning. Complexity of Presburger arithmetic with fixed quantifier dimension. *Theory of Computing Systems*, 30:423–428, 1997. 71, 72

Applicative Control and Computational Complexity

Daniel Leivant

Abstract

We establish a tight correspondence between three major complexity classes and simple syntactic restrictions on applicative programs in the simply typed lambda calculus with a recurrence operator. The syntactic restrictions considered are: recurrence arguments cannot be passed as computed values ("input-driven terms"), abstracted higher-order variables can appear at most once ("solitary terms"), and abstracted variables cannot be eventually nested ("separated terms").

We show that the functions over word algebras represented by input-driven terms are precisely the poly-time functions (a result akin to [8] (Chapter 24.2)). When input-driven recurrence is permitted over all finite types, the elementary functions are obtained (a result akin to [1]). When terms are further restricted to solitary ones, even recurrence in all finite type yields only the poly-time functions. Finally, separated terms generate exactly the poly-space functions.

The interest in the approach discussed here lies in its simplicity: the complexity characterizations are based on restricted use of standard applicative constructs, rather than a syntactic overlay as in ramified recurrence [3, 12, 15, 7, 21]. However, approaches based on ramified recurrence are more powerful than simple syntactic control, as well as more conceptually and methodologically coherent. Thus, the two approaches are complementary and of independent interest.

1 Introduction: Background and results

Intrinsic computational complexity.

Traditional computational complexity, based on resources such as computation time and space, has been matched in recent years with *"implicit"*, i.e. machine-independent and conceptually anchored, measures of complexity, such as the descriptive complexity of problems (finite model theory), the complexity of declarative programs (types, limited recurrence operators, bounding conditions), and

*Indiana University, Bloomington, IN 47405. leivant@cs.indiana.edu. Research partially supported by NSF grants CCR-9309824 and DMS-9870320. The author is grateful to Martin Hofmann, Neil Jones, Karl-Heinz Niggl and Helmut Schwichtenberg for illuminating exchanges related to this work.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 82−95, 1999.

principles needed to prove program convergence (e.g. restricted induction or set-existence).[1] Implicit characterizations of major computational complexity classes are appealing both for the theoretical insight they provide and for their potential applications. Conceptually, these characterizations link computational complexity to levels of definitional and inferential abstraction of significant independent interest. They also lend credence to the importance of the complexity classes characterized, yield insight into their nature, suggest new tools for separating them, and provide a framework for complexity theory for higher type. Practically, implicit computational complexity permits the streamlined incorporation of computational complexity into areas such as formal methods in software development, programming language theory, and database theory.

Applicative control

This paper is a contribution to an approach to implicit computational complexity which we dub *applicative control*. The idea is to syntactically restrict applicative programs so as to guarantee their computational complexity. Indeed, the restrictions sought characterize major complexity classes, such as poly-time, poly-space, and Kalmar-elementary. This approach was recently studied by Neil Jones, in his monograph [8](Chapter 24.2), and Beckmann and Weiermann, in their forthcoming [1]. Jones characterizes poly-time by recurrence with "read-only" variables, and [1] characterizes the (Kalmar-) elementary functions by a combinatory variant of Gödel's system T. Here we offer a uniform discourse for the method, giving simple proofs for the results of [8] and [1], as well as a control-based sub-calculus of [21]. Our main technical contribution is the characterization of poly-space by syntactic restrictions on applicative programs.

Relations with ramification.

The applicative control approach is closely related to the restriction of recurrence in applicative programs by an overlay of "ramification", or " data tiering". The motivation there is foundational: Recurrence schemas reflect different uses of data in computing, as was pointed out independently for recurrence [3], functional recurrence [23], lambda representability [9], and second order provability [11]. Notably, Bellantoni and Cook [3] formalized this distinction and obtained a functional characterization of poly-time that does away with the bounding condition of [5]. In [10] we outlined a predicative-finitistic critique of recurrence that leads to a generalization of [3], using a general form of ramified data and ramified recurrence. Variants of this method have been used to characterize, among others, alternating log time[4, 19]), alternating poly-log time [4], linear space [10, 6, 12], NP, the poly-time hierarchy [2], poly-space [17], (Kalmar-) elementary time [15], and NC [14][2]. Recently, Martin Hofmann [7] and Bellantoni,

[1] Recent workshops dedicated specifically to implicit computational complexity include *Implicit computational complexity in programming languages* (Baltimore, September 1998) and *Implicit computational complexity* (Trento, June/July 99).

[2] Data ramification underlies also the characterizations of poly-time by set-existence principles [11], by typed λ-calculi [16], and by a proof theoretic ramification [13].

Niggl and Schwichtenberg [21] have developed calculi that allow recurrence in higher type, and yet define only poly-time functions. To achieve this the use of tiers in types is refined by the use of modal operators, which make it possible to regulate the control flow of the computation, notably in forbidding repeated use of arguments.

These two approaches, ramification and syntactic-control, represent a trade-off. Ramification is a more powerful method, allowing more λ-terms to be typed. It is also more methodologically coherent, as it is closed under basic syntactic operations such as reductions and (type-correct) substitution. Also, ramification is natural also for proof systems, and those often map via Shönfinkel-Curry-Howard morphism to ramification of applicative programs, establishing an elegant and illuminating link between the proof theoretic and the algebraic approaches to implicit computational complexity. On the other hand, the syntactic control approach captures the combinatorial issues in hand in their simplest and purest forms. This simplicity, aside from its pedagogical interest, is of potential practical value, as it can be automatically recognized.

Apparent shortcomings of the syntactic control approach may be less consequential than may first seem. For example, the restrictions considered do not necessarily block the use of central notions of functional programming, such as modularization or internalization of patterns as higher-order functions. While the restricted terms are not necessarily closed under substitution, it is often possible to factor the construction of λ-terms that do *not* satisfy the restriction considered into composition of terms that do, implying that the given terms also fall into the complexity classes characterized, albeit not satisfying the given syntactic properties. To capture these more general cases one may require the programmer to use `let` constructs whose components satisfy the syntactic restrictions. Alternatively, the factoring of a given term may be done automatically in some cases, a task which albeit not in linear time is still of low time complexity.

In summary, ramification and syntactic-control are complementary techniques for incorporating computational complexity concerns in programming language methodology. Conceptually ramification is more coherent, and of greater theoretical interest. However, algorithmically syntactic-control is an attractive option for compile-time inference of computational complexity of functional program, and deserves independent study.

Plan and results.

The plan of the paper is as follows. We start by formulating variants of the results of [8] and [1], with as underlying formalism the simply typed λ-calculus with a recurrence operator. We offer simple proofs of these complexity characterizations, and unravel their similarity. We then consider in section 3 a characterization of poly-time that permits recurrence in higher type, namely by prohibiting multiple uses of abstracted higher-order variables. This restriction has been considered in the context of ramified formalisms in [21] and [7],

where it is expressed in a typed λ-calculus with modalities. Finally, we present in section 4 our most novel technical contribution, a control-based characterization of poly-space. Here the restriction on abstracted higher-order variables is looser: these are permitted to have multiple uses, but not in a way that can lead to their being nested after reduction. This is analogous to our work with Marion on characterization of poly-space by predicative ramified recurrence in higher type [17, 18]. Our result here seems to be of some interest also for the pure simply-typed lambda calculus (recurrence aside), as it exhibits a simple syntactic condition, viz. separated terms, for which normalization can be performed in polynomial space (for all terms of a given type complexity), compared to elementary time for arbitrary typed λ-terms.

2 Input-driven function representation

2.1 Terminology and notations

- $\mathbf{1\lambda}$ = The simply typed lambda calculus, with β-conversion.
- $\mathbf{1\lambda}(\mathbb{W}) = \mathbf{1\lambda}$ with basic functions and reductions for the algebra $\mathbb{W}$ of words over $\{0,1\}$. I.e. $\mathbf{1\lambda}$ augmented with a constant $\boldsymbol{\epsilon} : o$, constants $\mathbf{0}, \mathbf{1}, \mathbf{p} : o \to o$ (successors, predecessor) and $\mathbf{d} : o^4 \to o$ (discriminator), and with the added reductions

$$\begin{aligned} \mathbf{p}(\mathbf{0}E) &\Rightarrow E \\ \mathbf{p}(\mathbf{1}E) &\Rightarrow E \\ \mathbf{p}(\boldsymbol{\epsilon}) &\Rightarrow \boldsymbol{\epsilon} \\ \\ \mathbf{d}\boldsymbol{\epsilon}EFG &\Rightarrow E \\ \mathbf{d}(\mathbf{0}A)EFG &\Rightarrow F \\ \mathbf{d}(\mathbf{1}A)EFG &\Rightarrow G \end{aligned}$$

- $\mathbf{1\lambda R}(\mathbb{W}) = \mathbf{1\lambda}(\mathbb{W})$ augmented with a recurrence constant[3] $\mathbf{R} : o, o \to o, o \to o, o \to o, o \to o$, and the added reduction rules

$$\begin{aligned} \mathbf{R}\boldsymbol{\epsilon}E_0E_1E_\epsilon &\Rightarrow E_\epsilon \\ \mathbf{R}(\mathbf{0}A)E_0E_1E_\epsilon &\Rightarrow E_0(\mathbf{R}AE_0E_1E_\epsilon) \\ \mathbf{R}(\mathbf{1}A)E_0E_1E_\epsilon &\Rightarrow E_1(\mathbf{R}AE_0E_1E_\epsilon) \end{aligned}$$

  Note that focusing on word algebras is the natural thing to do, because computational complexity measures are defined for symbolic computing (on Turing machines), i.e. computing over word algebras.

[3] The recurrence operator is here "monotonic", i.e. iteration with parameters.

- $\mathbf{1\lambda R}^\omega(\mathbb{W})$ is $\mathbf{1\lambda R}(\mathbb{W})$, but with recurrence constants $\mathbf{R}_\tau : o, \tau \to \tau, \tau \to \tau, \tau \to \tau$ for each type τ, and reduction rules as above for them. We dub the first argument of $\mathbf{R}_\tau$ the *recurrence argument*.

Abbreviating $\mathbf{R}_o$ by $\mathbf{R}$, $\mathbf{1\lambda R}(\mathbb{W})$ is a sub-calculus of $\mathbf{1\lambda R}^\omega(\mathbb{W})$.

For each calculus we denote by $\rightleftharpoons$ the reflexive, symmetric, and transitive closure of the reduction relation $\Rightarrow$, performed on terms as well as subterms. A function $f : \mathbb{W}^r \to \mathbb{W}$ is *represented* in $\mathbf{1\lambda R}(\mathbb{W})$ (or $\mathbf{1\lambda R}^\omega(\mathbb{W})$) by a term E if for all $\mathbf{u}_1 \dots \mathbf{u}_r \in \mathbb{W}$, $E\mathbf{u}_1 \cdots \mathbf{u}_r \rightleftharpoons f(\mathbf{u}_1, \dots, \mathbf{u}_r)$ in $\mathbf{1\lambda R}(\mathbb{W})$ ($\mathbf{1\lambda R}^\omega(\mathbb{W})$, respectively). If PROP is a syntactic property of terms, we say that f is PROP-representable if f is represented by some term $E \equiv \lambda x_1 \dots x_r.F$ where F is PROP. (NB: we refer here to F, not to E.)

2.2 Poly-time and input driven representation in $\mathbf{1\lambda R}(\mathbb{W})$

A term E of $\mathbf{1\lambda R}^\omega(\mathbb{W})$ is *input driven* if no recurrence argument W in E has a free variable bound in E. For instance, $\mathbf{R}(\lambda z.\mathbf{R}z\mathbf{1}z)(\lambda z.\mathbf{R}z\mathbf{1}z)(\mathbf{1}\boldsymbol{\epsilon})$ (which represents a function of exponential growth) is not input-driven.

LEMMA **2.1** *Every poly-time function is input-driven-representable in* $\mathbf{1\lambda R}(\mathbb{W})$.

Proof. A direct proof can be obtained from the proof in [12] that two-tier recurrence captures poly-time, with insignificant modifications. Alternatively, one can prove that if E us a normal term of $\mathbf{1\lambda R}(\mathbb{W})$, with a two-tier ramification, must be input-driven. The proof is by structural induction on E. ⊣

To prove the converse implication, we use the following auxiliary notion. Say that a term E is *strictly-input-driven* if every recurrence argument in E is a free variable of E.

LEMMA **2.2** *If f is a function input-driven-representable in* $\mathbf{1\lambda R}(\mathbb{W})$ *(or* $\mathbf{1\lambda R}^\omega(\mathbb{W})$*) then f can be explicitly defined (i.e. using multi-valued composition) from functions that are strictly-input-driven-representable in* $\mathbf{1\lambda R}(\mathbb{W})$ *(in* $\mathbf{1\lambda R}^\omega(\mathbb{W})$*, respectively).*

Proof. Suppose that $f : \mathbb{W}^r \to \mathbb{W}$ is represented by $\lambda\vec{x}.\, F$, where F is input-driven. We prove the claim by induction on the size of F. Let $A_1, \dots, A_m$ be the recurrence arguments in F, which are not within the scope of more than one recurrence (as is, for instance, A in $\mathbf{R}(\mathbf{R}ABCD)xyz$). Since F is input-driven, all free variables in these A_i's are in $\vec{x}$. By induction assumption, each term $\lambda\vec{x}.A_i$ represents a function α_i that is definable by composition from functions that are strictly-input-driven-representable.

Let $z_1, \dots, z_m$ be fresh variables, and let F^* be F with A_i replaced by z_i ($i = 1 \dots m$). Then $\lambda\vec{x}\vec{z}.F^*$ strictly-input-driven represents a function $f^* : \mathbb{W}^{r+m} \to \mathbb{W}$, and $f(\vec{x}) = f^*(\vec{x}, \alpha_1(\vec{x}), \dots, \alpha_m(\vec{x}))$. Since each α_i is explicitly definable from functions that are strictly-input-driven-representable, this completes the proof. ⊣

LEMMA **2.3** *Every function strictly-input-driven-representable in* $\mathbf{1\lambda R}(\mathbb{W})$ *is poly-time.*

Proof. We shall prove in 3.12 a stronger result, but it is worth outlining a simple proof of the present statement. First, observe that if E is a strictly-input-driven normal term of $\mathbf{1\lambda R}(\mathbb{W})$, of type $o^r \to o$, then all subterms of E are of rank ≤ 1, except for terms of the form $\mathbf{R}$, $\mathbf{R}u$ or $\mathbf{R}E_0$.[4] This is straightforward by structural induction on E.

Now prove that every normal strictly-input-driven term E whose λ-closure — with respect to variables other than the input variables — has type of rank ≤ 2 maps functions that are computable in time polynomial in the input variables and constant in the (m) formal arguments to the same sort of functions. This is proved by structural induction on E, using the observation above. ⊣

Combining Lemmas 2.1, 2.2, and 2.3 we obtain

THEOREM **2.4** *A function is input-driven-representable in* $\mathbf{1\lambda R}(\mathbb{W})$ *iff it is poly-time.*

Theorem 2.4 is close to a characterization of poly-time proved by Jones [8].

2.3 Linear space and representation over $\mathbb{N}$

While we focus on recurrence over word algebras, it is of interest to consider recurrence over the algebra of unary numerals, i.e. the set $\mathbb{N}$ of the natural numbers. Here the base constant is written as $\mathbf{0}$ (rather than $\boldsymbol{\epsilon}$), and the unique successor function is denoted by $\mathbf{s}$. The discriminator function is similarly modified, to a ternary function, and the recurrence and discriminator reductions are restated accordingly. See e.g. [12] for details. Write $\mathbf{1\lambda R}(\mathbb{N})$ for the resulting calculus.

The result analogous to Theorem 2.4 is then:

THEOREM **2.5** *A function over* $\mathbb{N}$ *is input-driven-representable in* $\mathbf{1\lambda R}(\mathbb{N})$ *iff it is computable in linear space, i.e. iff it is in level* $\mathcal{E}_2$ *of the Grzegorczyk Hierarchy.*

We omit the proof, which parallels that of Theorem 2.4.

2.4 Elementary functions and input-driven representation in $\mathbf{1\lambda R}^{\omega}(\mathbb{W})$

We extend the notions of input-driven and strictly-input-driven terms to $\mathbf{1\lambda R}^{\omega}(\mathbb{W})$.

[4] The rank of a type is the count of negative nestings of $\to$: $\underline{\mathrm{rnk}}\,(0) = 0$, and $\underline{\mathrm{rnk}}\,(\sigma \to \tau) = \max(1 + \underline{\mathrm{rnk}}\,(\sigma), \underline{\mathrm{rnk}}\,(\tau))$.

A function f over $\mathbb{W}$ is *Kalmar-elementary* iff there is a $k \geq 1$ such that f is computable in time $O(2_k(n))$.[5]

LEMMA **2.6** *Every Kalmar-elementary function over $\mathbb{W}$ is input-driven-representable in $\mathbf{1\lambda R}^\omega(\mathbb{W})$.*

Proof. . Let $\tau_1 =_{\rm df} o \to o$, $\tau_{k+1} =_{\rm df} \tau_k \to \kappa_k$. Define

$$D_k =_{\rm df} \lambda f^{\tau_k}. f \circ f \equiv \lambda f \lambda v. f(f(v)),$$

a term of type τ_{k+1}. For a term A of type τ_k write A^n for the n-fold iterate of A, i.e. $A^n B$ is an abbreviation for $A(A(\cdots A(B)\cdots))$, with n occurrences of A. Thus the term $\mathbf{R}_{\tau_k}\mathbf{u}D_kD_kA$ converts to $(D_k)^nA$ (where $n = |\mathbf{u}|$), which in turns converts to A^{2^n}. Define now the input-driven term

$$E_k =_{\rm df} \mathbf{R}_k u D_k D_{k-1} \cdots D_1 \mathbf{1}\boldsymbol{\epsilon}$$

of type $o \to o$. Then $\lambda u.\underline{E_k}$ represents 2_k in unary: using $\bar{n}$ as an abbreviation for $\mathbf{1}^n\boldsymbol{\epsilon}$, we have $F_1\mathbf{u} \rightleftharpoons \overline{2_k(n)}$ whenever $|\mathbf{u}| = n$.

Now that we have a "clock" for 2_k, the simulation of computation in time $O(2_k(n))$ can be driven as in the representation proof for Kalmar-elementary functions by higher order ramified recurrence, in [15]. ⊣

LEMMA **2.7** *Every function f strictly-input-driven-representable in $\mathbf{1\lambda R}^\omega(\mathbb{W})$ is computable in elementary time.*

Proof. Let $E \equiv \lambda\vec{x}F$ represent $f : \mathbb{W}^r \to \mathbb{W}$ in $\mathbf{1\lambda R}^\omega(\mathbb{W})$. Let F arise from F by renaming variable occurrences, so that every recurrence argument is a different variable. Thus F represent a function f of arity $\geq r$, and from which f is obtained by identifying arguments (i.e. diagonalizing).

Replacing in F every subterm of the form $\mathbf{R}_\tau x$ by a variable x of type $(\tau \to \tau)^2, \tau \to \tau$, we obtain a representation of f in $\mathbf{1\lambda}(\mathbb{W})$, with input represented by Bohm-Berarducci terms at various types, and output at type o. As observed in [15], such functions are computable in elementary time. Thus f , and whence also f, are computable in elementary time. ⊣

Combining Lemmas 2.6, 2.2, and 2.7 we obtain

THEOREM **2.8** *A function is input-driven-representable in $\mathbf{1\lambda R}^\omega(\mathbb{W})$ iff it is computable in elementary time.*

[5] As usual, $2_k(n)$ is a k-deep exponential stack of 2's with n on top: $2_0(n) = n$; $2_{k+1}(n) = 2^{2_k(n)}$.

A variant of Theorem 2.8, for a combinatory calculus, was proved in [1], using a specialization of Schütte's ordinal assignment to Gödel's system T [22].

3 Poly-time and solitary representation, allowing recurrence in all finite types

We call a term E of $\mathbf{1\lambda R}^\omega(\mathbb{W})$ *solitary* if it is input-driven, and for every subterm $\lambda x.F$ of E, where x is a variable of higher type (i.e. other than o), x occurs in F at most once.

LEMMA **3.9** *Every poly-time function f over $\mathbb{W}$ is definable by composition from functions that are solitary-representable in $\mathbf{1\lambda R}^\omega(\mathbb{W})$.*

Proof. By Lemma 2.1 f is representable by the composition of input-driven terms in $\mathbf{1\lambda R}(\mathbb{W})$. Without loss of generality, these terms are normal. By structural induction, it is easy to see that normal terms of $\mathbf{1\lambda R}(\mathbb{W})$ can use no abstraction over higher-order variables. it follows that those terms are solitary. ⊣

The key property of solitary terms is the following straightforward observation.

LEMMA **3.10** *A β-reduction for a higher-order variable in a solitary term is size-reducing.*

Towards showing that every solitary-representable functions is poly-time we will use the following technical result.

LEMMA **3.11** *Let $E[\vec{u};\vec{v}]$ be a solitary $\mathbf{1\lambda R}^\omega(\mathbb{W})$ term of type $o^r \to o$ ($r \geq 0$), where $\vec{u} = u_1, \ldots, u_k$ are the recurrence variables occurring in E, each used only once, and $\vec{v} = v_1, \ldots, v_m$ are the remaining variables, all of type o. Further, assume that E has no occurrence of $\mathbf{R}_\tau$ for τ other than o, and whose redexes are either $\mathbf{R}_o$-redexes, or of type whose rank is ≤ 1. Then, for every $\mathbf{u}_1, \ldots, \mathbf{u}_k, \mathbf{v}_1, \ldots, \mathbf{v}_m, \mathbf{x}_1, \ldots, \mathbf{x}_r \in \mathbb{W}$, the term $E\ =_{\mathrm{df}} E[\vec{\mathbf{u}};\vec{\mathbf{v}}]\vec{\mathbf{x}}$ reduces to normal form within $|\mathbf{u}_1| \cdot |\mathbf{u}_2| \cdots |\mathbf{u}_k| \cdot \underline{\mathrm{size}}\,(E) + r$ steps.*

Proof. By induction on E. The cases where E is one of $\boldsymbol{\epsilon}$, $\mathbf{0}$, $\mathbf{1}$, $\mathbf{p}$, or $\mathbf{d}$ are straightforward. The case where $E = \lambda x.F$ is trivial from the induction assumption applied to F.

If E is of the form $\mathbf{R}_0 u_i F_0 F_1$ (of type $o \to o$), then $E\ = \mathbf{R}_0 \mathbf{u}_i F_0[\vec{\mathbf{u}};\vec{\mathbf{v}}]F_1[\vec{\mathbf{u}};\vec{\mathbf{v}}]\mathbf{x}$. Say $\mathbf{u}_i$ is $\mathbf{u}_1$. The convergence property is proved by induction on $|\mathbf{u}_1|$. Note that by our assumptions on E, u_1 is not a recurrence argument in either F_0 or F_1. If $\mathbf{u}_1 = \boldsymbol{\epsilon}$, the statement is trivial. If $\mathbf{u}_1 = \mathbf{0w}$, then E reduces in one step

to $F_0[\vec{\mathbf{u}};\vec{\mathbf{v}}]G$, where $G = \mathbf{R}_0\mathbf{w}F_0[\vec{\mathbf{u}};\vec{\mathbf{v}}]\,F_1[\vec{\mathbf{u}};\vec{\mathbf{v}}]\mathbf{x}$, and by induction assumption G converges to some $\mathbf{w} \in \mathbb{W}$ within $(|\mathbf{u}_1|-1)\cdot|\mathbf{u}_2|\cdots|\mathbf{u}_k|\cdot\underline{\text{size}}\,(G)$ steps. By induction assumption for F_0, $F_0\,\mathbf{w}$ reduces to normal form within $|\mathbf{u}_2|\cdots|\mathbf{u}_k|\cdot\underline{\text{size}}\,(F_0)$ steps. Adding these two values we get the desired bound for E . The case where $\mathbf{u}_1 = \mathbf{1}\mathbf{w}$ is similar.

If E is any other term of the form FG, then the conditions on E imply that F is of type $o^{r+1}\to o$ and G is of type o $(r \geq 0)$. We have $E = F[\vec{\mathbf{u}};\vec{\mathbf{v}}]\,G[\vec{\mathbf{u}};\vec{\mathbf{v}}]\vec{\mathbf{x}}$. By induction assumption G reduces to some $\mathbf{w} \in \mathbb{W}$ within $|\mathbf{u}_1|\cdots|\mathbf{u}_k|\cdot\underline{\text{size}}\,(G)$ steps, and then $F[\vec{\mathbf{u}};\vec{\mathbf{v}}]\mathbf{w}\vec{\mathbf{x}}$ reduces within $|\mathbf{u}_1|\cdots|\mathbf{u}_k|\cdot\underline{\text{size}}\,(F)$ steps. The result follows.

Finally, if $E = \lambda x.F$, then the statement of the lemma is trivial by induction assumption for F. ⊣

LEMMA **3.12** *If a function is solitary-representable in* $\mathbf{1\lambda R}^\omega(\mathbb{W})$ *then it is poly-time.*

Proof. Suppose that f is an r-ary function over $\mathbb{W}$ representable by $\lambda x_1,\ldots,x_r.F$, where F is solitary. By Lemma 2.2 we may assume that F is strictly-input-driven. By separating variable-occurrences into distinct variables, we may also assume, w.l.o.g., that no x_i occurs in F as recurrence argument more than once. Let $y_1,\ldots,y_q$ be the x_i's used for higher order recurrence, and $z_1,\ldots,z_k$ the x_i's used for recurrence in type o. Given $\mathbf{y}_1,\ldots,\mathbf{y}_q,\mathbf{z}_1,\ldots,\mathbf{z}_k \in \mathbb{W}$, consider the term $F = \{\vec{\mathbf{y}},\vec{\mathbf{z}}/\vec{y},\vec{z}\}F$. Unfolding the higher order recurrences in F can be done in time polynomial in $|\vec{\mathbf{y}}|$, yielding a term F of size polynomial in $\vec{\mathbf{y}}$. By Lemma 3.10 higher order β-redexes can be eliminated in time bounded by the size of F , yielding a smaller term F that satisfies the conditions of Lemma 3.11. The Lemma follows. ⊣

Combining Lemmas 3.9 and 3.12 we obtain

THEOREM **3.13** *The functions solitary-representable in* $\mathbf{1\lambda R}^\omega(\mathbb{W})$ *are precisely the poly-time functions.*

4 Separated representation and poly-space

4.1 Representability of poly-space

If $FG_1\cdots G_r$ is a subterm-occurrence in a term E we say that the term-occurrences G_i are *in the scope* of term-occurrence F and of its subterms. We call a term E of $\mathbf{1\lambda R}^\omega(\mathbb{W})$ *separated* if every two occurrences of a variable have the same bounded variables in their scope. For example, $\lambda x^{o\to o}\lambda z^o.x(x(z))$ is not separated, because the external occurrence of x has the bounded variable x in its scope, whereas the internal one does not. Similarly, $(\lambda y.\lambda z.x(y(z)))\,x$ is

not separated, because the first occurrence of x has y and z in its scope, whereas the second does not.[6]

LEMMA **4.14** *Every poly-space function is separated-representable in* $\mathbf{1\lambda R}^{\omega}(\mathbb{W})$.

Proof. It is proved in [17] that the functions computable in alternating polynomial time, i.e. the poly-space functions, are representable using a weak form of second-order ramified recurrence (which suffices to capture ramified recurrence with parameter substitution). Inspection of that construction shows that the terms constructed are in fact separated. ⊣

4.2 Reductions on separated terms

LEMMA **4.15** *If E is separated, and E reduces in one step to E in* $\mathbf{1\lambda R}^{\omega}(\mathbb{W})$, *then E is separated.* ⊣

Consider terms of $\mathbf{1\lambda R}^{\omega}(\mathbb{W})$ represented by their syntax tree, i.e. 1-2 trees where leaves are labeled by variables or constants, nodes with out-degree 1 represent λ-abstraction and labeled with the abstracted variable, and nodes with out-degree 2 labeled with APP and represent application. A β-reduction $(\lambda x.E)F \Rightarrow \{F/x\}E$ is represented by substituting in the tree of E, for each leaf labeled with x, the tree of F with its root node identified with that leaf. It is advantageous to refrain from doing the latter, and keeping the root of the tree for F as a descendent of the leaf for x: for one, this will preserve the tree addresses of internal nodes of E in $\{F/x\}E$. We do this by allowing representing tree to have nodes marked with NOOP.[7] Note that in course of a normalization sequence of a term, a given address can undergo two sorts of change: (a) from being empty (outside the tree) to becoming labeled with a variable, constant, or APP; (b) from being labeled, to being marked NOOP.

In a β-reduction as above, we call a node in a copy of F in $\{F/x\}E$ an *offspring* of the corresponding node in the main argument F in $(\lambda x.E)F$. In the course of a reduction sequence, nodes can have an ever increasing number of *descendents*, i.e. offsprings, the offsprings of those, etc. Just as we defined separated variables, let us say that two *nodes* in a syntax tree are *separated* if they have the same bound variables in their scope. We clearly have

LEMMA **4.16** *Let E be a separated term that reduces to E ; then the offsprings of each node, created by the reduction, are separated, and if α and β are separated nodes in E, then each offspring of α in E is separated from every offspring of β in E .*

[6] The notion of separated terms is related to Karl-Heinz Niggl's notion of "scope equivalence", see. e.g. [21].

[7] NB: NOOP is an algorithmic device used in the representation of λ-terms, and are *not* part of the syntax of the calculus itself.

From this we conclude

LEMMA **4.17** *If E reduces to E' in $\mathbf{1\lambda R}^\omega(\mathbb{W})$, and α and β are nodes in E' that are descendents of the same node in E, then they are not on the same branch.*

Proof. By induction on the length of the reduction. Suppose that E' satisfies the conclusion, and E reduces in one step to E'. Suppose that two descendents α and β of the same node in E are offsprings of α_0 and β_0 respectively, in E', and α is above β in E'. By Lemma 4.16 α_0 and β_0 are separated, and by induction assumption they are not on the same branch in E. Thus, the reduction from E to E' must place the offspring α of α_0 over β. This can happen only if the eigen-variable of the reduction is in the scope of α in E', but not in the scope of β. This contradicts the separation of α and β in E'. ⊣

We conclude

LEMMA **4.18** *If a separated term E reduces to E' by β-reductions, then the height of the syntax tree of E' is bound by the size of E.*

Because addresses in syntax-trees can change their label at most twice in the course of a reduction sequence (as noted above), Lemma 4.18 implies

LEMMA **4.19** *If E is a separated, then all β-redexes in E are eliminable in time exponential in the size of E.*

Using Lemmas 4.18 and 4.19 we obtain

PROPOSITION **4.20** *There is a linear-space algorithm for eliminating from separated terms all β-redexes.*

Proof. For a separated term E of size n, the algorithm uses a counter A of length n for addresses in the syntax-trees of terms (of height $\leq n$, i.e. size $\leq 2^n$), and a counter T for reduction-time (of $\leq 2^n$ steps). The value returned is the label at address A after T reductions. The details are tedious but straightforward. ⊣

4.3 Poly-space computability of separated-representable functions

LEMMA **4.21** *Every function separated-representable in $\mathbf{1\lambda R}^\omega(\mathbb{W})$ is poly-space.*

Proof. Suppose f is represented by $\lambda x_1 \ldots x_r.\, F$, where F is separated, whence also input-driven. W.l.o.g., F is strictly-input-driven. Given $\mathbf{u}_1, \ldots, \mathbf{u}_r \in \mathbb{W}$, consider the term $F' = \{\vec{\mathbf{u}}/\vec{x}\}F$. Unfolding all recurrences in F' yields a term F'' of height polynomial in $|\vec{\mathbf{u}}|$, whose local description can be given in poly-space. By Proposition 4.20 all redexes in F'' can be eliminated from F'' in space

linear in the size of F , i.e. polynomial in $|\vec{\mathbf{u}}|$. The result is a normal term of type o, which albeit potentially of length exponential in the input, is bitwise in poly-space. ⊣

Combining Lemmas 4.14 and 4.21 yields

THEOREM **4.22** *The functions separated-representable in* $\mathbf{1\lambda R}^{\omega}(\mathbb{W})$ *are precisely the poly-space functions.*

5 Directions for further research

One obvious project is to further expand the development presented here to syntax-restricted characterizations of additional classes, emulating such characterizations using data ramification. In particular, it seems that the characterizations of alternating log-time and of NC by ramified tree recurrence [19, 14] can be paralleled to yield analogous results for syntax-restricted recurrence on trees.

One of the benefits of resource-independent characterizations of complexity classes is that they generalize to higher type much more easily than machine-based definitions of such classes. For instance, we believe that the functionals solitary-representable in $\mathbf{1\lambda R}^{\omega}(\mathbb{W})$ form precisely the class BFF. A result analogous to the characterization of probabilistic polynomial time in [20] seems also to be at hand.

Most importantly, one would wish to see that the simplicity of the syntactic-control approach is put to use in actual implementations of functional programming languages. All syntactic properties considered here can be trivially checked in linear time. As mentioned in the introduction, one might wish to extend the applicability of the method by crafting algorithms that automatically factor given untyped terms that do not satisfy the syntactic restrictions considered into compositions and closures under substitution of terms that do satisfy those restrictions. While such algorithms would not be in linear time, they are still likely to be algorithmically more efficient, in both worst-case and average-case, than type inference algorithms for ramified systems, in particular ones with modal operators.

References

[1] Arnold Beckmann and Andreas Weiermann. Characterizing the elementary recursive functions by a fragment of Gödel's T. Manuscript, www.math.uni-muenster.de/math/inst/logik/publ/pap/20.html.

[2] S. Bellantoni. Predicative recursion and the polytime hierarchy. In Peter Clote and Jeffery Remmel, editors, *Feasible Mathematics II*, *Perspectives in Computer Science*, pages 15–29. Birkhäuser, 1994.

[3] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[4] S. Bloch. Functional characterizations of uniform log-depth and polylog-depth circuit families. In *Proceedings of the Seventh Annual Structure in Complexity Theory Conference*, pages 193–206. IEEE Computer Society Press, 1992.

[5] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam, 1962.

[6] W.G. Handley. Bellantoni and Cook's characterization of polynomial time functions. Typescript, August 1992.

[7] Martin Hofmann. Type systems for polynomial-time computation. Habilitationsschrift, 1998.

[8] N. Jones. *Computability and Complexity from a Programming Perspective*. MIT Press, Cambridge, MA, 1997.

[9] D. Leivant. Subrecursion and lambda representation over free algebras. In Samuel Buss and Philip Scott, editors, *Feasible Mathematics*, Perspectives in Computer Science, pages 281–291. Birkhauser-Boston, New York, 1990.

[10] D. Leivant. Stratified functional programs and computational complexity. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 325–333, New York, 1993. ACM.

[11] D. Leivant. A foundational delineation of poly-time. *Information and Computation*, 1994. (Special issue of selected papers from LICS'91, edited by G. Kahn).

[12] D. Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In Peter Clote and Jeffrey Remmel, editors, *Feasible Mathematics II*, pages 320–343. Birkhäuser-Boston, New York, 1994.

[13] D. Leivant. Intrinsic theories and computational complexity. In D. Leivant, editor, *Logic and Coputational Complexity*, volume 960 of *LNCS*, pages 177–194. Springer-Verlag, Berlin, 1995.

[14] D. Leivant. A characterization of NC by tree recurrence. In *Thirty Ninth Annual Symposium on Foundations of Computer Science (FOCS)*, pages 716–724, Los Alamitos, CA, 1998. IEEE Computer Society.

[15] D. Leivant. Ramified recurrence and computational complexity III: Higher type recurrence and elementary complexity. *Annals of Pure and Applied Logic*, 1998. Special issue in honor of Rohit Parikh's 60th Birthday; editors: M. Fitting, R. Ramanujam and K. Georgatos.

[16] D. Leivant and J.-Y. Marion. Lambda-calculus characterizations of poly-time. *Fundamenta Informaticae*, 19:167–184, 1993. Special Issue: Lambda Calculus and Type Theory (editor: J. Tiuryn).

[17] D. Leivant and J.-Y. Marion. Ramified recurrence and computational complexity II: substitution and poly-space. In L. Pacholski and J. Tiuryn, editors, *Proceedings of CSL 94*, pages 486–500. LNCS 933, Springer Verlag, 1995.

[18] D. Leivant and J.-Y. Marion. Ramified recurrence and computational complexity IV: Predicative functionals and poly-space. *Information and Computation*, 1999.

[19] D. Leivant and J.-Y. Marion. Ramified recurrence and computational complexity V: linear tree recurrence and alternating log-time. *Theoretical Computer Science*, 1999. Special issue for CAAP'98, editor: M. Duachet.

[20] J. Mitchell, M. Mithcell, and A. Scedrov. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *Thirty Ninth Annual Symposium on Foundations of Computer Science (FOCS)*, pages 725–733, Los Alamitos, CA, 1998. IEEE Computer Society.

[21] K.-H. Niggl S. Bellantoni and H. Schwichtenberg. Higher type ramification and polynomial time. manuscript, to appear, 1999.

[22] Kurt Schütte. *Proof Theory*. Springer-Verlag, 1977.

[23] H. Simmons. The realm of primitive recursion. *Archive for Mathematical Logic*, 27:177–188, 1988.

Applying Rewriting Techniques to the Verification of Erlang Processes

Thomas Arts[1] and Jürgen Giesl[2]

[1] Computer Science Laboratory, Ericsson Utvecklings AB, Box 1505, 125 25 Älvsjö, Sweden, E-mail: thomas@cslab.ericsson.se

[2] Dept. of Computer Science, Darmstadt University of Technology, Alexanderstr. 10, 64283 Darmstadt, Germany, E-mail: giesl@informatik.tu-darmstadt.de

Abstract. Erlang is a functional programming language developed by Ericsson Telecom which is particularly well suited for implementing concurrent processes. In this paper we show how methods from the area of term rewriting are presently used at Ericsson. To verify properties of processes, such a property is transformed into a termination problem of a conditional term rewriting system (CTRS). Subsequently, this termination proof can be performed automatically using *dependency pairs*. The paper illustrates how the dependency pair technique can be applied for termination proofs of *conditional* TRSs. Secondly, we present two refinements of this technique, viz. *narrowing* and *rewriting dependency pairs*. These refinements are not only of use in the industrial application sketched in this paper, but they are generally applicable to arbitrary (C)TRSs. Thus, in this way dependency pairs can be used to prove termination of even more (C)TRSs automatically.

Keywords: program verification, rewriting, termination, automated deduction

1 Introduction

In a patent application [HN99], Ericsson developed a new protocol for distributed telecommunication processes. This paper originates from an attempt to verify this protocol's implementation written in Erlang. To save resources and to increase reliability, the aim was to perform as much as possible of this verification automatically. Model checking techniques were not applicable, since the property to be proved requires the consideration of the infinite state space of the process. A user guided approach based on theorem proving was successful, but very labour intensive [AD99]. We describe one of the properties which had to be verified in Sect. 2 and show that it can be represented as a non-trivial termination problem of a CTRS. But standard techniques (see e.g. [Der87,Ste95,DH95]) and even recent advances like the dependency pair technique [AG97a,AG97b,AG98, AG99] could not perform the required termination proof automatically.

In Sect. 3 we show that termination problems of CTRSs can be reduced to termination problems of unconditional TRSs. After recapitulating the basic notions of dependency pairs in Sect. 4, we present two important extensions, viz. *narrowing* (Sect. 5) and *rewriting* dependency pairs (Sect. 6) which are particularly useful in the context of CTRSs. With these refinements, the dependency pair approach could solve the process verification problem automatically.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 96–110, 1999.

2 A Process Verification Problem

We have to prove properties of a process in a network. The process receives messages which consist of a list of data items and an integer `M`. For every item in the list, the process computes a new list of data items. For example, the data items could be telephone numbers and the process could generate a list of calls to that number on a certain date. The resulting list may have arbitrary length, including zero. The integer `M` in the message indicates how many items of the newly computed list should be sent to the next process. The restriction on the number of items that may be sent out is imposed for practical optimization reasons.

Of course, the process may have computed more than `M` new items and in that case, it stores the remaining answers in an accumulator (implemented by an extra argument `Store` of the process). However, whenever it has sent the first `M` items to the next process, our process may receive a new message. To respond to the new message, the process first checks whether its store already contains at least `M` items. In this case, it sends the first `M` items from its store and depending on the incoming message, probably some new items are computed afterwards. Otherwise, if the store contains fewer than `M` items, then the next process has to wait until the new items are computed. After this computation, the first `M` items from the newly obtained item list and the store are sent on to the next process. Again, those items that our process could not send out are stored in its accumulator.

Finally, in order to empty the store, the empty list is sent to our process repeatedly. In the end, so is the claim, this process will send the empty list as well. This article describes how we are able to formally and automatically verify this claim. The Erlang code is given below (because of space limitations the code for obvious library functions like `append` and `leq` is not presented).

```
process(NextPid,Store) ->
 receive {Items,M} ->
  case leq(M,length(Store)) of
  true -> {ToSend,ToStore} = split(M,Store),
          NextPid!{ToSend,M},
          process(NextPid,append(map_f(self(),Items),ToStore));
  false ->{ToSend,ToStore} = split(M,append(map_f(self(),Items),Store)),
          NextPid!{ToSend,M},
          process(NextPid,ToStore)
  end
 end.

map_f(Pid,nil) -> nil;
map_f(Pid,cons(H,T)) -> append(f(Pid,H),map_f(Pid,T)).
```

For a list `L`, `split(M,L)` returns a pair of lists $\{L_1, L_2\}$ where L_1 contains the first `M` elements (or `L` if its length is shorter than `M`) and L_2 contains the rest of `L`. The command '`!`' denotes the sending of data and `NextPid!{ToSend,M}` stands for sending the items `ToSend` and the integer `M` to the process with the identifier `NextPid`. A process can obtain its own identifier by calling the function `self()`. For every item in the list `Items`, the function `map_f(Pid,Items)` computes new

data items by means of the function `f(Pid,Item)`. So the actual computation that `f` performs depends on the process identifier `Pid`. Hence, to compute new data items for the incoming `Items`, our process has to pass its own identifier to the function `map_f`, i.e., it calls `map_f(self(),Items)`.

Note that this process itself is not a terminating function: in fact, it has been designed to be non-terminating. Our aim is not to prove its termination, but to verify a certain property, which can be expressed in terms of termination. As part of the correctness proof of the software, we have to prove that if the process continuously receives the message {`nil`,`M`} for any integer `M`, then eventually the process will send the message {`nil`,`M`} as well. This property must hold independent of the value of the store and of the way in which new data items are generated from given ones. Therefore, `f` has been left unspecified, i.e., `f` may be any terminating function which returns a list of arbitrary length.

The framework of term rewriting [DJ90,BN98] is very useful for this verification. We prove the desired property by constructing a CTRS containing a binary function process whose arguments represent the stored data items `Store` and the integer `M` sent in the messages. In this example, we may abstract from the process communication. Thus, the Erlang function `self()` becomes a constant and we drop the send command (`!`) and the argument `NextPid` in the CTRS. Since we assume that the process constantly receives the message {`nil`,`M`}, we hard-code it into the CTRS. Thus, the variable `Items` is replaced by nil. As we still want to reason about the variable `M`, we added it to the arguments of the process. To model the function `split` (which returns a *pair* of lists) in the CTRS, we use separate functions fstsplit and sndsplit for the two components of `split`'s result. Now the idea is to force the function process to terminate if `ToSend` is the empty list nil. So we only continue the computation if application of the function empty to the result of fstsplit yields false. Thus, if all evaluations w.r.t. this CTRS terminate, then the original process eventually outputs the demanded value.

$$\mathsf{leq}(m, \mathsf{length}(\mathit{store})) \to \mathsf{true},\ \ \mathsf{empty}(\mathsf{fstsplit}(m, \mathit{store})) \to \mathsf{false} \ \mid \\ \mathsf{process}(\mathit{store}, m) \to \mathsf{process}(\mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), \mathsf{sndsplit}(m, \mathit{store})), m) \quad (1)$$

$$\mathsf{leq}(m, \mathsf{length}(\mathit{store})) \to^* \mathsf{false},\ \ \mathsf{empty}(\mathsf{fstsplit}(m, \mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), \mathit{store}))) \to^* \mathsf{false} \ \mid \\ \mathsf{process}(\mathit{store}, m) \to \mathsf{process}(\mathsf{sndsplit}(m, \mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), \mathit{store})), m) \quad (2)$$

The auxiliary Erlang functions as well as the functions for empty, fstsplit, and sndsplit are straightforwardly expressed by unconditional rewrite rules.

$$\begin{aligned}
\mathsf{length}(\mathsf{nil}) &\to \mathsf{0} & \mathsf{sndsplit}(\mathsf{0}, x) &\to x \\
\mathsf{length}(\mathsf{cons}(h,t)) &\to \mathsf{s}(\mathsf{length}(t)) & \mathsf{sndsplit}(\mathsf{s}(n), \mathsf{nil}) &\to \mathsf{nil} \\
\mathsf{fstsplit}(\mathsf{0}, x) &\to \mathsf{nil} & \mathsf{sndsplit}(\mathsf{s}(n), \mathsf{cons}(h,t)) &\to \mathsf{sndsplit}(n,t) \\
\mathsf{fstsplit}(\mathsf{s}(n), \mathsf{nil}) &\to \mathsf{nil} & \mathsf{empty}(\mathsf{nil}) &\to \mathsf{true} \\
\mathsf{fstsplit}(\mathsf{s}(n), \mathsf{cons}(h,t)) &\to \mathsf{cons}(h, \mathsf{fstsplit}(n,t)) & \mathsf{empty}(\mathsf{cons}(h,t)) &\to \mathsf{false} \\
\mathsf{app}(\mathsf{nil}, x) &\to x & \mathsf{leq}(\mathsf{0}, m) &\to \mathsf{true} \\
\mathsf{app}(\mathsf{cons}(h,t), x) &\to \mathsf{cons}(h, \mathsf{app}(t,x)) & \mathsf{leq}(\mathsf{s}(n), \mathsf{0}) &\to \mathsf{false} \\
\mathsf{map_f}(\mathit{pid}, \mathsf{nil}) &\to \mathsf{nil} & \mathsf{leq}(\mathsf{s}(n), \mathsf{s}(m)) &\to \mathsf{leq}(n,m) \\
\mathsf{map_f}(\mathit{pid}, \mathsf{cons}(h,t)) &\to \mathsf{app}(\mathsf{f}(\mathit{pid}, h), \mathsf{map_f}(\mathit{pid}, t)) & &
\end{aligned}$$

The rules for the Erlang function `f` are not specified, since we have to verify the desired property for *any* terminating function `f`. However, as Erlang has an eager (call-by-value) evaluation strategy, if a terminating Erlang function `f` is straightforwardly transformed into a (C)TRS (such as the above library functions), then any evaluation w.r.t. these rules is finite. Now to prove the desired property of the Erlang process, we have to show that the whole CTRS with all its extra rules for the auxiliary functions only permits finite evaluations.

The construction of the above CTRS is rather straightforward, but it presupposes an understanding of the program and the verification problem and therefore it can hardly be mechanized. But after obtaining the CTRS, the proof that any evaluation w.r.t. this CTRS is finite should be done automatically.

In this paper we describe an extension of the dependency pair technique which can perform such automatic proofs. Moreover, this extension is of general use for termination proofs of TRSs and CTRSs. Hence, our results significantly increase the class of systems where termination can be shown mechanically.

3 Termination of Conditional Term Rewriting Systems

A CTRS is a TRS where conditions $s_1 = t_1, \ldots, s_n = t_n$ may be added to rewrite rules $l \to r$. In this paper, we restrict ourselves to CTRSs where all variables in the conditions s_i, t_i also occur in l. Depending on the interpretation of the equality sign in the conditions, different rewrite relations can be associated with a CTRS, cf. e.g. [Kap84,BK86,DOS88,BG89,DO90,Mid93,Gra94,SMI95,Gra96a, Gra96b]. In our verification example, we transformed the problem into an *oriented* CTRS [SMI95], where the equality signs in conditions of rewrite rules are interpreted as reachability ($\to$). Thus, we denote rewrite rules by

$$s_1 \to \ t_1, \ldots, s_n \to \ t_n \ \mid \ l \to r. \tag{3}$$

In fact, we even have a *normal* CTRS, because all t_i are ground normal forms w.r.t. the TRS which results from dropping all conditions.

A reduction of $C[l\sigma]$ to $C[r\sigma]$ with rule (3) is only possible if $s_i\sigma$ reduces to $t_i\sigma$ for all $1 \le i \le n$. Formally, the rewrite relation $\to_{\mathcal{R}}$ of a CTRS $\mathcal{R}$ can be defined as $\to_{\mathcal{R}} = \bigcup_{j \ge 0} \to_{\mathcal{R}_j}$, where $\mathcal{R}_0 = \emptyset$ and $\mathcal{R}_{j+1} = \{l\sigma \to r\sigma \mid s_i\sigma \to_{\mathcal{R}_j} t_i\sigma$ for all $1 \le i \le n$ and some rule (3) in $\mathcal{R}\}$, cf. e.g. [Mid93,Gra96b].

A CTRS $\mathcal{R}$ is *terminating* iff $\to_{\mathcal{R}}$ is well founded. But termination is not enough to ensure that every evaluation with a CTRS is finite. For example, assume that evaluation of the condition $\mathsf{leq}(m, \mathsf{length}(store))$ in our CTRS would require the reduction of $\mathsf{process}(store, m)$. Then evaluation of $\mathsf{process}(store, m)$ would yield an infinite computation. Nevertheless, $\mathsf{process}(store, m)$ could not be rewritten further and thus, the CTRS would be terminating. But in this case, the desired property would *not* hold for the original Erlang process, because this would correspond to a deadlock situation where no messages are sent out at all.

For that reason, instead of *termination* one is often much more interested in *decreasing* CTRSs [DOS88]. In this paper, we use a slightly modified notion of

decreasingness, because in our evaluation strategy conditions are checked from left to right, cf. [WG94]. Thus, the i-th condition $s_i \to t_i$ is only checked if all previous conditions $s_j \to t_j$ for $1 \leq j < i$ hold.

Definition 1 (Left-Right Decreasing). *A CTRS $\mathcal{R}$ is* left-right decreasing *if there exists a well-founded relation $>$ containing the rewrite relation $\to_{\mathcal{R}}$ and the subterm relation $\rhd$ such that $l\sigma > s_i\sigma$ holds for all rules like (3), all $i \in \{1,\ldots,n\}$, and all substitutions σ where $s_j\sigma \to_{\mathcal{R}} t_j\sigma$ for all $j \in \{1,\ldots,i-1\}$.*

This definition of left-right decreasingness exactly captures the finiteness of recursive evaluation of terms. (Obviously, decreasingness implies left-right decreasingness, but not vice versa.) Hence, now our aim is to prove that the CTRS corresponding to the Erlang process is left-right decreasing.

A standard approach for proving termination of a CTRS $\mathcal{R}$ is to verify termination of the TRS $\mathcal{R}$ which results from dropping all conditions (and for decreasingness one has to impose some additional demands). But this approach fails for CTRSs where the conditions are necessary to ensure termination. This also happens in our example, because without the conditions $\mathsf{empty}(\ldots) \to \mathsf{false}$ the CTRS is no longer terminating (and thus, not left-right decreasing either).

A solution for this problem is to transform CTRSs into *unconditional* TRSs, cf. [DP87,GM87,Mar96]. For unconditional rules, let $\mathrm{tr}(\, l \to r \,) = \{l \to r\}$. If ϕ is a conditional rule, i.e., $\phi =$ '$s_1 \to t_1, \ldots, s_n \to t_n \,|\, l \to r$', we define $\mathrm{tr}(\phi) =$

$$\{l \to \mathsf{if}_{1,\phi}(\boldsymbol{x}, s_1)\,\} \cup \{\mathsf{if}_{i,\phi}(\boldsymbol{x}, t_i) \to \mathsf{if}_{i+1,\phi}(\boldsymbol{x}, s_{i+1}) \,|\, 1 \leq i < n\} \cup \{\mathsf{if}_{n,\phi}(\boldsymbol{x}, t_n) \to r\},$$

where $\boldsymbol{x}$ is the tuple of all variables in l and the if's are new function symbols. To ease readability we often just write if_n for some $n \in \mathbb{N}$ where if_n is a function symbol which has not been used before.

Let $\mathcal{R}^{\mathrm{tr}} = \bigcup_{\phi \in \mathcal{R}} \mathrm{tr}(\phi)$. For CTRSs without extra variables, $\mathcal{R}^{\mathrm{tr}}$ is indeed an (unconditional) TRS. (An extension to *deterministic* CTRSs [BG89] with extra variables is also possible.) The transformation of Rule (1) results in

$$\mathsf{process}(\mathit{store}, m) \to \mathsf{if}_1(\mathit{store}, m, \mathsf{leq}(m, \mathsf{length}(\mathit{store}))) \tag{4}$$

$$\mathsf{if}_1(\mathit{store}, m, \mathsf{true}) \to \mathsf{if}_2(\mathit{store}, m, \mathsf{empty}(\mathsf{fstsplit}(m, \mathit{store}))) \tag{5}$$

$$\mathsf{if}_2(\mathit{store}, m, \mathsf{false}) \to \mathsf{process}(\mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), \mathsf{sndsplit}(m, \mathit{store})), m). \tag{6}$$

Now we aim to prove termination of $\mathcal{R}^{\mathrm{tr}}$ instead of $\mathcal{R}$'s left-right decreasingness.

In [GM87], this transformation is restricted to a limited class of convergent CTRSs. However, in the following we show that for our purpose this restriction is not necessary. In other words, termination of $\mathcal{R}^{\mathrm{tr}}$ indeed implies left-right decreasingness (and thus also termination) of $\mathcal{R}$. Thus, this transformation is a generally applicable technique to reduce the termination problem of CTRSs to a termination problem of unconditional TRSs. (A similar approach was presented in [Mar96] for decreasingness proofs (instead of *left-right* decreasingness) by using a transformation where all conditions of a rule have to be checked in parallel.) We first prove that any reduction with $\mathcal{R}$ can be simulated by $\mathcal{R}^{\mathrm{tr}}$.

Lemma 1. *Let q, q' be terms without* if*'s. If $q \to_{\mathcal{R}}^{+} q'$, then $q \to_{\mathcal{R}^{\mathrm{tr}}}^{+} q'$.*

Proof. There must be a $j \in \mathbb{N}$ such that $q \to^+_{\mathcal{R}_j} q$ (j is the *depth* of the reduction). We prove the theorem by induction on the depth and the length of the reduction $q \to^+_{\mathcal{R}} q$ (i.e., we use a lexicographic induction relation).

The reduction has the form $q \to_{\mathcal{R}} p \to_{\mathcal{R}} q$ and by the induction hypothesis we know $p \to_{\mathcal{R}^{\mathrm{tr}}} q$. Thus, it suffices to prove $q \to^+_{\mathcal{R}^{\mathrm{tr}}} p$.

If the reduction $q \to_{\mathcal{R}} p$ is done with an unconditional rule of $\mathcal{R}$, then the conjecture is trivial. Otherwise, we must have $q = C[l\sigma]$, $p = C[r\sigma]$ for some context C and some rule like (3). As the depth of the reductions $s_i\sigma \to_{\mathcal{R}} t_i\sigma$ is less than the depth of the reduction $q \to^+_{\mathcal{R}} q$, by the induction hypothesis we have $s_i\sigma \to_{\mathcal{R}^{\mathrm{tr}}} t_i\sigma$. This implies $q \to^+_{\mathcal{R}^{\mathrm{tr}}} p$. □

Now the desired result is a direct consequence of Lemma 1.

Corollary 1 (Left-Right Decreasing of $\mathcal{R}$ by Termination of $\mathcal{R}^{\mathrm{tr}}$). *If $\mathcal{R}^{\mathrm{tr}}$ is terminating, then $\mathcal{R}$ is left-right decreasing (and thus, it is also terminating).*

Proof. If $\to_{\mathcal{R}^{\mathrm{tr}}}$ is well founded, then $\to_{\mathcal{R}^{\mathrm{tr}}} \cup \rhd$ and hence, the transitive closure $(\to_{\mathcal{R}^{\mathrm{tr}}} \cup \rhd)^+$ are well founded, too. By Lemma 1, this relation satisfies all conditions imposed on the relation $>$ in Def. 1. Hence, $\mathcal{R}$ is left-right decreasing. □

In our example, the conditional rule (2) is transformed into three additional unconditional rules. But apart from the if-root symbol of the right-hand side, the first of these rules is identical to (4). Thus, we obtain two overlapping rules in the transformed TRS which correspond to the overlapping conditional rules (1) and (2). However, in the CTRS this critical pair is *infeasible* [DOS88], i.e., the conditions of both rules exclude each other. Thus, our transformation of CTRSs into TRSs sometimes introduces unnecessary rules and overlap.

Therefore, whenever we construct a rule of the form $q \to \mathsf{if}_k(\boldsymbol{t})$ and there already exists a rule $q \to \mathsf{if}_n(\boldsymbol{t})$, then we identify if_k and if_n. This does not affect the soundness of our approach, because termination of a TRS where all occurrences of a symbol g are substituted by a symbol f with the same arity always implies termination of the original TRS.[1] Thus, we obtain the additional rules:

$$\mathsf{if}_1(\mathit{store}, m, \mathsf{false}) \to \mathsf{if}_3(\mathit{store}, m, \mathsf{empty}(\mathsf{fstsplit}(m, \mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), \mathit{store})))) \quad (7)$$

$$\mathsf{if}_3(\mathit{store}, m, \mathsf{false}) \to \mathsf{process}(\mathsf{sndsplit}(m, \mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), \mathit{store})), m) \quad (8)$$

If termination of a CTRS depends on its conditions, then in general termination of the transformed TRS can only be shown if one examines which terms may follow each other in a reduction. However, in the classical approaches based on simplification orderings (cf. e.g. [Der87,Ste95]), such considerations do not take place. Hence, they fail in proving the termination of (4)-(8). For this reason, such transformations into unconditional TRSs have rarely been applied for

[1] This possibility to eliminate unnecessary overlap is an advantage of our transformation compared to the one of [Mar96], where the transformed unconditional TRSs remain overlapping. In practice, proving termination of non-overlapping TRSs is significantly easier, since one may use techniques specifically tailored to *innermost* termination proofs, see below.

termination (or decreasingness) proofs of CTRSs. However, we will demonstrate that with the *dependency pair* approach this transformation is very useful.

To verify our original goal, we now have to prove termination of the transformed TRS which consists of (4)-(8), the rules for all auxiliary (library) functions from Sect. 2, and the (unknown) rules for the unspecified function f. Note that if an Erlang function is straightforwardly transformed into a TRS, then this TRS is non-overlapping. Thus, we assume that all possible rules for the unspecified function f are non-overlapping as well. Then it is sufficient just to prove *innermost* termination of the resulting TRS, cf. e.g. [Gra95]. In order to apply verification on a large scale, the aim is to perform such proofs automatically. Extending the dependency pair technique makes this possible.

4 Dependency Pairs

Dependency pairs allow the use of existing techniques like simplification orderings for automated termination and innermost termination proofs where they were not applicable before. In this section we briefly recapitulate the basic concepts of this approach and we present the theorems that we need for the rest of the paper. For further details and explanations see [AG97b,AG98,AG99].

In contrast to the standard approaches for termination proofs, which compare left and right-hand sides of rules, we only examine those subterms that are responsible for starting new reductions. For that purpose we concentrate on the subterms in the right-hand sides of rules that have a defined[2] root symbol, because these are the only terms a rewrite rule can ever be applied to.

More precisely, for every rule $f(s_1, \ldots, s_n) \to C[g(t_1, \ldots, t_m)]$ (where f and g are defined symbols), we compare the argument tuples $s_1, \ldots, s_n$ and $t_1, \ldots, t_m$. To avoid the handling of tuples, for every defined symbol f we introduce a fresh *tuple* symbol F. To ease readability, we assume that the original signature consists of lower case function symbols only, whereas the tuple symbols are denoted by the corresponding upper case symbols. Now instead of the tuples $s_1, \ldots, s_n$ and $t_1, \ldots, t_m$ we compare the *terms* $F(s_1, \ldots, s_n)$ and $G(t_1, \ldots, t_m)$.

Definition 2 (Dependency Pair). *If* $f(s_1, \ldots, s_n) \to C[g(t_1, \ldots, t_m)] \in \mathcal{R}$ *and* g *is defined, then* $\langle F(s_1, \ldots, s_n), G(t_1, \ldots, t_m)\rangle$ *is a* dependency pair *of* $\mathcal{R}$.

For the rules (4)-(8), (besides others) we obtain the following dependency pairs.

$$\langle \mathsf{PROCESS}(store, m), \mathsf{IF}_1(store, m, \mathsf{leq}(m, \mathsf{length}(store)))\rangle \quad (9)$$

$$\langle \mathsf{IF}_1(store, m, \mathsf{true}), \mathsf{IF}_2(store, m, \mathsf{empty}(\mathsf{fstsplit}(m, store)))\rangle \quad (10)$$

$$\langle \mathsf{IF}_2(store, m, \mathsf{false}), \mathsf{PROCESS}(\mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), \mathsf{sndsplit}(m, store)), m)\rangle \quad (11)$$

$$\langle \mathsf{IF}_1(store, m, \mathsf{false}), \mathsf{IF}_3(store, m, \mathsf{empty}(\mathsf{fstsplit}(m, \mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), store)))))\rangle \quad (12)$$

$$\langle \mathsf{IF}_3(store, m, \mathsf{false}), \mathsf{PROCESS}(\mathsf{sndsplit}(m, \mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), store)), m)\rangle \quad (13)$$

To trace newly introduced redexes in an innermost reduction, we consider special sequences of dependency pairs, so-called *innermost chains*.

[2] Root symbols of left-hand sides are *defined* and all other functions are *constructors*.

Definition 3 (Innermost $\mathcal{R}$-chains). *Let $\mathcal{R}$ be a TRS. A sequence of dependency pairs $\langle s_1, t_1\rangle \langle s_2, t_2\rangle \ldots$ is called an* innermost $\mathcal{R}$-chain *if there exists a substitution σ, such that all $s_j\sigma$ are in normal form and $t_j\sigma \xrightarrow{i}_{\mathcal{R}} s_{j+1}\sigma$ holds for every two consecutive pairs $\langle s_j, t_j\rangle$ and $\langle s_{j+1}, t_{j+1}\rangle$ in the sequence.*

We always assume that different (occurrences of) dependency pairs have disjoint variables and we always regard substitutions whose domains may be infinite. In [AG97b] we showed that the absence of infinite innermost chains is a (sufficient and necessary) criterion for innermost termination. To improve this criterion we introduced the following graph which contains arcs between all those dependency pairs which may follow each other in innermost chains.

Definition 4 (Estimated Innermost Dependency Graph). *Let* $\mathrm{CAP}(t)$ *result from t by replacing all subterms with defined root symbols by different fresh variables. The* estimated innermost dependency graph *is the directed graph whose nodes are the dependency pairs and there is an arc from $\langle s, t\rangle$ to $\langle v, w\rangle$ iff* $\mathrm{CAP}(t)$ *and v are unifiable by a mgu μ where $s\mu$ and $v\mu$ are normal forms. A non-empty set $\mathcal{P}$ of dependency pairs is called a* cycle *iff for all $\langle s, t\rangle, \langle v, w\rangle \in \mathcal{P}$, there is a path from $\langle s, t\rangle$ to $\langle v, w\rangle$ in this graph, which only traverses pairs from $\mathcal{P}$.*

In our example, (besides others) there are arcs from (9) to (10) and (12), from (10) to (11), from (12) to (13), and from both (11) and (13) to (9). Thus, the dependency pairs (9)-(13) form the cycles $\mathcal{P}_1 = \{(9), (10), (11)\}$, $\mathcal{P}_2 = \{(9), (12), (13)\}$, and $\mathcal{P}_3 = \{(9), (10), (11), (12), (13)\}$. However, (9)-(13) are not on a cycle with any *other* dependency pair (e.g., dependency pairs from the rules of the auxiliary library functions or the unspecified function f, since we assume that f does not call $\mathsf{process}$). This leads to the following refined criterion.

Theorem 1 (Innermost Termination Criterion). *A finite TRS $\mathcal{R}$ is innermost terminating iff for each cycle $\mathcal{P}$ in the estimated innermost dependency graph there exists no infinite innermost $\mathcal{R}$-chain of dependency pairs from $\mathcal{P}$.*

Note that in our definition, a cycle is a *set* of dependency pairs. Thus, for a finite TRS there only exist finitely many cycles $\mathcal{P}$. The automation of the technique is based on the generation of inequalities. For every cycle $\mathcal{P}$ we search for a well-founded quasi-ordering $\geq_{\mathcal{P}}$ satisfying $s \geq_{\mathcal{P}} t$ for all dependency pairs $\langle s, t\rangle$ in $\mathcal{P}$. Moreover, for at least one $\langle s, t\rangle$ in $\mathcal{P}$ we demand $s >_{\mathcal{P}} t$. In addition, to ensure $t\sigma \geq_{\mathcal{P}} v\sigma$ whenever $t\sigma$ reduces to $v\sigma$ (for consecutive pairs $\langle s, t\rangle$ and $\langle v, w\rangle$), we have to demand $l \geq_{\mathcal{P}} r$ for all those rules $l \to r$ of the TRS that may be used in this reduction. As we restrict ourselves to *normal* substitutions σ, not all rules are usable in a reduction of $t\sigma$. In general, if t contains a defined symbol f, then all f-rules are *usable* and moreover, all rules that are *usable* for right-hand sides of f-rules are also *usable* for t. Now we obtain the following theorem for automated[3] innermost termination proofs.

Theorem 2 (Innermost Termination Proofs). *A finite TRS is innermost terminating if for each cycle $\mathcal{P}$ there is a well-founded weakly monotonic quasi-ordering $\geq_{\mathcal{P}}$ where both $\geq_{\mathcal{P}}$ and $>_{\mathcal{P}}$ are closed under substitution, such that*

[3] Additional refinements for the automation can be found in [AG97b,AG99].

- $l \geq_{\mathcal{P}} r$ *for all rules* $l \to r$ *that are usable for some* t *with* $\langle s, t\rangle \in \mathcal{P}$,
- $s \geq_{\mathcal{P}} t$ *for all dependency pairs* $\langle s, t\rangle$ *from* $\mathcal{P}$, *and*
- $s >_{\mathcal{P}} t$ *for at least one dependency pair* $\langle s, t\rangle$ *from* $\mathcal{P}$.

Note that for Thm. 1 and 2 it is crucial to consider *all* cycles $\mathcal{P}$, not just the minimal ones (which contain no other cycles as proper subsets).

In Sect. 2 we presented the rules for the auxiliary functions in our example. Proving absence of infinite innermost chains for the cycles of their dependency pairs is very straightforward using Thm. 2. (So all library functions of our TRS are innermost terminating.) Moreover, as we assumed f to be a terminating function, its cycles do not lead to infinite innermost chains either.

Recall that (9)-(13) are not on cycles together with the remaining dependency pairs. Thus, what is left for verifying the desired property is proving absence of infinite innermost chains for the cycles $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$, where all rules of the whole TRS are possible candidates for being usable rules (also the rules for the unspecified function f).

Thm. 2 demands $s \geq_{\mathcal{P}} t$ resp. $s >_{\mathcal{P}} t$ for dependency pairs $\langle s, t\rangle$ on cycles. However for (9)-(13), these inequalities are not satisfied by any quasi-simplification ordering.[4] Thus, the automated proof fails here. Moreover, it is unclear which inequalities we have to add for the usable rules, since the rules for f are not given. Therefore, we have to extend the dependency pair technique.

5 Narrowing Dependency Pairs

To prove the absence of infinite innermost chains, for a dependency pair $\langle v, w\rangle$ it would be sufficient to demand $v\sigma \geq_{\mathcal{P}} w\sigma$ resp. $v\sigma >_{\mathcal{P}} w\sigma$ just for those instantiations σ where an instantiated right component $t\sigma$ of a previous dependency pair $\langle s, t\rangle$ reduces to $v\sigma$. For example, (11) only has to be regarded for instantiations σ where the instantiated right component $\mathsf{IF}_2(\mathit{store}, m, \mathsf{empty}(\mathsf{fstsplit}(m, \mathit{store})))\sigma$ of (10) reduces to the instantiated left component $\mathsf{IF}_2(\mathit{store}, m, \mathsf{false})\sigma$ of (11). In fact, this can only happen if *store* is not empty, i.e., if *store* reduces to the form $\mathsf{cons}(h, t)$. However, this observation has not been used in the inequalities of Thm. 2 and hence, we could not find an ordering for them. Thus, the idea is to perform the computation of empty on the level of the dependency pair. For that purpose the well-known concept of *narrowing* is extended to pairs of terms.

Definition 5 (Narrowing Pairs). *If a term* t *narrows to a term* t' *via the substitution* μ, *then the pair of terms* $\langle s, t\rangle$ *narrows to the pair* $\langle s\mu, t'\rangle$.

For example, the narrowings of the dependency pair (10) are

$$\langle \mathsf{IF}_1(x, 0, \mathsf{true}), \mathsf{IF}_2(x, 0, \mathsf{empty}(\mathsf{nil}))\rangle \tag{10a}$$
$$\langle \mathsf{IF}_1(\mathsf{nil}, \mathsf{s}(n), \mathsf{true}), \mathsf{IF}_2(\mathsf{nil}, \mathsf{s}(n), \mathsf{empty}(\mathsf{nil}))\rangle \tag{10b}$$
$$\langle \mathsf{IF}_1(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{true}), \mathsf{IF}_2(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{empty}(\mathsf{cons}(h, \mathsf{fstsplit}(n, t))))\rangle. \tag{10c}$$

[4] Essentially, the reason is that the left-hand side of dependency pair (9) is embedded in the right-hand sides of the pairs (11) and (13).

Thus, if a dependency pair $\langle s, t\rangle$ is followed by some dependency pairs $\langle v, w\rangle$ in an innermost chain and if t is not already unifiable with v (i.e., at least one rule is needed to reduce $t\sigma$ to $v\sigma$), then in order to 'approximate' the possible reductions of $t\sigma$ we may replace $\langle s, t\rangle$ by *all* its narrowings. Hence, we can replace the dependency pair (10) by the new pairs (10a)-(10c).

This enables us to extract necessary information from the last arguments of if's, i.e., from the former conditions of the CTRS. Thus, the narrowing refinement is the main reason why the transformation of CTRSs into TRSs is useful when analyzing the termination behaviour with dependency pairs. The number of narrowings for a pair is finite (up to variable renaming) and it can easily be computed automatically. The soundness of this technique is proved in [AG99].

Theorem 3 (Narrowing Refinement). *Let $\mathcal{P}$ be a set of pairs of terms and let $\langle s, t\rangle \in \mathcal{P}$ such that $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ and such that for all (renamings of) $\langle v, w\rangle \in \mathcal{P}$, the terms t and v are not unifiable. Let $\mathcal{P}'$ result from $\mathcal{P}$ by replacing $\langle s, t\rangle$ by all its narrowings. If there exists no infinite innermost chain of pairs from $\mathcal{P}'$, then there exists no infinite innermost chain of pairs from $\mathcal{P}$ either.*

So we may always replace a dependency pair by all its narrowings. However, while this refinement is sound, in general it destroys the necessity of our innermost termination criterion in Thm. 1. For example, the TRS with the rules $\mathsf{f}(\mathsf{s}(x)) \to \mathsf{f}(\mathsf{g}(\mathsf{h}(x)))$, $\mathsf{g}(\mathsf{h}(x)) \to \mathsf{g}(x)$, $\mathsf{g}(\mathsf{0}) \to \mathsf{s}(\mathsf{0})$, $\mathsf{h}(\mathsf{0}) \to \mathsf{1}$ is innermost terminating. But if the dependency pair $\langle \mathsf{F}(\mathsf{s}(x)), \mathsf{F}(\mathsf{g}(\mathsf{h}(x)))\rangle$ is replaced by its narrowings $\langle \mathsf{F}(\mathsf{s}(\mathsf{0})), \mathsf{F}(\mathsf{g}(\mathsf{1}))\rangle$ and $\langle \mathsf{F}(\mathsf{s}(x)), \mathsf{F}(\mathsf{g}(x))\rangle$, then $\langle \mathsf{F}(\mathsf{s}(x)), \mathsf{F}(\mathsf{g}(x))\rangle$ forms an infinite innermost chain (using the instantiation $\{x/\mathsf{0}\}$).

Nevertheless, in the application domain of process verification, we can restrict ourselves to *non-overlapping* TRSs. The following theorem shows that for these TRSs, narrowing dependency pairs indeed is a completeness preserving technique. More precisely, whenever innermost termination can be proved with the pairs $\mathcal{P}$, then it can also be proved with the pairs $\mathcal{P}'$.

Theorem 4 (Narrowing Dependency Pairs Preserves Completeness). *Let $\mathcal{R}$ be an innermost terminating non-overlapping TRS and let $\mathcal{P}$, $\mathcal{P}'$ be as in Thm. 3. If there exists no infinite innermost $\mathcal{R}$-chain of pairs from $\mathcal{P}$, then there exists no infinite innermost $\mathcal{R}$-chain of pairs from $\mathcal{P}'$ either.*

Proof. We show that every innermost $\mathcal{R}$-chain $\ldots \langle v_1, w_1\rangle \langle s', t'\rangle \langle v_2, w_2\rangle \ldots$ from $\mathcal{P}'$ can be transformed into an innermost chain from $\mathcal{P}$ of same length. There must be a substitution σ such that for all pairs the instantiated left-hand side is a normal form and the instantiated right-hand side reduces to the instantiated left-hand side of the next pair in the innermost chain. So in particular we have

$$w_1\sigma \xrightarrow{\mathsf{i}}_{\mathcal{R}} s'\sigma \quad \text{and} \quad t'\sigma \xrightarrow{\mathsf{i}}_{\mathcal{R}} v_2\sigma.$$

We know that $\langle s, t\rangle$ narrows to $\langle s', t'\rangle$ via a substitution μ. As the variables in $\langle s, t\rangle$ are disjoint from all other variables, we may extend σ to 'behave' like $\mu\sigma$ on the variables of s and t. Then we have $s\sigma = s\mu\sigma = s'\sigma$ and hence, $w_1\sigma \xrightarrow{\mathsf{i}}_{\mathcal{R}} s\sigma$.

Moreover, by the definition of narrowing, $t\mu \to_{\mathcal{R}} t$. This implies $t\mu\sigma \to_{\mathcal{R}} t\ \sigma$ and as $t\sigma = t\mu\sigma$, we have $t\sigma \to_{\mathcal{R}} t\ \sigma \stackrel{i}{\to}_{\mathcal{R}} v_2\sigma$ where $v_2\sigma$ is a normal form. As $\mathcal{R}$ is innermost terminating and non-overlapping, it is convergent. Thus, every term has a unique normal form and hence, repeated application of *innermost* reduction steps to $t\sigma$ also yields the normal form $v_2\sigma$, i.e., $t\sigma \stackrel{i}{\to}_{\mathcal{R}} v_2\sigma$.
Thus, $\ldots \langle v_1, w_1\rangle \langle s, t\rangle \langle v_2, w_2\rangle \ldots$ is also an innermost $\mathcal{R}$-chain. □

Hence, *independent* of the technique used to check the absence of infinite innermost chains, narrowing dependency pairs can never destroy the success of the innermost termination proof. Moreover, narrowing can of course be repeated an *arbitrary* number of times. Thus, after replacing (10) by (10a)-(10c), we may subsequently replace (10a) and (10b) by their respective narrowings.

$$\langle \mathsf{IF}_1(x, \mathsf{0}, \mathsf{true}), \mathsf{IF}_2(x, \mathsf{0}, \mathsf{true})\rangle \tag{10aa}$$
$$\langle \mathsf{IF}_1(\mathsf{nil}, \mathsf{s}(n), \mathsf{true}), \mathsf{IF}_2(\mathsf{nil}, \mathsf{s}(n), \mathsf{true})\rangle \tag{10ba}$$

This excludes them from being on a cycle in the estimated innermost dependency graph. Thus, now instead of the dependency pairs (9)-(13) we consider (9), (10c), (11), (12), and (13). A further narrowing of (10c) is not necessary for our purposes (but according to Thm. 4 it would not harm either). The right component of the dependency pair (11) unifies with the left component of (9) and therefore, (11) must not be narrowed. Instead we narrow (9).

$$\langle \mathsf{PROCESS}(\mathsf{nil}, m), \mathsf{IF}_1(\mathsf{nil}, m, \mathsf{leq}(m, \mathsf{0}))\rangle \tag{9a}$$
$$\langle \mathsf{PROCESS}(\mathsf{cons}(h, t), m), \mathsf{IF}_1(\mathsf{cons}(h, t), m, \mathsf{leq}(m, \mathsf{s}(\mathsf{length}(t))))\rangle \tag{9b}$$
$$\langle \mathsf{PROCESS}(\mathit{store}, \mathsf{0}), \mathsf{IF}_1(\mathit{store}, \mathsf{0}, \mathsf{true})\rangle \tag{9c}$$

By narrowing (10) to (10c), we determined that we only have to regard instantiations where *store* has the form $\mathsf{cons}(h, t)$ and m has the form $\mathsf{s}(n)$. Thus, (9a) and (9c) do not occur on a cycle and therefore, (9) can be replaced by (9b) only.

As (11)'s right component does not unify with left components any longer, we may now narrow (11) as well. By repeated narrowing steps and by dropping those pairs which do not occur on cycles, (11) can be replaced by

$$\langle \mathsf{IF}_2(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{false}), \mathsf{PROCESS}(\mathsf{sndsplit}(n, t), \mathsf{s}(n))\rangle \tag{11aac}$$
$$\langle \mathsf{IF}_2(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{false}), \mathsf{PROCESS}(\mathsf{app}(\mathsf{nil}, \mathsf{sndsplit}(n, t)), \mathsf{s}(n))\rangle \tag{11ad}$$
$$\langle \mathsf{IF}_2(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{false}), \mathsf{PROCESS}(\mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), \mathsf{sndsplit}(n, t)), \mathsf{s}(n))\rangle \tag{11d}$$

Now for the cycle $\mathcal{P}_1$, it is (for example) sufficient to demand that (11aac), (11ad), and (11d) are strictly decreasing and that (9b), (10c), and all usable rules are weakly decreasing. Similar narrowings can also be applied for the pairs (12) and (13) which results in analogous inequalities for the cycles $\mathcal{P}_2$ and $\mathcal{P}_3$.

Most standard orderings amenable to automation are *strongly* monotonic path orderings (cf. e.g. [Der87,Ste95]), whereas here we only need *weak* monotonicity. Hence, before synthesizing a suitable ordering, some of the arguments of function symbols may be eliminated, cf. [AG99]. For example, in our inequalities one may eliminate the third argument of IF_2. Then every term $\mathsf{IF}_2(t_1, t_2, t_3)$ in the inequalities is replaced by $\mathsf{IF}_2(t_1, t_2)$ (where IF_2 is a new binary function

symbol). By comparing the terms resulting from this replacement instead of the original terms, we can take advantage of the fact that IF_2 does not have to be strongly monotonic in its third argument. Similarly, in our example we will also eliminate the third arguments of IF_1 and IF_3 and the first argument of sndsplit. Note that there are only finitely many (and only few) possibilities to eliminate arguments of function symbols. Therefore all these possibilities can be checked automatically. In this way, the recursive path ordering (rpo) satisfies the inequalities for (11aac), (9b), (10c), for the dependency pairs resulting from (12) and (13), and for all (known) usable rules. However, the inequalities resulting from (11ad) and (11d)

$$\mathsf{IF}_2(\mathsf{cons}(h,t),\mathsf{s}(n)) > \mathsf{PROCESS}(\mathsf{app}(\mathsf{nil},\mathsf{sndsplit}\ (t)),\mathsf{s}(n))$$
$$\mathsf{IF}_2(\mathsf{cons}(h,t),\mathsf{s}(n)) > \mathsf{PROCESS}(\mathsf{app}(\mathsf{map\_f}(\mathsf{self},\mathsf{nil}),\mathsf{sndsplit}\ (t)),\mathsf{s}(n))$$

are not satisfied because of the app-terms on the right-hand sides (as the app-rule forces app to be greater than cons in the precedence of the rpo). Moreover, the map_f-term in the inequalities requires us to consider the usable rules corresponding to the (unspecified) Erlang function f as well.

To get rid of these terms, one would like to perform narrowing on map_f and app. However, in general narrowing only *some* subterms of right components is unsound.[5] Instead, we always have to replace a pair by *all* its narrowings. But then narrowing (11ad) and (11d) provides no solution here, since narrowing the sndsplit-subterm results in pairs containing problematic app- and map_f-terms again. In the next section we describe a technique which solves the above problem.

6 Rewriting Dependency Pairs

While performing only some *narrowing* steps is unsound, for non-overlapping TRSs it is at least sound to perform only one of the possible *rewrite* steps.[6] So if $t \to r$, then we may replace a dependency pair $\langle s,t\rangle$ by $\langle s,r\rangle$. Note that this technique is only applicable to *dependency pairs*, but not to *rules* of the TRS. Indeed, by reducing the right-hand side of a rule, a non (innermost) terminating TRS can be transformed into a terminating one, even if the TRS is non-overlapping. As an example regard the TRS with the rules $\mathsf{0} \to \mathsf{f}(\mathsf{0})$, $\mathsf{f}(x) \to \mathsf{1}$ which is clearly not innermost terminating. However, if the right-hand side of the first rule is rewritten to 1, then the resulting TRS is terminating. The following theorem proves that our refinement of the dependency pair approach is sound.

[5] As an example regard the TRS $\mathsf{f}(\mathsf{0},\mathsf{1}) \to \mathsf{s}(\mathsf{1})$, $\mathsf{f}(x,\mathsf{0}) \to \mathsf{1}$, $\mathsf{a} \to \mathsf{0}$, and $\mathsf{g}(\mathsf{s}(y)) \to \mathsf{g}(\mathsf{f}(\mathsf{a},y))$. If we would replace the dependency pair $\langle \mathsf{G}(\mathsf{s}(y)), \mathsf{G}(\mathsf{f}(\mathsf{a},y))\rangle$ by only one of its narrowings, viz. $\langle \mathsf{G}(\mathsf{s}(\mathsf{0})), \mathsf{G}(\mathsf{1})\rangle$, then one could falsely prove innermost termination, although the term $\mathsf{g}(\mathsf{s}(\mathsf{1}))$ starts an infinite innermost reduction.

[6] Combining narrowing and rewriting is common in *normal narrowing* strategies to solve E-unification problems [Fay79,Han94]. However, in contrast to our approach, *normal narrowing* is only used for convergent TRSs and instead of performing one (or arbitrary) many rewrite steps, there one rewrites terms to normal forms.

Theorem 5 (Rewriting Dependency Pairs). *Let $\mathcal{R}$ be non-overlapping and let $\mathcal{P}$ be a set of pairs of terms. Let $\langle s,t\rangle \in \mathcal{P}$, let $t \to_{\mathcal{R}} r$ and let $\mathcal{P}$ result from $\mathcal{P}$ by replacing $\langle s,t\rangle$ by $\langle s,r\rangle$. If there exists no infinite innermost chain of pairs from $\mathcal{P}$, then there exists no infinite innermost chain from $\mathcal{P}$ either.*

Proof. By replacing all (renamed) occurrences of $\langle s,t\rangle$ with the corresponding renamed occurrences of $\langle s,r\rangle$, every innermost chain $\ldots \langle s,t\rangle \langle v,w\rangle \ldots$ from $\mathcal{P}$ can be translated into an innermost chain from $\mathcal{P}$ of same length. The reason is that there must be a substitution σ with $t\sigma \stackrel{i}{\to}_{\mathcal{R}} v\sigma$ where $v\sigma$ is a normal form. So $t\sigma$ is weakly innermost normalizing and thus, by [Gra96a, Thm. 3.2.11 (1a) and (4a)], $t\sigma$ is confluent and strongly normalizing. With $t \to_{\mathcal{R}} r$, we obtain $t\sigma \to_{\mathcal{R}} r\sigma$. Hence, $r\sigma$ is strongly normalizing as well and thus, it also reduces innermost to some normal form q. Now confluence of $t\sigma$ implies $q = v\sigma$. Therefore, $\ldots \langle s,r\rangle \langle v,w\rangle \ldots$ is an innermost chain, too. □

The converse of Thm. 5 holds as well if $\mathcal{P}$ is obtained from the dependency pairs by repeated narrowing and rewriting steps. So similar to *narrowing*, *rewriting* dependency pairs does not destroy the necessity of our criterion either.

Theorem 6 (Rewriting Dependency Pairs Preserves Completeness). *Let $\mathcal{R}$ be an innermost terminating non-overlapping TRS and let $\mathcal{P}$, $\mathcal{P}$ be as in Thm. 5. If there exists no infinite innermost $\mathcal{R}$-chain of pairs from $\mathcal{P}$, then there exists no infinite innermost $\mathcal{R}$-chain of pairs from $\mathcal{P}$ either.*

Proof. In an innermost chain $\ldots \langle s,r\rangle \langle v,w\rangle \ldots$ from $\mathcal{P}$, replacing all (renamed) occurrences of $\langle s,r\rangle$ by corresponding renamings of $\langle s,t\rangle$ yields an innermost chain from $\mathcal{P}$ of same length. The reason is that there must be a σ with $r\sigma \stackrel{i}{\to}_{\mathcal{R}} v\sigma$. Thus, $t\sigma \to_{\mathcal{R}} r\sigma \stackrel{i}{\to}_{\mathcal{R}} v\sigma$ implies $t\sigma \stackrel{i}{\to}_{\mathcal{R}} v\sigma$ by the convergence of $\mathcal{R}$. □

In our example we may now eliminate app and map_f by rewriting the pairs (11ad) and (11d). Even better, before narrowing, we could first rewrite (11), (12), and (13). Moreover, we could simplify (10c) by rewriting it as well. Thus, the resulting pairs on the cycles we are interested in are:

$$\langle \mathsf{PROCESS}(\mathsf{cons}(h,t),m), \mathsf{IF}_1(\mathsf{cons}(h,t),m,\mathsf{leq}(m,\mathsf{s}(\mathsf{length}(t))))\rangle \quad (9b)$$
$$\langle \mathsf{IF}_1(\mathsf{cons}(h,t),\mathsf{s}(n),\mathsf{true}), \mathsf{IF}_2(\mathsf{cons}(h,t),\mathsf{s}(n),\mathsf{false})\rangle \quad (10c)$$
$$\langle \mathsf{IF}_2(\mathit{store},m,\mathsf{false}), \mathsf{PROCESS}(\mathsf{sndsplit}(m,\mathit{store}),m)\rangle \quad (11)$$
$$\langle \mathsf{IF}_1(\mathit{store},m,\mathsf{false}), \mathsf{IF}_3(\mathit{store},m,\mathsf{empty}(\mathsf{fstsplit}(m,\mathit{store})))\rangle \quad (12)$$
$$\langle \mathsf{IF}_3(\mathit{store},m,\mathsf{false}), \mathsf{PROCESS}(\mathsf{sndsplit}(m,\mathit{store}),m)\rangle \quad (13)$$

Analogous to Sect. 5, now we narrow (11), (12), (13), perform a rewrite step for one of (12)'s narrowings, and delete those resulting pairs which are not on any cycle. In this way, (11), (12), (13) are replaced by

$$\langle \mathsf{IF}_2(\mathsf{cons}(h,t),\mathsf{s}(n),\mathsf{false}), \mathsf{PROCESS}(\mathsf{sndsplit}(n,t),\mathsf{s}(n))\rangle \quad (11)$$
$$\langle \mathsf{IF}_1(\mathsf{cons}(h,t),\mathsf{s}(n),\mathsf{false}), \mathsf{IF}_3(\mathsf{cons}(h,t),\mathsf{s}(n),\mathsf{false})\rangle \quad (12)$$
$$\langle \mathsf{IF}_3(\mathsf{cons}(h,t),\mathsf{s}(n),\mathsf{false}), \mathsf{PROCESS}(\mathsf{sndsplit}(n,t),\mathsf{s}(n))\rangle \quad (13)$$

By eliminating the first argument of $\mathsf{sndsplit}$ and the third arguments of IF_1, IF_2, and IF_3 (cf. Sect. 5), we obtain the following inequalities. Note that according to Thm. 2, these inequalities prove the absence of infinite innermost chains for all three cycles built from (9b), (10c), and (11)-(13), since for each of these cycles (at least) one of its dependency pairs is strictly decreasing.

$$\begin{array}{ll}
\mathsf{PROCESS}(\mathsf{cons}(h,t),m) \geq \mathsf{IF}_1(\mathsf{cons}(h,t),m) & \mathsf{sndsplit}\ (x) \geq x \\
\mathsf{IF}_1(\mathsf{cons}(h,t),\mathsf{s}(n)) \geq \mathsf{IF}_2(\mathsf{cons}(h,t),\mathsf{s}(n)) & \mathsf{sndsplit}\ (\mathsf{nil}) \geq \mathsf{nil} \\
\mathsf{IF}_1(\mathsf{cons}(h,t),\mathsf{s}(n)) \geq \mathsf{IF}_3(\mathsf{cons}(h,t),\mathsf{s}(n)) & \mathsf{sndsplit}'(\mathsf{cons}(h,t)) \geq \mathsf{sndsplit}'(t) \\
\mathsf{IF}_2(\mathsf{cons}(h,t),\mathsf{s}(n)) > \mathsf{PROCESS}(\mathsf{sndsplit}'(t),\mathsf{s}(n)) & l \geq r \text{ for all rules } l \to r \\
\mathsf{IF}_3(\mathsf{cons}(h,t),\mathsf{s}(n)) > \mathsf{PROCESS}(\mathsf{sndsplit}'(t),\mathsf{s}(n)) & \text{with } \mathit{root}(l) \in \{\mathsf{leq},\mathsf{length}\}
\end{array}$$

Now these inequalities are satisfied by the rpo. The right column contains all inequalities corresponding to the usable rules, since the rules for $\mathsf{map\_f}$ and f are no longer usable. Hence, the TRS of Sect. 3 is innermost terminating. In this way, left-right decreasingness of the CTRS from Sect. 2 could be proved automatically. Therefore, the desired property holds for the original Erlang process.

7 Conclusion

We have shown that rewriting techniques (and in particular, the dependency pair approach) can be successfully applied for process verification tasks in industry. While our work was motivated by a specific process verification problem, in this paper we developed several new techniques which are of general use in term rewriting. First of all, we showed how dependency pairs can be utilized to prove that *conditional* term rewriting systems are decreasing and terminating. Moreover, we presented two refinements which considerably increase the class of systems where dependency pairs are successful. The first refinement of *narrowing* dependency pairs was already introduced in [AG99], but completeness of the technique for non-overlapping TRSs is a new result. It ensures that application of the narrowing technique can never destroy the success of such an innermost termination proof. In fact, our narrowing refinement is the main reason why the approach of handling CTRSs by transforming them into TRSs is successful in combination with the dependency pair approach (whereas this transformation is usually not of much use for the standard termination proving techniques). Finally, to strengthen the power of dependency pairs we introduced the novel technique of *rewriting* dependency pairs and proved its soundness and completeness for innermost termination of non-overlapping TRSs.

Acknowledgements. We thank the anonymous referees for their helpful comments.

References

[AD99] T. Arts & M. Dam, Verifying a distributed database lookup manager written in Erlang. In *Proc. FM '99*, Toulouse, France, 1999.

[AG97a] T. Arts & J. Giesl, Automatically proving termination where simplification orderings fail. *TAPSOFT '97*, LNCS 1214, pp. 261–273, Lille, France, 1997.

[AG97b] T. Arts & J. Giesl, Proving innermost normalisation automatically. In *Proc. RTA-97*, LNCS 1232, pp. 157–172, Sitges, Spain, 1997.

[AG98] T. Arts & J. Giesl, Modularity of termination using dependency pairs. In *Proc. RTA-98*, LNCS 1232, pp. 226–240, Tsukuba, Japan, 1998.

[AG99] T. Arts & J. Giesl, Termination of term rewriting using dependency pairs. *TCS*. To appear. Preliminary version under http://www.inferenzsysteme.informatik.tu-darmstadt.de/~reports/notes/ibn-97-46.ps

[BN98] F. Baader & T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1998.

[BK86] J. A. Bergstra & J. W. Klop, Conditional rewrite rules: confluence and termination. *JCSS*, 32:323–362, 1986.

[BG89] H. Bertling & H. Ganzinger, Completion-time optimization of rewrite-time goal solving. *Proc. RTA-89*, LNCS 355, pp. 45–58, Chapel Hill, USA, 1989.

[DP87] N. Dershowitz & D. A. Plaisted, Equational programming. *Machine Intelligence*, 11:21–56, Oxford University Press, 1987.

[Der87] N. Dershowitz, Termination of rewriting. *JSC*, 3:69–116, 1987.

[DOS88] N. Dershowitz, M. Okada, & G. Sivakumar, Canonical conditional rewrite systems. In *Proc. CADE-9*, LNCS 310, pp. 538–549, Argonne, USA, 1988.

[DO90] N. Dershowitz & M. Okada, A rationale for conditional equational programming. *TCS*, 75:111–138, 1990.

[DJ90] N. Dershowitz & J.-P. Jouannaud, Rewrite Systems. In *Handbook of Theoretical Computer Science, Vol. B*, pp. 243-320, Elsevier, 1990.

[DH95] N. Dershowitz & C. Hoot, Natural termination. *TCS*, 142(2):179–207, 1995.

[Fay79] M. J. Fay, First-order unification in an equational theory. *Proc. 4th Workshop on Automated Deduction*, pp. 161-167, Austin, TX, Academic Press, 1979.

[GM87] E. Giovanetti & C. Moiso, Notes on the eliminations of conditions. In *Proc. CTRS '87*, LNCS 308, pp. 91–97, Orsay, France, 1987.

[Gra94] B. Gramlich, On termination and confluence of conditional rewrite systems. In *Proc. CTRS '94*, LNCS 968, pp. 166–185, Jerusalem, Israel, 1994.

[Gra95] B. Gramlich, Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundamenta Informaticae*, 24:3–23, 1995.

[Gra96a] B. Gramlich, *Termination and confluence properties of structured rewrite systems*. PhD Thesis, Universität Kaiserslautern, Germany, 1996.

[Gra96b] B. Gramlich, On termination and confluence properties of disjoint and constructor-sharing conditional rewrite systems. *TCS*, 165:97–131, 1996.

[Han94] M. Hanus, The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19,20:583-628, 1994.

[HN99] Patent pending, Ericsson Telecom AB, 1999.

[Kap84] S. Kaplan, Conditional rewrite rules. *TCS*, 33:175–193, 1984.

[Mar96] M. Marchiori, Unravelings and Ultra-properties, *Proc. ALP '96*, LNCS 1139, pp. 107-121, Aachen, Germany, 1996.

[Mid93] A. Middeldorp, Modular properties of conditional term rewriting systems. *Information and Computation*, 104:110–158, 1993.

[Ste95] J. Steinbach, Simplification orderings: history of results. *Fundamenta Informaticae*, 24:47–87, 1995.

[SMI95] T. Suzuki, A. Middeldorp, & T. Ida, Level-confluence of conditional rewrite systems with extra variables in right-hand sides. *Proc. RTA-95*, LNCS 914, pp. 179–193, Kaiserslautern, Germany, 1995.

[WG94] C.-P. Wirth & B. Gramlich, A constructor-based approach for positive/negative conditional equational specifications. *JSC*, 17:51–90, 1994.

Difference Decision Diagrams*

Jesper Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard

The IT University in Copenhagen
Glentevej 67, DK-2400 Copenhagen NV, Denmark
Tel./Fax: +45 38 11 64 33/+45 38 11 41 39
Email: {jm,jl,hra,henrik}@itu.dk

Abstract. This paper describes a new data structure, difference decision diagrams (DDDs), for representing a Boolean logic over inequalities of the form $x - y \leq c$ where the variables are integer or real-valued. We give algorithms for manipulating DDDs and for determining validity, satisfiability, and equivalence. DDDs enable an efficient verification of timed systems modeled as, for example, timed automata or timed Petri nets, since both the states and their associated timing information are represented *symbolically*, similar to how BDDs represent Boolean predicates. We demonstrate the efficiency of DDDs by analyzing a timed system and compare the results with the tools KRONOS and UPPAAL.

1 Introduction

Today model checking [13] is used extensively for formal verification of finite state systems such as digital circuits and embedded software. The success of the technique is primarily due to the use of BDDs [9] for representing sets of and relations over Boolean variables *symbolically*, making it possible to verify systems with a very large number of states. However, if the model contains non-Boolean (e.g., real-valued) variables, BDDs and other symbolic representations of Boolean predicates are inefficient. As a consequence, state-of-the-art techniques for analyzing systems with time, modeled for example as timed automata, are only capable of analyzing systems with a handful of timers and a few thousand states.

In this paper we consider a Boolean logic extended with difference constraints, i.e., inequalities of the form $x - y \leq c$, where x and y are integer or real-valued variables and c is a constant. Difference constraints arise naturally when analyzing systems with time, expressing relations between the timers in the model, e.g., that the difference between two timers is within some bound. We call the Boolean logic over difference constraints for *difference constraint expressions* given by the following grammar:

$$\phi ::= x - y \leq c \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \exists x.\phi\,, \tag{1}$$

where $x, y \in \mathbf{Var}$ denote variables and $c \in \mathbb{D}$ denotes a constant. We will allow the usual derived operators such as $x - y > c$, $\phi_1 \vee \phi_2$, and $\forall x.\phi$. In this paper, the domain $\mathbb{D}$ of the logic is either the real numbers $\mathbb{R}$ or the integers $\mathbb{Z}$.

* This work was carried out while the authors were at the Department of Information Technology, Technical University of Denmark, and was financially supported by a grant from the Danish Technical Research Council.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 111–125, 1999.

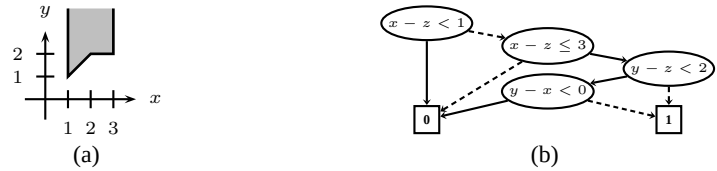


Fig. 1. The expression ϕ in (2) as (a) an (x, y)-plot for $z = 0$, and (b) a difference decision diagram.

The main contribution of this paper is a data structure, called *difference decision diagrams* (DDDs), for representing difference constraint expressions *symbolically*, making it possible to represent the state space of timed systems (and other systems with non-Boolean variables) efficiently. DDDs represent difference constraint expressions using a decision tree in a manner similar to the BDD representation of a Boolean expression. Consider the following expression ϕ over x, y, $z \in \mathbb{R}$:

$$\phi = 1 \leq x - z \leq 3 \ \wedge \ (y - z \geq 2 \ \vee \ y - x \geq 0)\,. \tag{2}$$

Figure 1 shows ϕ as an (x, y)-plot for $z = 0$ and the corresponding DDD. Each non-terminal vertex in a DDD contains a test expression α (a difference constraint) and has two outgoing edges called the high- and low-branch which are drawn with solid and dashed lines, respectively. The high-branch is followed when α evaluates to true; the low-branch when α evaluates to false.

1.1 Related Work

One approach to analyze systems with time or other continuous variables is to make the dense domains discrete. For example, in a timed model it is assumed that the clocks only can take integer or rational values. Such a discretization makes it possible to use BDDs for representing both the state graph and the associated timing information [2,8,10,11]. However, this way of representing dense domains is often inefficient; the BDD representation is very sensitive to the granularity of the discretization and to the size of the delay ranges. Another approach based on BDDs is to have a Boolean variable representing each constraint, and use an external decision procedure to determine implications among these variables [12]. These implications are used to prune the representation of the state space. The advantage is that any kind of decidable constraints can be used. Our approach can be seen as a simplified version of this where we take advantage of restricting the types of constraints to difference constraints and perform reductions on-the-fly.

Several algorithms for analyzing timed automata have been developed. The unit-cube approach [1] models time as dense but represents the timing information using a finite number of equivalence classes. Again, the number of timed states is dependent on the size of the delay ranges and easily becomes unmanageable. Several recent timing analysis methods use difference bound matrices (DBMs) [15] for representing the timing information [7,18,23,28]. In these approaches, a set of DBMs representing the possible timer configurations is associated with each discrete state of the system. Although DBMs provide a compact representation of a clock configuration, there are several serious problems with the approaches based on DBMs: first, the number of DBMs for representing

the timing information associated with a given state can become very large, secondly, there is no sharing or reuse of DBMs among the different discrete states, and finally, each discrete state is represented explicitly, thus these approaches are limited by the number of reachable states of the system. Several researchers have attempted to remedy these shortcomings, for example by using partial order methods [6,24,26] or by using approximate methods [3,5,27]. Although these approaches do address the first problem, they are still susceptible to the last two problems since each state is represented explicitly. Using DDDs it is possible to combat all three problems since first, unlike DBMs, DDDs are not limited to representing the timing information as a union of convex sets, secondly, DDDs represent all states and the associated timing information in a single shared data structure, and finally, states and the timing information are represented symbolically using difference constraint expressions. Another approach [25] suggests using a partition refinement algorithm for efficient model checking. However, the reported running times are still exponential.

Based on the initial ideas of this paper, Behrmann et al. [4] have implemented a minor variation of DDDs allowing a fanout of more than two (which they call CDDs). They have shown a significant improvement in memory consumption in UPPAAL, even though the experiments in contrast to ours do not use a fully symbolic approach (the discrete states are enumerated explicitly). Thus, this approach will not be able to handle the larger instances of the timed system in Sect. 6.

2 Difference Decision Diagrams

The data structure *difference decision diagrams* (DDDs) is developed to efficiently represent and manipulate difference constraint expressions. Difference decision diagrams share many properties with binary decision diagrams (BDDs): they can be ordered, they can be reduced making it possible to check for validity and satisfiability in constant time, and many of the algorithms and techniques for BDDs can be modified to apply to DDDs.

Definition 1 (Difference Decision Diagram). *A* difference decision diagram *(DDD) is a directed acyclic graph (V, E). The vertex set V contains two terminals **0** and **1** with out-degree zero, and a set of non-terminal vertices with out-degree two and the following attributes:*

Attribute	*Type*	*Description*
$pos(v), neg(v)$	$\mathbf{Var}$	*Positive variable x_i, and negative variable x_j.*
$op(v)$	$\{\text{LE}, \text{LEQ}\}$	*Operator $<$ or $\leq$.*
$const(v)$	$\mathbb{D}$	*Constant c.*
$high(v), low(v)$	V	*High-branch h, and low-branch l.*

The set E contains the edges $(v, low(v))$ and $(v, high(v))$, where $v \in V$ is a non-terminal vertex.

Similar to BDDs, the non-terminal vertices of a DDD corresponds to the if-then-else operator $\alpha \rightarrow \phi_1, \phi_0$ defined by

$$\alpha \rightarrow \phi_1, \phi_0 = (\alpha \wedge \phi_1) \vee (\neg\alpha \wedge \phi_0)\,,$$

where α is a test expression and ϕ_0, ϕ_1 are difference constraint expressions. However, unlike BDDs, the test expression α is not a Boolean variable, but a difference constraint of the form $x - y \lesssim c$, where the symbol $\lesssim$ represents either $<$ or $\leq$. Each vertex v in a DDD denotes a difference constraint expression ϕ^v. If v is a terminal vertex, i.e., either **0** or **1**, ϕ^v is false or true, respectively. Otherwise, v represents the expression ϕ^v given by:

$$\phi^v = x_i - x_j \lesssim c \rightarrow \phi^{high(v)}, \phi^{low(v)} ,$$

where $x_i = pos(v)$, $x_j = neg(v)$, $\lesssim = op(v)$, and $c = const(v)$. We use the following notational shorthands:

$$\begin{array}{ll} var(v) & = (pos(v), neg(v)) \\ bound(v) & = (const(v), op(v)) \\ cstr(v) & = (var(v), bound(v)) . \end{array}$$

Adding two bounds (c_1, o_1) and (c_2, o_2) gives $(c_1 + c_2, o_1 + o_2)$, where $o_1 + o_2$ is LEQ if both o_1 and o_2 are LEQ and LE otherwise. Negating a bound (c, o) gives $(-c, \neg o)$, where $\neg$LE is LEQ and $\neg$LEQ is LE. We use $v \rightsquigarrow u$ to denote that the vertex u is reachable from v (i.e., there is a path from v to u). The size of a DDD v, denoted $|v|$, is the number of vertices reachable from v; that is, $|v| = |\{u \in V : v \rightsquigarrow u\}|$.

2.1 Ordering

To define ordered DDDs, we assume given a total ordering $\prec$ of the variables $x_1, \ldots, x_n$ which furthermore must totally order pairs of variables (x_i, x_j).[1] We extend this ordering to attributes $cstr(v)$ of vertices v in a DDD. Constants, $const(v)$, are ordered as usually in $\mathbb{D}$, and the two operators, $op(v)$, are ordered as LE $\prec$ LEQ. Bounds, $(const(v), op(v))$ and constraints, $(var(v), bound(v))$, are ordered lexicographically. For example, $\big((x_2, x_1), (0, \text{LE})\big) \prec \big((x_2, x_1), (0, \text{LEQ})\big) \prec \big((x_2, x_1), (1, \text{LE})\big)$. We assume that the two terminal vertices have attributes that are greater than all non-terminals.

Definition 2 (Ordered DDD). *An ordered DDD (ODDD) is a DDD where each non-terminal vertex v satisfies:*

1. $neg(v) \prec pos(v)$,
2. $var(v) \prec var(high(v))$,
3. $var(v) \prec var(low(v))$ *or*
   $var(v) = var(low(v))$ *and* $bound(v) \prec bound(low(v))$.

Requirement 1 expresses that the pair of variables $var(v) = (pos(v), neg(v)) = (x_i, x_j)$ of a vertex v is *normalized*; that is, $x_j \prec x_i$. This does not restrict what we can represent with DDDs, because the two variables in a vertex can always be swapped by negating the bound and swapping the low- and high-branches. We further require in an ordered DDD,

[1] Pairs of variables can for example be ordered reversed lexicographically, that is $(x_i, x_j) \prec (x'_i, x'_j)$ iff $x_j \prec x'_j$ or $(x_j = x'_j \wedge x_i \prec x'_i)$.

that either the children of a vertex have variables later in the ordering (requirement 2 and first part of 3) or the variables along the low-branch are identical (second part of 3). The second part of requirement 3 makes it possible to have multiple tests on the same pair of variables, which is needed because of the disjunctive abilities of DDDs. The last two requirements imply $cstr(v) \prec cstr(high(v))$ and $cstr(v) \prec cstr(low(v))$. The DDD in Fig. 1 is an example of an ordered DDD with the ordering $z \prec x \prec y$ extended reversed lexicographically to pairs of variables.

2.2 Locally Reduced DDDs

Similar to ROBDDs, we define a set of local reduction rules that reduce the size of the DDD representation.

Definition 3 (Locally Reduced DDD). *A locally reduced DDD (R_LDDD) is an ODDD satisfying, for all non-terminals u and v:*

1. $\mathbb{D} = \mathbb{Z}$ *implies* $op(v) = \text{LEQ}$,
2. $(cstr(u), high(u), low(u)) = (cstr(v), high(v), low(v))$ *implies* $u = v$,
3. $low(v) \neq high(v)$,
4. $var(v) = var(low(v))$ *implies* $high(v) \neq high(low(v))$.

Requirement 2 and 3 are identical to the reduction requirements for ROBDDs. Thus, if we encode a Boolean variable b_i as $x_i - x_i \leq 0$, any Boolean expression over $b_1, b_2, \ldots, b_n$ is represented in a canonical form using locally reduced DDDs. Requirement 4 ensures that any two consecutive vertices with the same pair of variables have different high-branches. This requirement is fulfilled using the following equivalence for ordered DDDs:

$$x - y \lesssim_1 c_1 \rightarrow h, (x - y \lesssim_2 c_2 \rightarrow h, l) \;=\; x - y \lesssim_2 c_2 \rightarrow h, l\,.$$

3 Construction of DDDs

In this section we present efficient algorithms for manipulating locally reduced DDDs. For a more detailed description see [21]. Orderedness ensures that the basic algorithm for computing the Boolean connectives is polynomial. However, for existential quantification the situation is different. Although the algorithm in polynomial time computes the modified and additional constraints, its worst-case running time is exponential since it needs to regain orderedness.

The algorithms are all based on a function MK for creating DDD vertices. The function MK normalizes the two variables and ensures that the created vertex is locally reduced: If x is different from y, $\text{MK}\big((x, y), (c, o), h, l\big)$ returns the identity of a vertex, equivalent to a vertex v with $var(v) = (x, y)$, $bound(v) = (c, o)$, $high(v) = h$, and $low(v) = l$. If x is equal to y, MK returns **0** if the bound is less than $(0, \text{LEQ})$, and **1** otherwise. Using MK as the only function for constructing a DDD ensures that it is locally reduced. As for BDDs, MK can be implemented with an expected running time of $O(1)$.

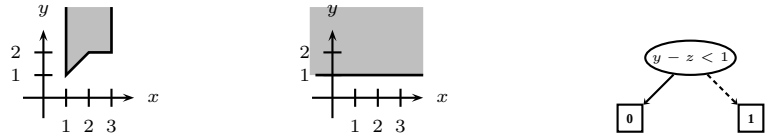


Fig. 2. Existential quantification of x in (2). (a) An (x, y)-plot of ϕ for $z = 0$. (b) An (x, y)-plot of $\exists x.\phi$ for $z = 0$. (c) The DDD for $\exists x.\phi$.

3.1 Boolean Combination of DDDs

The function $\textsc{Apply}(op, u, v)$ is used to combine two DDDs rooted at u and v with a Boolean operator op. APPLY is a generalization of the version used for ROBDDs, which is based on the fact that any binary Boolean operator op distributes over the if-then-else operator:

$$(\alpha \to h, l)\ op\ (\alpha\ \to h\ , l\ )\ =\ \alpha \to \big(h\ op\ (\alpha\ \to h\ , l\ )\big), \big(l\ op\ (\alpha\ \to h\ , l\ )\big)\,. \quad (3)$$

This equivalence provides a method to combine two DDDs with a Boolean operator. Reading the equivalence from left to right, we see that we can move the Boolean operator down one level in the DDD. If we continuously do so until both arguments of op are **0** or **1**, we can evaluate the expression and return the appropriate result.

If the two pairs of variables are equal, we can simplify (3):

$$(\alpha \to h, l)\ op\ (\alpha \to h\ , l\ ) = \begin{cases} \alpha\ \to (h\ op\ h\ ), (l\ op\ (\alpha \to h\ , l\ )) & \text{if } \alpha \prec \alpha\ , \\ \alpha\ \to (h\ op\ h\ ), (l\ op\ l\ ) & \text{if } \alpha = \alpha\ , \\ \alpha\ \to (h\ op\ h\ ), ((\alpha \to h, l)\ op\ l\ ) & \text{if } \alpha \succ \alpha\ . \end{cases} \quad (4)$$

Together, (3) and (4) yield the algorithm APPLY: We use (3) when $(x, y) \prec (x\ , y\ )$ or $(x, y) \succ (x\ , y\ )$ and (4) when $(x, y) = (x\ , y\ )$. Using MK to construct new vertices and applying dynamic programming, the runtime of APPLY is the same as the ROBDD version, that is, $O(|u||v|)$.

3.2 Quantifications

Since the domain of the variables is infinite, quantification is more complicated than the binary Boolean connectives. Based on the Fourier-Motzkin method [16], we perform an existential quantification of a variable x in a DDD rooted at u by removing all vertices reachable from u containing x, but keeping all *implicit* constraints induced by x among the other variables. For example, quantifying out x in the expression ϕ given in (2) yields $\exists x.\phi = y - z \geq 1$, see Fig. 2. Here, the constraint $y - z \geq 1$ does not occur explicitly in ϕ, but implicitly because of $y - x \geq 0$ and $x - z \geq 1$.

To compute $\exists x.(x_i - x_j \lesssim c \to h, l)$, we consider two cases: If x is different from both x_i and x_j, we can push down the quantifier one level in the DDD:

$$\exists x.(x_i - x_j \lesssim c \to h, l)\ =\ x_i - x_j \lesssim c \to \exists x.h, \exists x.l \qquad \text{if } x \notin \{x_i, x_j\}.$$

If x is equal to x_i or x_j, we *relax* all paths in h and l with $x_i - x_j \leq c$ and $x_i - x_j > c$, respectively, and combine the results with disjunction:

$$\begin{aligned}\exists x.(x_i - x_j \leq c \rightarrow h, l) &= \exists x.\textsc{Relax}(h, x, x_i - x_j \leq c) \\ &\quad \vee \exists x.\textsc{Relax}(l, x, x_j - x_i < -c) \qquad \text{if } x \in \{x_i, x_j\}.\end{aligned}$$

If x is equal to x_i, relaxation of a path p with a constraint $x_i - x_j \lesssim c$ consists of adding a new constraint $x_i - x_j \lesssim c + c$ to p for each constraint $x_i - x_i \lesssim c$ in p.[2] The case where x is equal to x_j is symmetric. In worst case, each relaxation generates a quadratic number of new constraints. Thus, a conservative bound on the number of added constraints in an existential quantification $\exists x.u$ is $O(|u|^3)$ because each vertex in u is relaxed once. However, to maintain orderedness these new constraints cannot be added where they are discovered through calls to MK, but need to be added through calls to APPLY. The repeated calls to APPLY imply that the running time of $\exists x.u$ is worst-case exponential.

3.3 Assignment and Replacement

The operations of *assignment* and *replacement* are often used in verification. After performing an *assignment* $\phi[x \leftarrow y + c]$ the variable x is given the value of another variable y plus a constant c in the expression ϕ. When $x \neq y$, performing an assignment corresponds to removing all explicit bounds on x, and then updating x with a new value. The assignment operation $\phi[x \leftarrow y + c]$ is therefore performed as:

$$\phi[x \leftarrow y + c] = (\exists x.\phi) \wedge (x - y = c) \qquad \text{if } x \neq y\,.$$

If x is equal to y, an assignment corresponds to incrementing x by the value c. In these cases, the assignment is performed in linear time by adjusting the constants of all vertices containing the variable x.

The *replacement* operator $\phi[y + c/x]$ syntactically substitutes all occurrences of x in ϕ with another variable y plus a constant c. When the two variables are different, a replacement is performed as:

$$\phi[y + c/x] = \exists x.\big(\phi \wedge (x - y = c)\big) \qquad \text{if } x \neq y\,.$$

If x is equal to y, the replacement $\phi[x + d/x]$ is defined as $\phi[t/x][x + d/t]$, where t is a variable different from x and not occurring in ϕ.

We can avoid the quantification by performing the replacement $\phi[y + c/x]$ directly on each vertex in ϕ by replacing all occurrences of x with $y + c$. This is advantageous when x and y are neighbors in the variable ordering (this is often the case in model checking), since replacement then can be performed in linear time.

[2] In terms of the constraint graph [14, p. 541] defined by p, relaxation with $x_i - x_j \lesssim c$ corresponding to an edge from x_j to x_i creates a new edge from x_j to x_i' with weight $c + c'$ for each edge from x_i to x_i' with weight c' (i.e., the edge from x_j to x_i' is now explicit, not implicit via x_i).

4 Path Reduced DDDs

The previous section describes algorithms for constructing locally reduced DDDs. However, locally reduced DDDs are not a canonical representation of difference constraint expressions. In this section we show how to remove some of the redundant constraints in a path, making the representation semi-canonical. In a semi-canonical representation, there is exactly one DDD for a tautology (the terminal **1**) and exactly one DDD for an unsatisfiable expression (the terminal **0**). Thus, with semi-canonical DDDs it is straightforward to test for validity, satisfiability, and equivalence (after using APPLY with a biimplication).

4.1 Paths and Semi-canonical DDDs

A *path* in a DDD corresponds to a conjunction of difference constraints or negated difference constraints (whenever the path follows a low-branch). Since the negations always can be removed by swapping the variables, changing the comparison operator, and negating the constant, a path corresponds to a conjunction of difference constraints, also called a *system of difference constraints* [14, Sect. 25.5]. We denote the system of difference constraints induced by a path p by $[p]$. A path p is defined to be *feasible* if the corresponding system of difference constraints has a feasible solution. If the constraint system has no solution, the path is *infeasible*.

Definition 4 (Path-reduced DDD). *A path-reduced DDD (R_PDDD) is a locally reduced DDD where all paths are feasible.*

Paths ending at the terminals **0** and **1** are called **0**-paths and **1**-paths, respectively. If a DDD has no infeasible **0**-paths and **1**-paths, then it has no infeasible paths because a feasible constraint system will still be feasible if we remove some of the difference constraints from it. So if all **0**-paths and **1**-paths in a DDD u are feasible, then u is path reduced. For R_PDDDs it is straightforward to decide satisfiability and validity:

Theorem 1 (R_PDDDs are semi-canonical). *In an R_PDDD, the terminal vertex* **1** *is the only representation of a tautology and the terminal vertex* **0** *is the only representation of an unsatisfiable expression.*

Proof. We show that if v is a non-terminal in a path reduced DDD, then v represents neither a tautology nor an unsatisfiable expression. Because v is path reduced, it is also locally reduced, so all vertices u reachable from v satisfy $low(u) \neq high(u)$. Furthermore, because v is a non-terminal vertex in an (acyclic) ordered DDD, there exists some vertex u reachable from v that has $low(u) = \mathbf{0}$ and $high(u) = \mathbf{1}$ or $low(u) = \mathbf{1}$ and $high(u) = \mathbf{0}$. Consequently, both **0** and **1** are reachable from v. Let p be some **0**-path from v. Per definition of path reducedness, we know that p is feasible. This implies that there exists a variable assignment satisfying $[p]$, meaning that there exists a falsifying variable assignment for v. Thus, v cannot represent a tautology. Similarly, because there is a feasible **1**-path from v, v is satisfiable. □

4.2 Reduce

An algorithm for making a DDD rooted at u path reduced is:

```
1 PathReduce(u) = Reduce(u, ⟨u⟩)
2 where Reduce(v, p) =
3   if [p] is infeasible then return ⊥
4   elsif v ∈ {0, 1} then return v
5   else h ← Reduce(high(v), p^⟨high(v)⟩)
6        l ← Reduce(low(v), p^⟨low(v)⟩)
7        if l ≠ ⊥ and h ≠ ⊥ then return Mk(var(v), bound(v), h, l)
8        elsif h ≠ ⊥ then return h
9        else return l
```

The operator $\widehat{\ }$ denotes path concatenation. The function $\textsc{Reduce}(v, p)$ returns $\bot$ if and only if the path p is infeasible. Clearly, if p is infeasible, $\textsc{Reduce}(v, p)$ returns $\bot$ in line 3. On the other hand, if p is feasible, it is simple to see that either $p\widehat{\ }\langle high(v)\rangle$ or $p\widehat{\ }\langle low(v)\rangle$ is feasible, and thus $\textsc{Reduce}(v, p)$ cannot return $\bot$ in line 9. Hence, $\textsc{Reduce}(v, p) = \bot$ if and only if p is infeasible.

The correctness of PathReduce then follows from the following observation: if either $h = \bot$ or $l = \bot$ in lines 5 and 6, the vertex v can be removed. To see this, let $[p]$ denote the system of difference constraints corresponding to the path p and let $\alpha = cstr(v)$ denote the difference constraint of vertex v. Assume $l = \bot$, i.e., the path $p\widehat{\ }\langle low(v)\rangle$ is infeasible, and thus $[p] \wedge \neg\alpha =$ **false**. Then,

$$[p] \wedge \alpha = ([p] \wedge \alpha) \vee ([p] \wedge \neg\alpha) = [p] \wedge (\alpha \vee \neg\alpha) = [p]\,.$$

It follows from a symmetric argument that the vertex v can be removed if $h = \bot$.

Let us consider a small example. Figure 3 shows an $\mathrm{R_L}$DDD for the expression

$$\phi = x - z \geq 0 \ \vee \ y - z \leq 0 \ \vee \ y - x \geq 0. \tag{5}$$

The **0**-path corresponds to the system of difference constraints $x - z < 0$, $z - y < 0$, and $y - x < 0$, which has no feasible solution. Thus, if we call PathReduce on the root vertex, the Reduce-call on the vertex containing $y - x < 0$ returns **1**, and because of the third local reduction requirement the result is the terminal **1**.

There are several algorithms for determining whether a system of difference constraints is feasible. Two well-known ones are Floyd-Warshall's algorithm and Bellman-Ford's algorithm [14], which both have worst-case running times $O(n^3)$, where n is the number of variables. $\textsc{PathReduce}(u)$ enumerates all paths in u, and because the

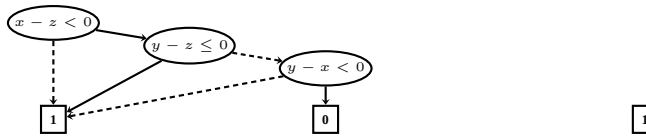


Fig. 3. The expression ϕ from (5) as (a) a locally reduced DDD, and (b) a path reduced DDD.

number of paths can be exponential in the size of u, the complexity of PATHREDUCE(u) is $O(2^{|u|}n^3)$. PATHREDUCE can be improved by using a faster algorithm to determine feasibility of a path, and by reusing the result of the feasibility check in the two recursive calls. These optimizations can be realized by an *incremental* version of the Bellman-Ford algorithm, but although these optimizations in practice improve the performance of PATHREDUCE, they do not improve on the worst-case runtime.

As shown in Theorem 1, it is straightforward to determine whether a path reduced DDD represents a tautology and whether it is satisfiable. However, in practice it is often more efficient to search for a counterexample when checking for validity or satisfiability. For instance, when checking for validity, PATHREDUCE can be modified to stop (and report **false**) if a feasible **0**-path is found. Similarly, when checking for satisfiability, the algorithm can stop (and report **true**) if a feasible **1**-path is found. This approach also leads to a practical algorithm for finding a satisfying variable assignment, called ANYSAT. The algorithm searches for a feasible **1**-path and if one is found, the corresponding system of difference constraints is solved, yielding a satisfying assignment.

5 Fully Reduced DDDs

The reductions ensuring local and path reducedness are quite powerful. As an example consider the two sets built from nine triangles as shown in Fig. 4(a). They each contain nine convex regions representable by 15 non-terminal DDD vertices using the ordering $(x, z) \prec (y, z) \prec (y, x)$. Computing the disjunction of the two sets using APPLY results in the 3×3-square represented with only four non-terminal vertices in an R$_\text{P}$DDD. As another example consider the nine sets shown in Fig. 4(b). Combined they yield a simple convex square although no two sets together form a convex region. Using difference bound matrices similar powerful reductions are very expensive to obtain.

However, path-reducedness is not enough to ensure a canonical representation. As an example, consider the three path-reduced DDDs of Fig. 5 which all represent the same triangular area shown in Fig. 5(d). Local and path reductions are too weak to identify them. One problem (shown in Fig. 5(a)) is that the constraints may contain a certain amount of slack. For instance, the constraint $x - z \geq 0$ could be tightened to $x - z \geq 2$ without changing the semantics. To avoid this kind of slack we introduce a notion of a path being *tight* which strengthens the notion of path reducedness.

To introduce tightness we need to distinguish the dominating constraints in a path. Formally, a constraint $x_i - x_j \lesssim c$ is *dominating* in a path p if all other constraints $x_i - x_j \lesssim c$ on the same pair of variables in p, are less restrictive, i.e., $(c, \lesssim) < (c\,, \lesssim)$.

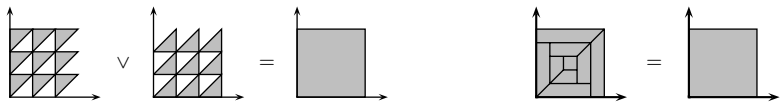


Fig. 4. Disjunctions of complex sets can reduce to simple R$_\text{P}$DDDs.

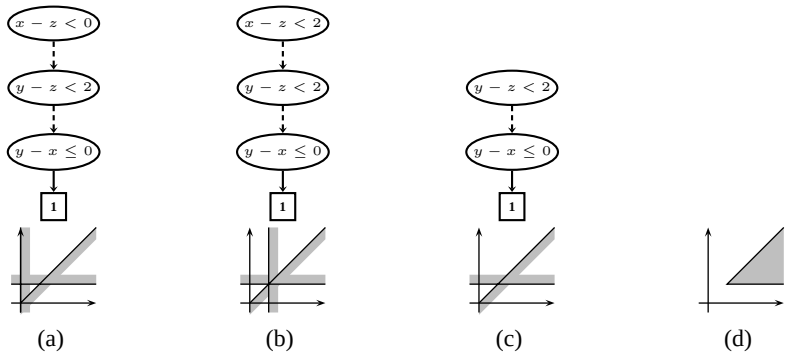


Fig. 5. Three R$_P$DDDs representing the same set (all plots are for $z = 0$).

Non-dominating constraints occur only in paths that through low-edges pass through several vertices with constraints on the same pair of variables.

Definition 5 (Tightness). *A dominating constraint $\alpha = x_i - x_j \lesssim c$ is* tight *in a feasible path $[p] = [p_1] \wedge \alpha \wedge [p_2]$ if for all tighter constraints $(c\ , \lesssim\ ) < (c, \lesssim)$, the systems $[p_1] \wedge (x_i - x_j \lesssim\ c\ ) \wedge [p_2]$ and $[p]$ have different solutions. A path p is* tight *if it is feasible and all dominating constraints on it are tight. An R$_L$DDD u is* tight *if all paths from u are tight.*

From the definition it is clear that tightness generalizes path reducedness since any tight DDD is also an R$_P$DDD. Hence, Theorem 1 implies that it is trivial to determine satisfiability and validity of tight DDDs.

Adding tightness as a condition prevents the existence of the DDD in Fig. 5(a). A DDD can be made tight by enumerating all paths, for each path solve the associated system of difference constraints, replacing the bounds of the constraints by the bounds from the solution, and finally combine all the tight paths by disjunction using APPLY. Hence, the DDD (a) will get reduced to the DDD (b).

Tight DDDs are still not canonical due to implicit constraints that arise as consequences of the constraints in the vertices. The solution set will not depend on how many of these implicit constraints are made explicit but the resulting DDDs will be different. To remove this arbitrariness, we add these implicit constraints to the DDD:

Definition 6 (Saturation). *A tight path p from an R$_P$DDD is* saturated *if for all constraints α not on p, if α is added to p either (1) α is not dominating and tight, or (2) the constraint system $[p_1] \wedge \neg\alpha$ is infeasible, when $[p]$ is written $[p] = [p_1] \wedge [p_2]$ with all constraints on p_1 smaller than α with respect to $\prec$ and all constraints on p_2 larger than α. An R$_P$DDD u is* saturated *if all paths from u are saturated.*

Saturation can be obtained by making as many implicit constraints as possible explicit without introducing any infeasible paths in the DDD. As an example, the DDD in Fig. 5(c) will be saturated into the DDD in Fig. 5(b). However, tight and saturated DDDs are still not canonical. Figure 6 shows an example of two tight, saturated R$_P$DDDs that are equivalent. Intuitively, the problem is that the vertex with the constraint $y - x \leq 0$ is redundant, since the solution set is the area $x - z \geq 1, y - z \geq 1$. To detect such situations, a further check is necessary:

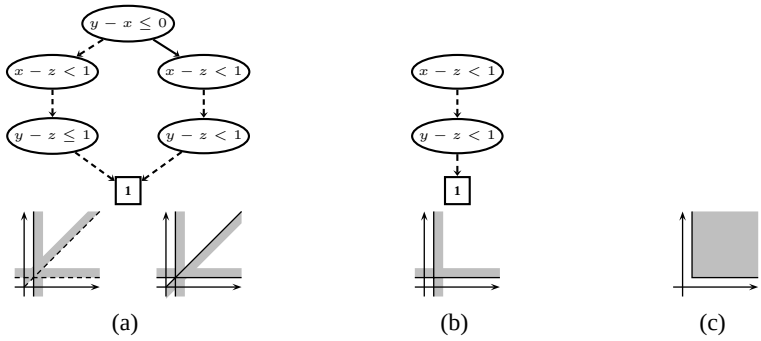


Fig. 6. An example where the mergeability test is necessary to merge two paths into one, making the top constraint redundant (all plots are for $z = 0$).

Definition 7 (Disjunctive vertex). *Let p be a path leading to the vertex u in a DDD, and assume $\alpha = cstr(u)$, $h = high(u)$, and $l = low(u)$. Then u is* disjunctive in p *if $[p] \wedge (\alpha \rightarrow h, l)$ and $[p] \wedge (h \vee l)$ have the same set of solutions.*

This leads us to the following definition and accompanying conjecture:

Definition 8 (Fully reduced DDD). *An R_PDDD u is a* fully-reduced DDD *(R_FDDD) if it is tight, saturated, and has no disjunctive vertices.*

Conjecture 1 (Canonicity). If u and v are $\mathrm{R_F}$DDDs with the same set of solutions then $u = v$.

As it is illustrated by the above discussion, canonicity is rather difficult to obtain in DDDs. This is quite unlike the situation for BDDs, where local reductions and a total ordering of the variables is enough to obtain it. The reason is that in DDDs there are non-local dependencies among the various constraints giving rise to not only untight constraints but also implied constraints that may or may not be explicitly present. Pragmatically, the lack of canonicity of path-reduced DDDs might not be a problem. The main benefit of the canonicity of ROBDDs is that the questions of equivalence, satisfiability, and validity are trivial to answer. However, as pointed out in Theorem 1, satisfiability and validity is trivial for path-reduced DDDs and even for local-reduced DDDs the questions can be solved by a simple on-the-fly search for feasible paths. The crucial issue is whether the representation during computations stay compact which can occur with just a semi-canonical representation.

6 Experimental Results

DDDs can be used to analyze timed system efficiently by representing sets of discrete states and their associated timing information implicitly. The DDD algorithms implement all operations necessary for analyzing general systems with time such as timed guarded commands [17], timed automata [1] or timed Petri nets [7].

Table 1. Experimental results for Milner's scheduler with (a) one clock using the bounds $[H^l, H^u] = [25, 200]$, and (b) one clock per task using the bounds $[H^l, H^u] = [25, 200]$ and $[T^l, T^u] = [80, 100]$. The first column shows the number of cyclers, and the following three columns show the CPU time (in seconds) to build the reachable state space using KRONOS (ver. 2.2b), UPPAAL (ver. 2.17), and DDDs, respectively. The results were obtained on a Pentium II PC with 64 MB of memory. A '$-$' denotes that the analysis did not complete within an hour.

N	KRONOS	UPPAAL	DDD
4	0.2	0.1	0.1
5	0.7	0.2	0.1
6	22.6	0.6	0.1
7	339.2	2.3	0.1
8	–	9.0	0.2
9	–	35.0	0.2
10	–	138.4	0.2
11	–	529.8	0.2
12	–	2560.7	0.3
16	–	–	0.5
32	–	–	2.2
64	–	–	15.9
128	–	–	123.3
256	–	–	1104.8

(a)

N	KRONOS	UPPAAL	DDD
4	0.4	0.2	0.2
5	2.4	1.7	0.3
6	24.2	17.6	0.5
7	346.6	201.7	0.5
8	–	2460.2	0.6
16	–	–	1.5
32	–	–	5.7
64	–	–	31.7
128	–	–	217.3

(b)

In [22] we show how to analyze two different timed versions of Milner's scheduler. Milner's scheduler [20] consists of N cyclers, connected in a ring, that cooperate on controlling N tasks. The two versions of Milner's scheduler are simple, regular and highly concurrent systems, and they illustrate the advantages of a symbolic approach based on difference decision diagrams. With an implementation based on DDDs, the runtimes for computing the reachable state space are several orders of magnitudes better than those obtained with two state-of-the-art tools, KRONOS [28] and UPPAAL [19].

In the first version we use a clock H to ensure that a cycler passes the token on to the following cycler within a bounded amount of time $[H^l, H^u]$. Table 1(a) shows the runtimes to build the reachable state space for increasing N. The number of discrete states in this version of Milner's scheduler is exponential in N since a task can terminate independently of the other tasks. Thus, state space exploration based on enumerating all discrete states as in UPPAAL and KRONOS only succeeds for small systems. The DDD-based approach represents discrete states implicitly yielding polynomial runtimes.

In the second version of Milner's scheduler we use a clock T_i for each task to ensure that it terminates within a certain bound $[T^l, T^u]$ after it is started. Table 1(b) shows the runtimes to build the reachable state space for increasing N. Again, the runtimes of KRONOS and UPPAAL are exponential in N, while using the DDD data structure results in polynomial runtimes. The problem for KRONOS and UPPAAL is the large number of clock variables which is handled in the DDD-based approach by eliminating unused clocks from the representation (i.e., we quantify out T_i whenever task t_i terminates).

7 Conclusion

The problem addressed in this paper is how to efficiently represent and manipulate a Boolean logic over integer- or real-valued inequalities of the form $x - y \leq c$. We have

proposed a data structure inspired by BDDs for representing the expressions from the logic as a decision diagram in which the test conditions are difference constraints.

Introducing an ordering of the constraints makes it possible to extend the APPLY algorithm for ordered BDDs to ordered DDDs without changing its runtime complexity. However, since the domain of the variables in the logic is infinitary, other operations such as existential quantification, are more difficult than for BDDs. For ordered DDDs, these algorithms are basically polynomial, but they become exponential due to the ordering requirement. Another complication is that there are implicit constraints among the variables causing the DDD data structure to be non-canonical even when local reductions are used. A first step towards canonicity is to eliminate all infeasible paths. Such a path-reduced DDD can be tested for validity and satisfiability in constant time. However, semantically equivalent DDDs may still have different representations. We have defined several additional restricting conditions which we conjecture will result in canonical DDDs. It is clearly difficult to obtain an efficient canonical representation. Although canonicity would be intriguing to obtain and allow one to check for equivalence in constant time, it is not necessarily desirable in practice. A canonical representation will not necessarily be more compact than a non-canonical representation and the equivalence check can be performed as a validity check.

Boolean variables can be modeled as difference constraints, making it possible to combine Boolean, continuous, and integer variables within a single data structure. All operations on the Boolean variables in the DDD are performed as efficiently as with BDDs. One use of combining Boolean and real-valued variables is in constructing the set of reachable states for a concurrent timed system. The effectiveness of the data structure and associated algorithms is demonstrated by analyzing two timed versions of Milner's scheduler for which the set of reachable states are computed in polynomial time using DDDs, while the tools KRONOS and UPPAAL both take exponential time.

One path that could be taken when extending the results of the paper would be to generalize the difference constraints to linear inequalities $\sum_{i=1}^{n} a_i x_i \lesssim c$ ordered by a total ordering. The basic data structure and the APPLY algorithm would be unchanged. In the existential quantification the only change is in RELAX, where x is isolated and new inequalites are obtained by substituting the inequality for x. In eliminating infeasible paths, a general linear programming solver must be used, e.g, the simplex algorithm.

References

1. R. Alur and D. Dill. The theory of timed automata. In *Real-Time: Theory in Practice*, LNCS 600, pages 28–73. Springer-Verlag, 1991.
2. E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data-structures for the verification of timed automata. In *Int. Workshop on Hybrid and Real-Time System*, 1997.
3. F. Balarin. Approximate reachability analysis of timed automata. In *Real-Time Systems Symposium*, pages 52–61. IEEE, 1996.
4. G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. Technical Report 99/105, Uppsala Univ., 1999.
5. W. Belluomini and C. J. Myers. Efficient timing analysis algorithms for timed state space exploration. In *Int. Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1997.

6. W. Belluomini and C. J. Myers. Verification of timed systems using POSETs. In *Computer Aided Verfication (CAV)*, June 1998.
7. B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.
8. M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *Computer Aided Verification*, LNCS 1254, pages 179–190, 1997.
9. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
10. J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, Carnegie Mellon, August 1992.
11. S. V. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *Real-Time Systems Symposium*, pages 266–70. IEEE, December 1994.
12. W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *Computer Aided Verification*, pages 316–27, 1997.
13. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.
14. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1994.
15. D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, LNCS 407. Springer, 1989.
16. J.B.J. Fourier. Second extrait. In *Oeuvres*, pages 325–328, Paris, 1890. Gauthiers-Villars.
17. T. A. Henzinger, Z. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.
18. K. G. Larsen, P. Pettersson, and W. Yi. Model-checking for real-time systems. In *Conference on Fundamentals of Computation Theory*, LNCS 965, pages 62–88, August 1995.
19. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1+2), 1997.
20. Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
21. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. Technical Report IT-TR-1999-023, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, February 1999.
22. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Fully symbolic model checking of timed systems using difference decision diagrams. In *FLoC'99 Workshop on Symbolic Model Checking*, July 1999. Available from the Electronic Notes in Theoretical Computer Science repository: `http://www.elsevier.nl/locate/entcs`.
23. T. G. Rokicki. *Representing and Modeling Digital Circuits*. PhD thesis, Stanford University, 1993.
24. T. G. Rokicki and C. J. Myers. Automatic verification of timed circuits. In D. L. Dill, editor, *Computer Aided Verification (CAV)*, LNCS 818, pages 468–480, 1994.
25. R. L. Spelberg, H. Toetenel, and M. Ammerlaan. Partition refinement in real-time model checking. In *Proceedings of FTRTFT'98*, pages 143–57, 1998.
26. E. Verlind, G. de Jong, and B. Lin. Efficient partial enumeration for timing analysis of asynchronous systems. In *ACM/IEEE Design Automation Conference*, 1996.
27. H. Wong-Toi and D.L. Dill. Approximations for verifying timing properties. In *Theories and Experiences for Real-Time Systems Development*. World Scientific Publishing, 1994.
28. S. Yovine. Kronos: A verification tool for real-time systems. *Springer Int. Journal of Software Tools for Technology Transfer*, 1(1/2), October 1997.

Analysis of Hybrid Systems: An Ounce of Realism Can Save an Infinity of States*

Martin Fränzle**

Department of Computer Science
Carl-von-Ossietzky Universität Oldenburg
P.O. Box 2503, D-26111 Oldenburg, Germany

Abstract. Hybrid automata have been introduced in both control engineering and computer science as a formal model for the dynamics of hybrid discrete-continuous systems. In the case of so-called linear hybrid automata this formalization supports semi-decision procedures for state reachability, yet no decision procedures due to inherent undecidability [4]. Thus, unlike finite or timed automata, already linear hybrid automata are out-of-scope of fully automatic verification.
In this article, we devise a new semi-decision method for safety of linear and polynomial hybrid systems which may only fail on pathological, practically uninteresting cases. These remaining cases are such that their safety depends on the *complete* absence of noise, a situation unlikely to occur in real hybrid systems. Furthermore, we show that if low probability effects of noise are ignored akin to the way they are suppressed in digital modelling then safety becomes fully decidable.

Keywords: Hybrid Systems, Verification, Decision Procedures

1 Introduction

Hybrid systems consist of interacting discrete and continuous components. Most embedded systems belong to this class of systems, as they operate within tightly coupled networks of both types of components. Consequently, integration of discrete and continuous reasoning within a single formal model has recently attracted much interest. The hope is that such combined formalisms may ultimately help in developing real embedded systems.

Among such formalisms, the automata-based ones provide the most immediate prospect for mechanization. Roughly, hybrid automata can be characterized as a *combination of finite automata* whose transitions are triggered by predicates on the continuous plant state *with a description of the evolution of the continuous plant*. The latter consists of a set of real-valued variables that are governed by sets of syntactically restricted differential (in-)equations from which a currently active set is selected depending on the current automaton state. Mechanization

* This article reflects work that has been partially funded by the Deutsche Forschungsgemeinschaft under contract DFG Da 206/5-2.

** Email: Martin.Fraenzle@Informatik.Uni-Oldenburg.De,
Phone/Fax: +49-441-798-2412/2145.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 126–140, 1999.

of hybrid-automaton verification has partly become true, as e.g. the HyTech tool [3] supports among others a semi-decision procedure for state reachability in linear hybrid automata, which can — at least in principle, if complexity does not become prohibitive — be used for effectively *falsifying* safety properties. However, as state reachability has been shown to be undecidable for linear hybrid automata [4], the complementary problem, i.e. *verifying* a safety property by showing that no undesirable state may ever be reached, cannot always be done effectively.

However, it is illustrating to take a closer look at the proof technique used in [4] for showing undecidability of state reachability in linear hybrid automata. The core machinery is an instantiation of the following proof pattern:

> *Effectively encode two-counter machines by hybrid automata, representing the counter values by two continuous variables of bounded range. E.g., by variables of range* $[0, 1]$ *through the embedding* ε *defined as* $\varepsilon(k) = 2^{-k}$.

Although the results thus obtained are formally correct and absolutely well-done, their relevance to the practical design problems hybrid automata are intended to cover is questionable. The encodings used (e.g. ε) encode infinite information, namely the set of natural numbers, within a compact interval of continuous states, whereas the ubiquity of noise limits the information content encodable by any bounded continuous variable encountered in real hybrid systems to a finite value. Hence, on simple information-theoretic grounds, the undecidability results thus obtained can be said to be artefacts of an overly idealized formalization.

However, while this implies that the particular proof pattern sketched above lacks physical interpretation, it does not yield any insight as to whether the state reachability problem for hybrid systems featuring noise is decidable or not. We conjecture that there is a variety of realistic noise models for which the problem is indeed decidable. Within this article, we demonstrate this on a very simple model of noise, which we combine with a pragmatic attitude towards thresholds for noise being considered relevant. In Sect. 3 we devise a new decision method that is able to decide safety for those hybrid automata where safety does not depend on the complete absence of noise, i.e. which, if not unsafe, can tolerate some amount of noise without becoming unsafe. Furthermore, Sect. 4 shows that the aforementioned decision method can cope with arbitrary hybrid systems whenever low-probability effects of noise are neglected. A technical side-condition for both results is that the safety region or the continuous state space of the hybrid automaton be bounded or — which is slightly more general — has strongly finite diameter (cf. Def. 1) wrt. an arbitrary metrics.

Related work. To the best of our knowledge, dynamics of hybrid automata under noise has so far only been analyzed for the subclass of timed automata, where two fundamentally different models of the "robust", noise-resistant behaviour of timed automata have been proposed by Gupta, Henzinger, and Jagadeesan [2] and by Puri [6] (both reports do, however, sketch generalizations to hybrid automata). While in the former line of work, the idealized behavioural model of timed automata is essentially kept and only filtered a posteriori by removing

accepted and adding rejected trajectories that are isolated wrt. some topology on trajectories, the latter is more akin to our approach in that it models the impact of noise (i.e. clock drift, if only timed automata are dealt with) as a widening of the transition relation and defines the reachable states as those that are reachable under arbitrarily small positive noise.

2 Hybrid automata

We start our investigation by providing a formalization of hybrid automata. The class of hybrid automata we will deal with goes beyond linear hybrid automata in that we will allow polynomial (instead of only linear) activities and polynomial (instead of linear) predicates for state invariants, transition guards, and transition effects. The reasons for adopting a more general class than usual are twofold. First, all our results can be shown for this more general class without any extra effort. Second, all definitions, statements, and proofs concerning this class of hybrid automata are more compact as no need to keep state invariants separate from activity predicates and transition guards separate from transition effects arises. Instead, every state and transition can be described by just one polynomial predicate, formalised through the first-order logic over the real-closed field, denoted $\mathrm{FOL}(\mathbb{R},+,\times)$ in the remainder.

Therefore, within this article, a *(polynomial) hybrid automaton of dimensionality* d ($d \in \mathbb{N}$) is a six-tuple

$$(\Sigma, \mathbf{x}, (act_\sigma)_{\sigma\in\Sigma}, (trans_{\sigma\to\sigma'})_{\sigma,\sigma'\in\Sigma}, (initial_\sigma)_{\sigma\in\Sigma}, (safe_\sigma)_{\sigma\in\Sigma}) \ ,$$

where Σ is a finite set, representing the discrete states, and $\mathbf{x} = (x_1, \ldots, x_d)$ is a vector of length d of variable names, the continuous variables of the hybrid system.[1] $(act_\sigma)_{\sigma\in\Sigma}$ is a Σ-indexed family of formulae from $\mathrm{FOL}(\mathbb{R},+,\times)$ with free variables $\overleftarrow{\mathbf{x}}, \mathbf{x}$, representing the continuous activities and corresponding state constraints, and $(trans_{\sigma\to\sigma'})_{\sigma,\sigma'\in\Sigma}$ a doubly Σ-indexed family of formulae from $\mathrm{FOL}(\mathbb{R},+,\times)$ with free variables $\overleftarrow{\mathbf{x}}, \mathbf{x}$, representing the discrete transitions and their guarding conditions. Finally, $(initial_\sigma)_{\sigma\in\Sigma}$ and $(safe_\sigma)_{\sigma\in\Sigma}$ are Σ-indexed families of formulae from $\mathrm{FOL}(\mathbb{R},+,\times)$ with free variables $\mathbf{x}$ representing the initial and the safe states of the hybrid automaton.

The interpretation is as follows:

- An *activity predicate* act_σ defines the possible evolution of the continuous state while the system is in discrete state σ. Hooked variable names in the predicate refer to the values of the corresponding system variables beforethe activity, while undecorated variable names refer to the values thereafter. A satisfying valuation $I_{\overleftarrow{\mathbf{x}},\mathbf{x}}$ of its free variables $\overleftarrow{\mathbf{x}}, \mathbf{x}$ is interpreted as: if the system is in state σ and its continuous variables have values $I_{\overleftarrow{\mathbf{x}}}$ then the continuous variables may evolve to $I_{\mathbf{x}}$ while staying in state σ.

[1] Here and in the following, we use the convention to print vectors of variables or constants in boldface. All these vectors have length d.

Note that this single predicate thus generalizes both the state invariants and the activity functions of classical hybrid automata. E.g. the activity predicate

$$\exists\, \Delta t\,.\ \overleftarrow{x} + 3 \times \Delta t \leq x \ \wedge\ x < \overleftarrow{x} + 5 \times \Delta t \ \wedge\ 4 < x \ \wedge\ x \leq 10$$

encodes both the linear activity $\dot{x} \in [3, 5)$ and the state invariant $x \in (4, 10]$.

– A *transition predicate* $trans_{\sigma \to \sigma'}$ defines when the system may evolve from state σ to state σ' by a discrete transition (i.e., it specifies the transition guard) and what the effect of that transition on the continuous variables is. A satisfying valuation $I_{\overleftarrow{\mathbf{x}}, \mathbf{x}}$ of its free variables $\overleftarrow{\mathbf{x}}, \mathbf{x}$ is interpreted as: if the system is in state σ and its continuous variables have values $I_{\overleftarrow{\mathbf{x}}}$ then the system may evolve to state σ' with its continuous variables taking the new values $I_{\mathbf{x}}$.
Assignments as present in hybrid automata are simple to represent in this framework: e.g. for a hybrid system with continuous variables $(x_1, \ldots, x_8)$ the assignment $x_7 := 5$ (where all other continuous variables are left unchanged) is encoded by the predicate $x_7 = 5 \wedge \bigwedge_{i \in \{1,\ldots,6,8\}} x_i = \overleftarrow{x_i}$. Accordingly, guards are simply encoded through predicates over the hooked variables: e.g. the guard $x_3 > 11$ is represented by $\overleftarrow{x_3} > 11$. A transition with guard $x_3 > 11$ and assignment $x_7 := 5$ is thus encoded by the predicate

$$\overleftarrow{x_3} > 11 \ \wedge\ x_7 = 5 \ \wedge\ \bigwedge_{i \in \{1,\ldots,6,8\}} x_i = \overleftarrow{x_i} \quad .$$

Multiple different transitions between the same state pair can be represented by disjunction of their encodings.

It should be noted that there is no strict need to distinguish between activities and transitions in $\mathrm{FOL}(\mathbb{R}, +, \times)$. This is just a matter of convenience for the further development.

Dynamic behaviour. During execution, hybrid automata engage in an alternating sequence of discrete transitions and evolution phases, where the continuous variables evolve according to an activity. Hence a (partial) execution containing $n \in \mathbb{N}$ transitions comprises n transitions interspersed between $n+1$ evolution phases, where the final states (wrt. both discrete and continuous state components) of the evolution phases meet the initial states of the following transition and vice versa the final states of the transitions meet the initial states of the following evolution phase. Thus reachability of a final discrete state σ' and a final continuous state $I_{\mathbf{x}}$ from an initial discrete state σ and a initial continuous state $I_{\overleftarrow{\mathbf{x}}}$ through an execution containing n transitions can be formalised through the inductively defined predicate $\phi^n_{\sigma \to \sigma'}$, where

$$\phi^0_{\sigma \to \sigma'} = \begin{cases} \mathtt{false}, & \text{if } \ \sigma \neq \sigma' \ , \\ act_\sigma \ , & \text{if } \ \sigma = \sigma' \ , \end{cases}$$

$$\phi^{n+1}_{\sigma \to \sigma'} = \bigvee_{\tilde{\sigma} \in \Sigma} \exists\, \mathbf{x}_1, \mathbf{x}_2\,. \begin{pmatrix} \phi^n_{\sigma \to \tilde{\sigma}}[\mathbf{x}_1/\mathbf{x}] \wedge \\ trans_{\tilde{\sigma} \to \sigma'}[\mathbf{x}_1, \mathbf{x}_2/\ \overleftarrow{\mathbf{x}}, \mathbf{x}] \wedge \\ act_{\sigma'}[\mathbf{x}_2/\ \overleftarrow{\mathbf{x}}] \end{pmatrix}$$

A hybrid automaton is called *safe* iff it may never reach an unsafe state. As usual, an *unsafe state* is considered *reachable* iff there is a partial execution (of arbitrary length) from some initial state to a state not belonging to the set of safe states. An unsafe state is *reachable in at most n steps* iff there is such a partial execution containing at most n discrete transitions.

The latter property can be easily formalised using the formalization of partial run: an *unsafe state is reachable within n steps* iff $\phi^i_{\sigma\to\sigma'} \wedge initial_\sigma[\overleftarrow{\mathbf{x}}\ /\mathbf{x}] \wedge \neg safe_{\sigma'}$ is satisfiable for some discrete states σ and σ' and some $i \leq n$. This is equivalent to satisfiability of the formula

$$Unsafe_n \stackrel{\text{def}}{=} \bigvee_{\sigma' \in \Sigma} Reach^{\leq n}_{\sigma'} \wedge \neg safe_{\sigma'} \ ,$$

where

$$Reach^{\leq n}_{\sigma'} \stackrel{\text{def}}{=} \bigvee_{i \in \mathbb{N}_{\leq n}} \bigvee_{\sigma \in \Sigma} \phi^i_{\sigma\to\sigma'} \wedge initial_\sigma[\overleftarrow{\mathbf{x}}\ /\mathbf{x}] \tag{1}$$

characterizes the continuous states reachable in at most n steps within discrete state σ'. Consequently, an *unsafe state is reachable* iff there is some $n \in \mathbb{N}$ for which $Unsafe_n$ is satisfiable.

Note that $Unsafe_n$ is a formula of $\mathrm{FOL}(\mathbb{R}, +, \times)$ such that reachability of an unsafe state within at most n steps is decidable due to the decidability of $\mathrm{FOL}(\mathbb{R}, +, \times)$ [7]. By successively testing increasing n, this does immediately yield a (well-known) semi-decision procedure for reachability of unsafe states:

Lemma 1 (Semi-decidability of safety). *It is semi-decidable whether a given polynomial hybrid automaton is unsafe.* □

However, this semi-decision procedure does not generalize to a decision procedure: state reachability and thus safety is known to be undecidable even for hybrid automata featuring just two clocks and a single stop-watch [4], where a clock is a continuous variable having constant slope 1 within any activity, and a stop-watch is a continuous variable alternating between slopes 0 and 1 only. Thus, undecidability applies already to hybrid automata with just three continuous variables x_1, x_2, x_3, and activity predicates of only the two forms $(x_1 = \dot{x}_1) \wedge \bigwedge_{i=\{2,3\}}(x_i = \dot{x}_i + \Delta t)$ and $\bigwedge_{i=\{1,2,3\}}(x_i = \dot{x}_i + \Delta t)$.

Of course, there are lucky cases where the set of reachable hybrid states stabilizes finitely, i.e. where $n \in \mathbb{N}$ exists s.t. for each $\sigma' \in \Sigma$, the formula $Reach^{\leq n}_{\sigma'}$ characterizing the continuous state set reachable in at most n steps within discrete state σ' is logically equivalent to its successor $Reach^{\leq n+1}_{\sigma'}$. In such cases, these finite approximations of the reachable hybrid state set cannot only be used for falsifying safety properties (as in the semi-decision procedure of Lemma 1), but also for verifying safety. However, as the undecidability result shows, such stabilization need not occur. Therefore, this analysis, which is supported e.g. by the HyTech tool [3], is only a partial remedy.

3 Noise in hybrid automata

Let us step back for a moment and take a philosophical perspective on the problem just encountered. Unlike finite automata, where the full reach set can always be constructed in a finite number of steps, namely at most the size of the state space, the reach set of hybrid automata need not converge finitely. This suggests that hybrid automata are indeed infinite state systems, where the infinite memory is provided by the continuous state components. However, real hybrid systems are always subject to noise, and thus one might suspect that their continuous components can provide only finite memory. If so, they would in fact be finite state system, with the size of the state space being the product of the size of the discrete state space and the effective size of the continuous state space modulo noise.

But then, the reach set computation of hybrid automata modeling real systems should also converge finitely, just as with finite automata, yielding decidability of state reachability and safety. And, vice versa, any hybrid automaton for which safety cannot be determined finitely would then be an *unrealistic* one which crucially relies on complete absence of noise for realizing an infinite state set. We conjecture that the reach set of such a hybrid automaton is practically uninteresting in that it changes drastically under even the slightest disturbance.

It is the theme of the remainder of this article to make these ideas operational. In particular, we devise a new procedure for determining state reachability in polynomial hybrid automata which may fail only on extremely noise-sensitive borderline cases. Non-termination of this procedure may only occur with hybrid automata that cannot reach the questioned states, yet may reach them under even the slightest disturbance. Thus, safety (in the sense of not reaching an undesirable state) of these automata is practically uninteresting as it crucially depends on complete absence of noise, and therefore is just a fiction.

3.1 A simple model of noise

We begin by formalizing a simple model of noise in hybrid automata. Within this model we assume that noise will leave the discrete state set and transitions unaffected, but will make the evolution phases more nondeterministic. I.e., given a hybrid automaton

$$A = (\Sigma, \mathbf{x}, (act_\sigma)_{\sigma\in\Sigma}, (trans_{\sigma\to\sigma'})_{\sigma,\sigma'\in\Sigma}, (initial_\sigma)_{\sigma\in\Sigma}, (safe_\sigma)_{\sigma\in\Sigma}) \ ,$$

a *disturbed variant* of A is any hybrid automaton

$$\tilde{A} = (\Sigma, \mathbf{x}, (\widetilde{act}_\sigma)_{\sigma\in\tilde{\Sigma}}, (trans_{\sigma\to\sigma'})_{\sigma,\sigma'\in\tilde{\Sigma}}, (initial_\sigma)_{\sigma\in\tilde{\Sigma}}, (safe_\sigma)_{\sigma\in\tilde{\Sigma}})$$

with $act_\sigma \Rightarrow \widetilde{act}_\sigma$ for each $\sigma \in \Sigma$.

Now assume that the continuous state space comes equipped with a first-order definable metrics *dist*, with $dist(\mathbf{x}, \mathbf{y})$ being its definition in $\mathrm{FOL}(\mathbb{R}, +, \times)$.

Furthermore, let ε be a first-order definable constant in $\mathbb{R}_{>0}$. We say that a disturbance $\tilde{A}$ of A is a *disturbance of noise level ε or more* wrt. *dist* iff

$$\exists \mathbf{y} \,.\, \big(act_\sigma[\mathbf{y}/\mathbf{x}] \wedge dist(\mathbf{x},\mathbf{y}) < \varepsilon\big) \Rightarrow \widetilde{act}_\sigma \tag{2}$$

for each $\sigma \in \Sigma$. Thus, evolution phases can drift away within at least an open ball of radius ε under such disturbances. It seems reasonable to assume that within any realistic noise field such disturbances exist for some sufficiently small ε.[2]

Obviously, a disturbance will yield additional possible behaviours, possibly leading to otherwise unreachable states. In particular, the disturbed system yields an overapproximation of the undisturbed system. It comes as a surprise that, under fairly mild extra conditions, such an overapproximation can be constructed within a finite number of steps, as the following lemma shows.

Before we can turn to this central lemma, we have to define the crucial notion of sets of strongly finite diameter.

Definition 1 (Strongly finite diameter). *Let $S \subseteq \mathbb{R}^d$. We say that S has* finite diameter *wrt. the metrics dist iff* $\sup\{dist(x,y) \mid x,y \in S\} < \infty$. *We say that S has* strongly finite diameter *wrt. the metrics dist iff for any $\varepsilon > 0$, each subset $P \subseteq S$ containing only points that have a mutual distance of at least ε is finite, i.e.* $\forall \varepsilon > 0, P \subseteq S \,.\, \big(\forall x,y \in P \,.\, \big(x \neq y \Rightarrow dist(x,y) \geq \varepsilon\big) \Rightarrow |P| < \infty\big)$.

Note that the notions of finite diameter and strongly finite diameter coincide for most of the "natural" metrics on $\mathbb{R}^d$, like Euclidean distance, maximum-norm, and taxi-cab metrics, yet differ for some others, like discrete metrics or the so-called radar-screen metrics.

In the following, let A be a hybrid automaton, $\varepsilon > 0$, and $\tilde{A}$ be a disturbance of A of noise level ε or more. By $\widetilde{Reach}^{\leq n}_{\sigma'}$ we denote the predicate obtained from the defining equation (1) when applied to $\tilde{A}$ instead of A. Thus, $\widetilde{Reach}^{\leq n}_{\sigma'}$ is a formula in $\mathrm{FOL}(\mathbb{R},+,\times)$ formalizing the continuous states reachable by the disturbed automaton $\tilde{A}$ within discrete state σ'. Finally, we denote in the remainder for any $\mathrm{FOL}(\mathbb{R},+,\times)$-formula ϕ by $[\![\phi]\!]_{\mathbf{x}}$ the subset

$$\Big\{ (c_1,\ldots,c_d) \in \mathbb{R}^d \Big| \, I \oplus [x_1 \mapsto c_1,\ldots,x_d \mapsto c_d] \models \phi \text{ for some valuation } I \Big\}$$

of $\mathbb{R}^d$.

[2] Concerning the range of applicability of the model, it is worth noting that for the theory exposed in the remainder, the fact that *every* activity is subject to drift of at least ε for some $\varepsilon > 0$ is not essential. While we have chosen such a model in order to simplify exposition, the theory itself can be easily extended to the more general situation that only every cycle in the discrete state space of the hybrid automaton (i.e. every alternating sequence of activities and transitions going from some discrete state σ via other discrete states back to σ) contains at least one activity with drift of at least ε.

Lemma 2 (Finite overapproximation). *Assume that the reach set* $RS_\sigma = \bigcup_{n\in\mathbb{N}} [\![\widetilde{Reach}_\sigma^{\leq n}]\!]_{\mathbf{x}}$ *of* $\tilde{A}$ *has strongly finite diameter wrt. the metrics dist. Then there is* $i \in \mathbb{N}$ *s.t. the dynamics of the undisturbed hybrid automaton* A *is contracting on* $[\![\widetilde{Reach}_\sigma^{\leq i}]\!]_{\mathbf{x}}$*, i.e.*

$$\bigvee_{\sigma'\in\Sigma} \exists\, \mathbf{x}_1, \mathbf{x}_2 \,.\, \begin{pmatrix} \widetilde{Reach}_{\sigma'}^{\leq i}[\mathbf{x}_1/\mathbf{x}] \wedge \\ trans_{\sigma'\to\sigma}[\mathbf{x}_1, \mathbf{x}_2/ \overleftarrow{\mathbf{x}}, \mathbf{x}] \wedge \\ act_\sigma[\mathbf{x}_2/ \overleftarrow{\mathbf{x}}] \end{pmatrix} \;\Rightarrow\; \widetilde{Reach}_\sigma^{\leq i} \tag{3}$$

holds for each $\sigma \in \Sigma$*. I.e., there is some* $i \in \mathbb{N}$ *s.t. the state space reachable in the disturbed automaton* $\tilde{A}$ *within* i *steps is closed under any possible evolution of* A*.*

Proof. We use contraposition and show that RS_σ does not have strongly finite diameter if (3) is invalid for all $i \in \mathbb{N}$. Therefore, we start from the assumption that $out_i \stackrel{\text{def}}{=} \neg(3)$ is satisfiable for each $i \in \mathbb{N}$. We will show that this implies existence of an infinite set $P \subset RS_\sigma$ of ε-separated points, which implies that RS_σ does not have strongly finite diameter. An appropriate P is defined as $\bigcup_{i\in\mathbb{N}}\{p_i\}$ with p_i being an arbitrary element of $[\![out_i]\!]_{\mathbf{x}}$. Note that for each $i \in \mathbb{N}$ such a p_i exists due to satisfiability of out_i. It remains to be shown that

(a) $P \subseteq RS_\sigma$, (b) P is infinite, and (c) P contains only ε-separated points.

For both (b) and (c) it suffices to show that $dist(p_i, p_j) \geq \varepsilon$ for $i < j$. The key argument towards this is illustrated in Fig. 1. For a formal proof, we show that $p \in [\![out_i]\!]_{\mathbf{x}}$ implies that $p' \in [\![\widetilde{Reach}_\sigma^{\leq i+1}]\!]_{\mathbf{x}}$ for each p' with $dist(p, p') < \varepsilon$. Therefore observe that by definition

$$\begin{aligned}
&[\![\widetilde{Reach}_\sigma^{\leq i+1}]\!]_{\mathbf{x}} \\
&= [\![\bigvee_{\sigma'\in\Sigma} \exists\, \mathbf{x}_1, \mathbf{x}_2 \,.\, \begin{pmatrix} \widetilde{Reach}_{\sigma'}^{\leq i}[\mathbf{x}_1/\mathbf{x}] \wedge \\ \widetilde{trans}_{\sigma'\to\sigma}[\mathbf{x}_1, \mathbf{x}_2/ \overleftarrow{\mathbf{x}}, \mathbf{x}] \wedge \\ \widetilde{act}_\sigma[\mathbf{x}_2/ \overleftarrow{\mathbf{x}}] \end{pmatrix}]\!]_{\mathbf{x}} && \text{[Def. of } \widetilde{Reach}_\sigma^{\leq i+1}\text{]} \\
&\supseteq [\![\bigvee_{\sigma'\in\Sigma} \exists\, \mathbf{x}_1, \mathbf{x}_2, \mathbf{y} \,.\, \begin{pmatrix} \widetilde{Reach}_{\sigma'}^{\leq i}[\mathbf{x}_1/\mathbf{x}] \wedge \\ trans_{\sigma'\to\sigma}[\mathbf{x}_1, \mathbf{x}_2/ \overleftarrow{\mathbf{x}}, \mathbf{x}] \wedge \\ act_\sigma[\mathbf{x}_2, \mathbf{y}/ \overleftarrow{\mathbf{x}}, \mathbf{x}] \wedge \\ dist(\mathbf{x}, \mathbf{y}) < \varepsilon \end{pmatrix}]\!]_{\mathbf{x}} && \text{[Property (2) of } \widetilde{act}\text{]} \\
&\supseteq [\![\exists\, \mathbf{y} \,.\, \begin{pmatrix} out_i[\mathbf{y}/\mathbf{x}] \wedge \\ dist(\mathbf{x}, \mathbf{y}) < \varepsilon \end{pmatrix}]\!]_{\mathbf{x}} && \text{[Def. } out_i\text{]} \\
&= \left\{ p' \in \mathbb{R}^d \,\middle|\, dist(p, p') < \varepsilon \text{ for some } p \in [\![out_i]\!]_{\mathbf{x}} \right\} \;.
\end{aligned}$$

Hence, $p \in [\![out_i]\!]_{\mathbf{x}}$ and $dist(p, p') < \varepsilon$ implies $p' \in [\![\widetilde{Reach}_\sigma^{\leq i+1}]\!]_{\mathbf{x}}$. In particular,

$$dist(p_i, p') < \varepsilon \;\text{ implies }\; p' \in [\![\widetilde{Reach}_\sigma^{\leq i+1}]\!]_{\mathbf{x}} \;. \tag{4}$$

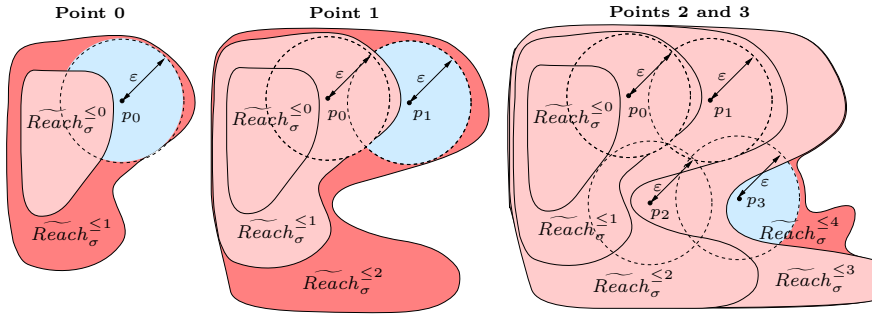


Fig. 1. *Constructing an infinite set of ε-separated points in phase-space.* Points p_i are selected s.t. $p_i \notin [\![\widetilde{Reach}_\sigma^{\leq i}]\!]_{\mathbf{x}}$, yet is reachable from $[\![\widetilde{Reach}_\sigma^{\leq i}]\!]_{\mathbf{x}}$ under a transition followed by an activity of the *undisturbed* automaton A. Consequently, the disturbed automaton $\tilde{A}$ may under the same transition and the corresponding activity reach any point within the ε-ball around p_i s.t. this neighborhood is fully covered by $[\![\widetilde{Reach}_\sigma^{\leq i+1}]\!]_{\mathbf{x}}$. Hence, for $j > i$, point p_j has a minimum distance of ε from p_i.

As $dist(p_i, p_i) = 0$, this shows that $p_i \in [\![\widetilde{Reach}_\sigma^{\leq i+1}]\!]_{\mathbf{x}} \subseteq RS_\sigma$ and thus proves (a). Furthermore, if $i < j$ then $p_j \notin [\![\widetilde{Reach}_\sigma^{\leq i+1}]\!]_{\mathbf{x}}$ can be inferred from the fact that by definition $p_j \in [\![out_j]\!]_{\mathbf{x}}$ and out_j entails $\neg\widetilde{Reach}_\sigma^{\leq j}$, which in turn entails $\neg\widetilde{Reach}_\sigma^{\leq i+1}$ as $i + 1 \leq j$. Therefore, (4) yields $dist(p_i, p_j) \geq \varepsilon$ for $i < j$, which proves (b) and (c). □

Furthermore, it is easy to see that such an $i \in \mathbb{N}$ with the property that the state space reachable in the disturbed automaton $\tilde{A}$ within i steps is closed under any possible dynamic evolution of A can be determined effectively.

Corollary 1. *If the reach set RS_σ of $\tilde{A}$ has strongly finite diameter for each $\sigma \in \Sigma$, then an $i \in \mathbb{N}$ can be* effectively *determined s.t. property (3) holds for each $\sigma \in \Sigma$.*

Proof. For each $i \in \mathbb{N}$, property (3) is a formula of $\mathrm{FOL}(\mathbb{R}, +, \times)$ and thus decidable. Hence, by e.g. μ-recursion, an $i \in \mathbb{N}$ s.t. property (3) holds for each $\sigma \in \Sigma$ can be determined effectively iff there is such an i. However, existence of an $i \in \mathbb{N}$ s.t. property (3) holds is guaranteed by Lemma 2. □

Now assume that we have determined an $i \in \mathbb{N}$ such that the state space reachable in the disturbed automaton $\tilde{A}$ within i steps is closed under any possible evolution of A. As the state space reachable by $\tilde{A}$ within i steps trivially covers the initial state set of $\tilde{A}$ and thus of A, closure of this state space under the possible dynamic evolutions of A implies that this state space covers all states reachable by A. This yields the following corollary:

Corollary 2. *If the reach set RS_σ of $\tilde{A}$ has strongly finite diameter for each $\sigma \in \Sigma$, then an $i \in \mathbb{N}$ can be effectively determined s.t. for each $\sigma \in \Sigma$*

$$\bigcup_{n \in \mathbb{N}} [\![Reach_\sigma^{\leq n}]\!]_{\mathbf{x}} \subseteq [\![\widetilde{Reach}_\sigma^{\leq i}]\!]_{\mathbf{x}} \ .$$

Proof. By Cor. 1 an $i \in \mathbb{N}$ satisfying property (3) can be effectively determined if the reach set RS_σ of $\tilde{A}$ has strongly finite diameter for each $\sigma \in \Sigma$. Now, property (3) for each $\sigma \in \Sigma$ implies that $R \stackrel{\text{def}}{=} \bigcup_{\sigma\in\Sigma}\{\sigma\} \times [\![\widetilde{Reach}_\sigma^{\leq i}]\!]_{\mathbf{x}}$ is closed under the possible evolutions of A. As R contains any initial state of $\tilde{A}$ and thus of A, this implies that R covers all reachable states of A. Therefore, $\bigcup_{n\in\mathbb{N}}[\![Reach_\sigma^{\leq n}]\!]_{\mathbf{x}} \subseteq [\![\widetilde{Reach}_\sigma^{\leq i}]\!]_{\mathbf{x}}$ for each $\sigma \in \Sigma$. □

This implies that if $\tilde{A}$ is safe and its reach set has strongly finite diameter then we obtain within a finite number of iterations — and thus effectively — a witness for safety of A.

Now assume that the safety condition is such that it implies strongly finite diameter of the reachable state set, i.e. that $[\![safe_\sigma]\!]_{\mathbf{x}}$ has strongly finite diameter for each $\sigma \in \Sigma$. This gives rise to the peculiar situation that falsification of $\tilde{A}$ and verification of A become tightly coupled:

Corollary 3. *If safety implies strongly finite diameter, i.e. if $[\![safe_\sigma]\!]_{\mathbf{x}}$ has strongly finite diameter for each $\sigma \in \Sigma$,*[3] *then either the disturbed automaton $\tilde{A}$ is unsafe (which can be determined effectively) or safety of the undisturbed automaton A can be determined effectively.*

Proof. We distinguish the two cases that either the reach set RS_σ of $\tilde{A}$ has strongly finite diameter for each $\sigma \in \Sigma$, or not. In the first case there is an $i \in \mathbb{N}$ satisfying property (3) for each $\sigma \in \Sigma$ due to Cor. 1. In the second case, the premiss that safety implies strongly finite diameter implies that $\exists\ \overleftarrow{\mathbf{x}}, \mathbf{x}\,.\,\widetilde{Unsafe}_i$ holds for some $i \in \mathbb{N}$. As both (3) and $\exists\ \overleftarrow{\mathbf{x}}, \mathbf{x}\,.\,\widetilde{Unsafe}_i$ are decidable formulae, the minimum $i \in \mathbb{N}$ such that

$$\bigwedge_{\sigma\in\Sigma}(3)\ \vee\ \exists\ \overleftarrow{\mathbf{x}}, \mathbf{x}\,.\,\widetilde{Unsafe}_i \tag{5}$$

can be determined effectively in either case.

Once such an i has been determined, it remains to check the (decidable) property $\exists\ \overleftarrow{\mathbf{x}}, \mathbf{x}\,.\,\widetilde{Unsafe}_i$. If it holds then $\tilde{A}$ is unsafe. Otherwise (5) implies that (3) holds for each $\sigma \in \Sigma$. But then, according to Cor. 2, $\bigcup_{n\in\mathbb{N}}[\![Reach_\sigma^{\leq n}]\!]_{\mathbf{x}} \subseteq [\![\widetilde{Reach}_\sigma^{\leq i}]\!]_{\mathbf{x}}$, which implies

$$\bigcup_{n\in\mathbb{N}}[\![Reach_\sigma^{\leq n}]\!]_{\mathbf{x}} \subseteq [\![safe_\sigma]\!]_{\mathbf{x}}\ ,$$

as $\exists\ \overleftarrow{\mathbf{x}}, \mathbf{x}\,.\,\widetilde{Unsafe}_i$ does not hold. I.e., A has then been shown to be safe. □

An illustration of the verification procedure outlined in above corollary can be found in Fig. 2.

Now, let us take a look at what is a pragmatically reasonable correctness criterion for hybrid systems. It is pragmatically clear that a system should not be called safe if no more than the slightest disturbance is necessary to render it unsafe. This motivates the following definition of robustness.

Definition 2 (Robustness). *A hybrid system is called* fragile *if it is safe, yet any disturbance of arbitrarily small positive noise level is unsafe. All other hybrid systems are called* robust. *I.e., a hybrid automaton A is robust iff it is unsafe or there is an $\varepsilon > 0$ and a safe disturbance $\tilde{A}$ of noise level ε or more.*

[3] Note that for Euclidean distance or equivalent metrics, this condition is equivalent to the safety region being finitely bounded, a case frequently encountered in practice.

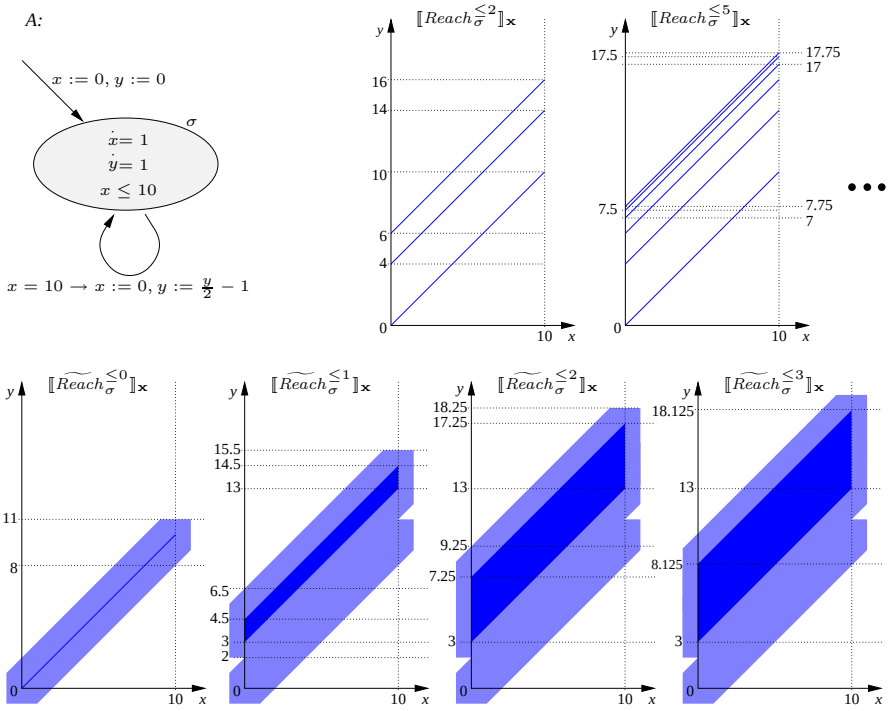


Fig. 2. *Verification using the technique of Cor. 3.* Top left: A hybrid automaton A. Top right: Some steps of its reach set computation. Bottom: Reach set computation for a disturbed variant of noise-level 1 under the max. norm. Shaded parts denote $[\![\widetilde{Reach}_\sigma^{\leq i}]\!]_{\mathbf{x}}$ while solid colour denotes the states reachable by the *undisturbed* automaton from $[\![Reach_\sigma^{\leq i-1}]\!]_{\mathbf{x}}$. Note that reach set computation neither terminates for A nor for its disturbed variant, yet A is contracting on $[\![\widetilde{Reach}_\sigma^{\leq 2}]\!]_{\mathbf{x}}$.

As practical interest clearly is in robust systems only, the following theorem, which states that robust systems can (at least in principle) be automatically verified, is a strong result.

Theorem 1 (Decidability of reachability for robust systems). *If A is a robust hybrid automaton, and if safety implies strongly finite diameter, then it is decidable whether A is safe.*

Proof. A semi-decision procedure for A being unsafe has already been devised in Sect. 1. Hence, it remains to establish a semi-decision procedure for safety of robust automata. This can be done as follows:

1. Select some $n \in \mathbb{N}$.
2. Build the hybrid automaton $\tilde{A} \stackrel{\text{def}}{=} A_{\frac{1}{n}}$, where for arbitrary $\delta \geq 0$,

$$A_\delta \stackrel{\text{def}}{=} (\Sigma, I, \mathbf{x}, (\widetilde{act}_\sigma)_{\sigma\in\Sigma}, (trans_{\sigma\rightarrow\sigma'})_{\sigma,\sigma'\in\Sigma})$$

with $\widetilde{act}_\sigma \stackrel{\text{def}}{=} \exists \mathbf{y} \,.\, act_\sigma[\mathbf{y}/\mathbf{x}] \wedge dist(\mathbf{x},\mathbf{y}) < \delta$.

3. Using Cor. 3, determine whether $\tilde{A}$ is unsafe or A is safe. In the former case select a strictly greater $n \in \mathbb{N}$ than before and redo from step 2. Otherwise a witness for safety of A has been found. Terminate successfully. □

Now, given the fact that *robust* systems can be automatically verified, it remains the question whether robustness can be automatically detected.

Corollary 4 (Undecidability of robustness). *It is undecidable whether a hybrid automaton is robust or fragile, even when we restrict interest to safety predicates that imply strongly finite diameter. It is, however, semi-decidable in the latter case.*

Proof. Let us restrict attention to safety predicates that imply strongly finite diameter. All fragile systems are by definition safe (cf. Def. 2). Hence, safety of a hybrid automaton is trivially decidable once it is known to be fragile. Likewise, by the extra condition on safety predicates, safety of a hybrid automaton is decidable due to theorem 1 once it is known to be robust. Consequently, decidability of being robust would imply decidability of safety for arbitrary hybrid automata. However, state reachability and thus safety is undecidable even for linear hybrid automata with finitely bounded state space (where safety trivially implies strongly finite diameter wrt. e.g. Euclidean distance) [4], and therefore robustness is undecidable.

It is, however, semi-decidable by a minor variation of the decision procedure of theorem 1: a hybrid automaton A is robust iff it is either unsafe or there is some noise level $\varepsilon > 0$ and some disturbed variant $\tilde{A}$ of noise level ε or more that is safe. However, in the latter case A also has a disturbed variant of positive noise level that is not only safe, but also robust itself, e.g. the automaton $A_{\frac{\varepsilon}{2}}$. Therefore, a hybrid automaton A is robust iff it is either unsafe or there is some noise level $\varepsilon > 0$ such that $A_{\frac{\varepsilon}{2}}$ is robust and safe. The first case is semi-decidable by the semi-decision procedure for being unsafe devised in Sect. 1, while the latter case is semi-decidable by applying the semi-decision procedure for safety of robust automata of Theorem 1 to the automata $A_{\frac{1}{2n}}$ for successively smaller n. □

The undecidability result, while disappointing, should nevertheless be no serious obstacle in practice. It is good engineering practice to make systems tolerant against noise. Thus, any well-engineered system should be robust s.t. the decision procedure of theorem 1 comes to an answer. A simple sufficient criterion for robustness is nevertheless currently unknown. This comes as no surprise, as the same applies even for the related notion of robustness of timed automata [6]. However, robustness of timed automata is decidable.

Even without decidability of robustness, it is in principle possible to compute the exact noise margin that a hybrid automaton can cope with. Here, the noise margin that a hybrid automaton A can cope with is defined as

$$Noisemargin(A) \stackrel{\text{def}}{=} \sup \left\{ \varepsilon > 0 \;\middle|\; \begin{array}{l} A \text{ has a } \textit{safe} \text{ disturbance } \tilde{A} \\ \text{of noise level } \varepsilon \text{ or more} \end{array} \right\} ,$$

with $\sup \emptyset \stackrel{\text{def}}{=} 0$, for the sake of completeness.

Theorem 2 (Computability of robustness margins). *For any hybrid automaton A where safety implies strongly finite diameter, the maximum noise margin $Noisemargin(A)$ that A can cope with can be computed effectively (in the sense of computable real numbers).*

Proof. We use the following definition of computable real numbers: a real number x is computable iff there is an algorithm that given as input a positive rational number denoting the desired precision yields an approximation of x of at least that precision. I.e., given any rational number $\delta > 0$ it yields a rational number $\tilde{x}$ with $x-\delta \leq \tilde{x} \leq x+\delta$. To achieve this, it suffices to have an effective procedure that determines for every pair $a < b$ of rational numbers whether $x \leq b$ or $x \geq a$ in the following sense: if $a \leq x \leq b$ then it may give an arbitrary answer; if however $x < a$ ($x > b$, resp.) then it gives the definitive answer $x \leq b$ ($x \geq a$, resp.).

Such a procedure exists for the noise margin: It suffices to apply Cor. 3 to the two automata A_a and A_b instead of A and $\tilde{A}$. The decision procedure outlined there will either prove A_a to be safe, in which case $\mathit{Noisemargin}(A) \geq a$ follows, or proves A_b to be unsafe, in which case $\mathit{Noisemargin}(A) \leq b$ follows. □

4 Practical modelling issues

Let us finally take a pragmatic attitude towards hybrid system modelling. To this end, we would like to abstract from the impact of noise on the continuous components in a similar way as is generally done for the discrete components in finite-state modelling. The digital model used for modelling digital components as finite state systems is inherently approximative, as low probability deviations — e.g., due to noise — from the ideal digital behaviour are simply neglected. Thereby, no identifiable threshold probability is used for distinguishing effects that are to be modelled from those which may be neglected. Instead, a wide variety of thresholds is accepted in favor of a uniform model, ranging from the extremely low probability of noise-induced error within synchronously clocked subsystems to the far higher probability of error at the interface to an asynchronous environment occurring due to metastable states — known as synchronization failure [1]. This leads to the effect that some unmodelled behavioural aspects may well have higher overall probability than some of the modelled effects.

There is no reason to be more demanding wrt. modelling of the continuous behaviour than wrt. discrete behaviour. I.e., just as with discrete transitions the model-builder may freely include or exclude some of the low-probability effects. However, instead of letting the model-builder do the selection we may as well let the verification algorithm do so itself. I.e., the model builder is asked to devise both a hybrid automaton A excluding all (or most) low-probability effects of noise on the continuous components and a hybrid automaton $\tilde{A}$ including all (or most) low-probability effects. This would apparently lead to $\tilde{A}$ being a disturbance of A of noise level ε or more for some $\varepsilon > 0$. As we do not really care for the low-probability effects, the verification algorithm is then deliberately free to construct an arbitrary intermediate model and to decide its safety. I.e., we insist that our verification procedure gives a positive answer whenever both A and $\tilde{A}$ (and hence all intermediate models) are safe and a negative answer whenever both A and $\tilde{A}$ (and hence all intermediate models) are unsafe. But we will accept any answer if A is safe and $\tilde{A}$ is unsafe.

This might look like a complication, but in fact ideally fits the verification procedures outlined in Sect. 3: if safety implies strongly finite diameter then

above proof obligation can be discharged automatically according to Cor. 3. I.e., given such a pragmatic modelling discipline, the safety verification problem for linear or polynomial hybrid automata can be solved fully automatically in the important case of a bounded safety region.

Practitioners may object that practical problems with tools like HyTech [3] are not primarily caused by nontermination of the verification procedure due to the inherent undecidability of state reachability. Often, problems arise due to the extremely large number of iterations needed until the reachable state set stabilizes and due to the numerical precision needed for representing those intermediate state sets. However, the pragmatic modelling discipline sketched here, together with the verification technology of Cor. 3, helps overcome these other problems also. First of all, less applications of the transition relation and the evolution predicates are generally needed for constructing an overapproximation with Cor. 3 than for reaching the fixed point in an incremental calculation of the reachable state space. Second, we may reduce the precision needed in the calculations as we may freely replace $\tilde{A}$ by a slightly less disturbed variant B of A that upon each activity applies some rounding discipline to the corner points of the reachable state sets in order to reduce the necessary numerical precision.

5 Discussion

We have been able to show that safety of linear and polynomial hybrid automata can be decided algorithmically in most practically interesting cases featuring a bounded (or at least strongly finite wrt. some metrics) safety region. The remaining cases are such that their safety depends on the complete absence of noise and, furthermore, apply dissimilar approximation schemes for modelling the discrete and continuous parts of the system, as explained in Sect. 4.

Instrumental to that success has been the insight that a common proof pattern of undecidability results for hybrid-systems formalisms does rely on artefacts of the formalization rather than on an encoding of inherent complexity of the design problem. In a nutshell, those proofs rely on storing infinite information, namely an encoding of the state set of a counter automaton, by continuous variables of bounded range, whereas only a finite part thereof can become effective in any embedded control application due to the ubiquity of noise. We have shown that already with a simplistic model of noise, combined with a pragmatic attitude towards thresholds for noise being considered relevant, the problem of undecidability can be overcome. In this sense, an ounce of realism can save an infinity of states in the analysis of hybrid systems.

However, it may be argued that the model of noise employed in Sect. 3 is too simplistic. Indeed, a realistic model of noise should better be quantitative, representing noise as a probabilistic process. Within this article we have deliberately refrained from this, as such a model would hardly yield any practical verification procedure due to the large computational overhead caused by calculating the density function when iterating the probabilistic transition and evolution steps. I.e., in contrast to the positive effects on the practical complexity of the

Verifying Liveness by Augmented Abstraction

Yonit Kesten * and Amir Pnueli **

Abstract. The paper deals with the proof method of *verification by augmented finitary abstraction* (VAA), which presents an effective approach to the verification of the temporal properties of (potentially infinite-state) reactive systems. The method consists of a two-step process by which, in a first step, the system and its temporal specification are combined an then abstracted into a finite-state Büchi automaton. The second step uses model checking to establish emptiness of the abstracted automaton.
The VAA method can be considered as a viable alternative to verification by temporal deduction which, up to now, has been the main method shown to be complete for the verification of infinite-state systems.
The paper presents a general recipe for the abstraction of Büchi automata which is shown to be *sound*, where soundness means that emptiness of the abstract automaton implies emptiness of the concrete (infinite-state) automaton. To make the method applicable for the verification of liveness properties, pure abstraction is sometimes no longer adequate. We show that by augmenting the system by an appropriate (and standardly constructible) *progress monitor*, we obtain an augmented system, whose computations are essentially the same as the original system, and which may now be abstracted while preserving the desired liveness properties. We then proceed to show that the VAA method is sound and complete for proving all properties expressible by temporal logic (including both safety and liveness). Completeness establishes that whenever an infinite-state Büchi automaton has no computations, there exists a finitary abstraction which abstracts the automaton, augmented by an appropriate progress monitor, into a finite-state Büchi automaton with no computations.

Keyword: Verification, Abstraction, Deduction, Infinite-Systems, Fair Discrete Systems, Completeness, Linear Temporal Logic, Liveness properties.

1 Introduction

When verifying temporal properties of reactive systems, the common wisdom is: if it is finite-state, model-check it, otherwise one must use temporal deduction, supported by theorem provers such as STeP, PVS, etc. The study of abstraction as an aid to verification demonstrated that, in some interesting cases, one can

* Contact author: Dept. of Communication Systems Engineering, Ben Gurion University, Beer-Sheva, Israel, ykesten@bgumail.bgu.ac.il
** Dept. of Applied Mathematics and Computer Science, the Weizmann Institute of Science, Rehovot, Israel, amir@wisdom.weizmann.ac.il

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 141−156, 1999.

abstract an infinite-state system into a finite-state one. This suggests an alternative approach to the temporal verification of infinite-state systems: abstract first and model check later.

In this work, we present a general framework for abstracting an arbitrary (infinite-state) reactive system $\mathcal{D}$ and its specification expressed as a linear temporal logic (LTL) formula ψ, to a finite state problem that can be model-checked. The unique features of this abstraction method is that it takes full account of all the fairness assumptions (including strong fairness) associated with the system $\mathcal{D}$ and can, therefore, establish liveness properties, in contrast to most other abstraction approaches that can only support verification of safety properties.

Applying the method of finitary abstraction for the proofs of liveness properties, we find that, sometimes, pure abstraction is no longer adequate. For these cases, it is possible to construct an additional module M, to which we refer as a *progress monitor*, such that the augmented system $\mathcal{D} \,|||\, M$ has essentially the same set of computations as the original $\mathcal{D}$ and can be abstracted in a way which preserves the desired liveness property. We refer to this extended proof method as the method of *verification by augmented abstraction* (VAA). We proceed in showing that the VAA method is both *sound* and *complete*, thus promoting it to the status of becoming an alternative to the verification of infinite-state systems by temporal deduction.

Our presentation takes the automata-theoretic approach to program verification. This approach reduces the verification problem to the emptiness problem of Büchi automata. The approach was first developed for finite state programs [VW86], then augmented to deal with infinite-state programs in [Var91].

We represent the verified system by a *fair discrete system* (FDS) which is an infinite-state Streett automaton. The negated LTL property is represented by a *tester* which is an infinite-state multi-Büchi automaton. We form the synchronous composition of the two automata and transform the resulting FDS into a *Büchi discrete system* (BDS), which is an infinite-state Büchi automaton $\mathcal{A}$. Following the automata theoretic approach, we have to prove the emptiness of $\mathcal{A}$. Using abstraction, we transform the problem of checking the emptiness of an infinite-state automaton ($\mathcal{A}$) into the emptiness problem of a finite automaton. Our completeness result means that every infinite state Büchi automata can be abstracted into a finite state automaton, with a weak preservation of the emptyness property. Equivalently, every LTL property on an infinite-state program, can be transformed by augmented abstraction into the problem of emptiness of a finite automaton.

The idea of using abstraction for simplifying the task of verification is certainly not new with us. Even the observation that, in many interesting cases, infinite-state systems can be abstracted into finite-state systems which can be model checked has been made before. The main contributions of the paper can be summarized as

- Observing that for some verification tasks involving liveness, pure abstraction is inadequate, and devising the method of *verification by augmented abstraction* (VAA).

- Establishing a (remarkably simple) deductive rule for proving emptiness of a simple (Büchi) FDS, which is an infinite-state automata with Büchi acceptance conditions.
- Establishing completeness of the VAA method.

1.1 Related Work

There has been an extensive study of the use of data abstraction techniques, mostly based on the notions of *abstract interpretation* (a partial list [CC77], [CH78], [CGL94], [CGL96] [DGG97]). Most of the previous work was done in a branching context which complicates the problem if one wishes to preserve both existential and universal properties. None of these articles considers explicitly the question of fairness requirements and how they are affected by the abstraction process. Approaches based on simulation and studies of the properties they preserve are considered in [BBLS92] and [GL93]. A linear-time application of abstract interpretation is proposed in [BBM95], applying the abstractions directly to the computational model of *fair transition systems* which is very close to the FDS model considered here. However, the method is only applied for the verification of safety properties. Liveness, and therefore fairness, are not considered.

In [MP91a], a deductive methodology for proving temporal properties over infinite state system is presented. This methodology, based on a set of proof rules, is proved to be complete, relative to the underlying assertion language. This proof rules and the completeness proof are based on the FTS computation model [MP91b]. The translation of the rules and completeness proof to the fair discrete system (FDS) model is presented in [KP98].

Verification diagrams, presented in [MP94], provide a graphical representation of the deductive proof rules, summarizing the necessary verification conditions. A verification diagram is a finite graph, which can be viewed as a finite abstraction of the verified system, with respect to the verified property. In [BMS95], [MBSU98], the notion of a verification diagram is generalized, allowing a uniform verification of arbitrary temporal formulas. The GVD can be viewed as an abstraction of the verified system which is justified deductively and verified by model checking. The GVD method is also shown to be sound and complete. The abstraction constructed by this method is based on the FTS computation model, and can be viewed as an ω-automaton with either Street ([BMS95]) or Muller ([MBSU98]) acceptance condition. A dual method to VD and GVD is the deductive model checking (DMC) presented in [SUM96]. Similar to VD and GVD, this method tries to verify a temporal property over an infinite state system, using a finite graph representation. The method is demonstrated to terminate on many infinite state systems, but is not shown to be complete. An (LTL-based) general approach to abstraction has been independently developed in [Uri99].

An important development in the theory and implementation of verification by finitary (and other types of) abstraction is reported in [BLO98]. The paper describes the support system InVest, which employs various heuristics for the automatic generation of finitary abstractions for a given system. InVest has managed to compute automatically the abstraction presented in our example Fig. 9.

2 A Computational Model: Fair Discrete Systems

As a computational model for reactive systems, we take the model of a *fair discrete system* (FDS). An FDS $\mathcal{D}: \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ consists of the following components.

- $V = \{u_1, ..., u_n\}$: A finite set of typed *system variables*, containing data and control variables. The set of *states* (interpretation) over V is denoted by Σ.
- Θ : The *initial condition* – an *assertion* characterizing the initial states.
- ρ : A *transition relation* – an assertion $\rho(V, V')$, relating the values V of the variables in state $s \in \Sigma$ to the values V' in a $\mathcal{D}$-successor state $s' \in \Sigma$.
- $\mathcal{J} = \{J_1, \ldots, J_k\}$: A set of *justice* requirements (*weak fairness*). The justice requirement $J \in \mathcal{J}$ is an assertion, intended to guarantee that every computation contains infinitely many J-states (states satisfying J).
- $\mathcal{C} = \{\langle p_1, q_1 \rangle, \ldots \langle p_n, q_n \rangle\}$: A set of *compassion* requirements (*strong fairness*). The compassion requirement $\langle p, q \rangle \in \mathcal{C}$ is a pair of assertions, intended to guarantee that every computation containing infinitely many p-states also contains infinitely many q-states.

We require that every state $s \in \Sigma$ has at least one $\mathcal{D}$-successor. A *computation* of an FDS $\mathcal{D}$ is an infinite sequence of states $\sigma : s_0, s_1, s_2, ...$, satisfying the requirements:

- *Initiality:* s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution:* For each $j = 0, 1, ...$, the state s_{j+1} is a $\mathcal{D}$-successor of s_j.
- *Justice:* For each $J \in \mathcal{J}$, σ contains infinitely many J-positions
- *Compassion:* For each $\langle p, q \rangle \in \mathcal{C}$, if σ contains infinitely many p-positions, it must also contain infinitely many q-positions.

We denote by $Comp(\mathcal{D})$ the set of all computations of $\mathcal{D}$. An FDS $\mathcal{D}$ is called *feasible* if $Comp(\mathcal{D}) \neq \emptyset$. The feasibility of a finite-state FDS can be checked algorithmically, using symbolic model checking, as presented in [KPR98]. A state is called $\mathcal{D}$-*reachable* if it appears in some computation of $\mathcal{D}$.

Let $U \subseteq V$ be a set of variables. Let σ be an infinite sequence of states. We denote by $\sigma \Downarrow_U$ the *projection* of σ onto the subset U. We denote by $Comp(\mathcal{D}) \Downarrow_U$ the set of computations of $\mathcal{D}$, projected onto the set of variables U. Let $\mathcal{D}_1: \langle V_1, \Theta_1, \rho_1, \mathcal{J}_1, \mathcal{C}_1 \rangle$ and $\mathcal{D}_2: \langle V_2, \Theta_2, \rho_2, \mathcal{J}_2, \mathcal{C}_2 \rangle$ be two FDS's and $U \subseteq V_1 \cap V_2$. We say that D_1 is U-*equivalent* to $\mathcal{D}_2$ ($\mathcal{D}_1 \sim_U \mathcal{D}_2$) if $Comp(\mathcal{D}_1) \Downarrow_U = Comp(\mathcal{D}_2) \Downarrow_U$.

All our concrete examples are given in SPL (Simple Programming Language), which is used to represent concurrent programs (e.g., [MP95], [MAB+94]). Every SPL program can be compiled into an FDS in a straightforward manner (see [KPR98]). The predicates at_ℓ_0 and $at\_\ell'_1$ stand, respectively, for the assertions $\pi_i = 0$ and $\pi'_i = 1$, where π_i is the control variable denoting the current location within the process to which the statement belongs.

2.1 Synchronous Parallel Composition

Let $\mathcal{D}_1 : \langle V_1, \Theta_1, \rho_1, \mathcal{J}_1, \mathcal{C}_1 \rangle$ and $\mathcal{D}_2 : \langle V_2, \Theta_2, \rho_2, \mathcal{J}_2, \mathcal{C}_2 \rangle$ be two fair discrete systems. We define the *synchronous parallel composition* of $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 \,|||\, \mathcal{D}_2$, to be the system $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where,

$$V \;=\; V_1 \cup V_2 \qquad \Theta \;=\; \Theta_1 \wedge \Theta_2 \qquad \rho = \rho_1 \;\wedge\; \rho_2 \qquad \mathcal{J} = \mathcal{J}_1 \cup \mathcal{J}_2 \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$$

As implied by the definition, each of the basic actions of system $\mathcal{D}$ consists of the joint execution of an action of $\mathcal{D}_1$ and an action of $\mathcal{D}_2$. We can view the execution of $\mathcal{D}$ as the *joint execution* of $\mathcal{D}_1$ and $\mathcal{D}_2$. The main use of the synchronous parallel composition is for coupling a system with a *tester* which tests for the satisfaction of a temporal formula, and then checking the feasibility of the combined system. In this work, synchronous composition is also used for coupling the system with a *monitor*, used to ensure completeness of the data abstraction methodology.

2.2 From FDS to JDS

An FDS with no compassion requirements is called a *just discrete system* (JDS). Let $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ be an FDS such that $\mathcal{C} = \{(p_1, q_1), \ldots, (p_m, q_m)\}$ and $m > 0$. We define a JDS $\mathcal{D}' : \langle V', \Theta', \rho', \mathcal{J}', \mathcal{C}' : \emptyset \rangle$ which is V-equivalent to $\mathcal{D}$ as follows. First we construct m similar JDS's, $\mathcal{D}_1, \ldots, \mathcal{D}_m$, one for each compassion requirement $(p_i, q_i) \in \mathcal{C}$. The JDS $\mathcal{D}_i$ representing a compassion requirement (p_i, q_i), is presented in Fig. 1.

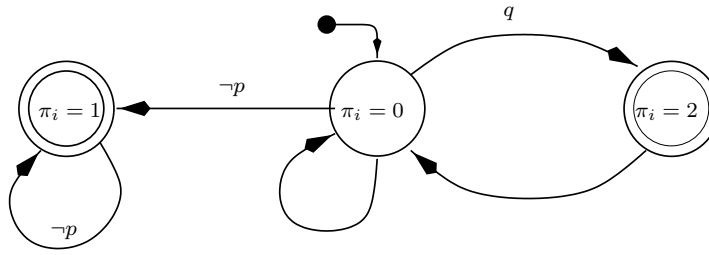


Fig. 1. A JDS $\mathcal{D}_i$ for a single compassion requirement $(p_i, q_i) \in \mathcal{C}$

Each $\mathcal{D}_i$ consists of the components $V_i = \{\pi_i : [0..2]\}$, initial condition $\Theta_i : (\pi_i = 0)$, a single justice requirement $J_i : (\pi_i > 0)$ and no compassion requirements. The JDS $\mathcal{D}'$ is given by $\mathcal{D}' : \quad \mathcal{D} ||| \mathcal{D}_1 ||| \ldots ||| \mathcal{D}_m$.

2.3 From JDS to BDS

A JDS with a single justice requirement is called a *Büchi discrete system* (BDS). Let $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} : \emptyset \rangle$ be a JDS such that $\mathcal{J} = \{J_1, \ldots, J_k\}$ and $k > 1$. We define a BDS $\mathcal{B} : \langle V_\mathcal{B}, \Theta_\mathcal{B}, \rho_\mathcal{B}, \mathcal{J}_\mathcal{B} : \{J\}, \mathcal{C}_\mathcal{B} : \emptyset \rangle$ which is V-equivalent to $\mathcal{D}$:

- $V_\mathcal{B} = V \cup \{u\}$, where u is a new variable not in V, interpreted over $[0..k]$.
- $\Theta_\mathcal{B} : \quad u = 0 \;\wedge\; \Theta$.

- $\rho_{\mathcal{B}}: \rho(V,V') \wedge \bigvee_{i=0}^{k} (u=i) \wedge u' = \left[\begin{array}{ll} \textbf{case} & \\ u=0 & : 1 \\ u=i>0 \ \wedge \ J'_i & : (u+1) \ mod \ (k+1) \\ true & : u \\ \textbf{esac} & \end{array}\right]$
- $\mathcal{J}_{\mathcal{B}} = \{J\}$, where J is the single justice requirement $J : (u = 0)$.

3 Requirement Specification Language: Temporal Logic

As a requirement specification language for reactive systems we take *linear temporal logic* (LTL) [MP91b]. For simplicity, we consider only the future fragment of the logic. Extending the approach to the full logic is straightforward.

We assume an underlying assertion language $\mathcal{L}$ which contains the predicate calculus and interpreted symbols for expressing the standard operations and relations over some concrete domains. A *temporal formula* is constructed out of state formulas (assertions) to which we apply the boolean operators $\neg$ and $\vee$, and the basic temporal operators $\bigcirc$ (*next*) and $\mathcal{U}$ (*until*). A *model* for a temporal formula p is an infinite sequence of states $\sigma : s_0, s_1, ...$, where each state s_j provides an interpretation for the variables in p. We refer the reader to [MP91b, MP95] for the semantics of temporal formulas.

Given a model σ we denote by $(\sigma, j) \models p$ the notion of a temporal formula p holding at a position $j \geq 0$ in σ. If $(\sigma, 0) \models p$, we say that p *holds* on σ, and denote it by $\sigma \models p$. A formula p is called *satisfiable* if it holds on some model. A formula p is called *valid* ($\models p$), if it holds on all models. Two formulas p and q are *equivalent* ($p \sim q$,) if $p \leftrightarrow q$ is valid. Given an FDS $\mathcal{D}$ and a temporal formula p, we say that p is $\mathcal{D}$-*valid* ($\mathcal{D} \models p$,) if p holds on all models which are computations of $\mathcal{D}$. We say that a temporal formula p is *finitary* if the vocabulary V of p is finite and, for each variable $v \in V$, v ranges over a finite domain.

Testers for Temporal Formulas

Given an LTL formula φ, we construct a *tester* T_φ which is a JDS characterizing the set of all sequences satisfying φ. The variables of T_φ consist of the vocabulary of φ plus a set of auxiliary boolean variables

$$X_\varphi : \quad \{x_p \mid p \in \varphi \text{ a principally temporal sub-formula of } \varphi\}$$

We refer the reader to [KPR98] for the construction of testers. This construction was inspired by [CGH94], which is based on the non-symbolic constructions [LP85], [VW86].

4 Reducing Verification to Infeasibility

Let $\mathcal{D}$ be an FDS and ψ be a temporal property. The verification problem $\mathcal{D} \stackrel{?}{\models} \psi$ can be reduced to an infeasibility problem, as follows:

- Construct a tester $T_{\neg\psi}$ for the negated property $\neg\psi$.
- Construct the synchronous parallel composition $\mathcal{D} \| T_{\neg\psi}$.
- Transform the FDS $\mathcal{D} \| T_{\neg\psi}$ into an equivalent BDS $\mathcal{B}_{(\mathcal{D},\neg\psi)}$.

Claim 1. $\mathcal{D} \models \psi$ iff $Comp(\mathcal{B}_{(\mathcal{D},\neg\psi)}) = \emptyset$, i.e., $\mathcal{B}_{(\mathcal{D},\neg\psi)}$ is infeasible [Var91].

5 Infeasibility of a BDS: A Deductive Verification

The standard approach for proving infeasibility of a BDS $\mathcal{B}: \langle V, \Theta, \rho, \mathcal{J} : \{J\}, \mathcal{C} : \emptyset \rangle$, is to define a ranking function δ which maps the reachable states of $\mathcal{B}$ into a well founded domain. The ranking function is required to satisfy the conditions that every transition of $\mathcal{B}$ does not increase the rank and every transition into a state satisfying J, the single justice requirement of $\mathcal{B}$, decreases the rank. The (possibly infinite) set of reachable states of $\mathcal{B}$ can be characterized (over-approximated) by an inductive assertion φ. In Fig. 2 we present rule WELL, a deductive rule that can be used to establish infeasibility of a BDS $\mathcal{B}$.

For an assertion φ,
a single justice requirement J,
a well founded domain $(\mathcal{W}, \prec)$,
and a ranking function $\delta : \Sigma_V \mapsto \mathcal{W}$

$$\begin{array}{ll} \text{W1. } \Theta & \rightarrow \varphi \\ \text{W2. } \rho \wedge \varphi & \rightarrow \varphi' \wedge \delta' \preceq \delta \\ \text{W3. } \rho \wedge \varphi \wedge J' & \rightarrow \varphi' \wedge \delta' \prec \delta \end{array}$$

$$\mathcal{C}omp(\mathcal{B}) = \emptyset$$

Fig. 2. Rule WELL.

Rule WELL is both sound and (relatively) complete. Soundness of the rule means that, given a BDS $\mathcal{B}$, if we can find a ranking function δ and an assertion φ, such that φ and δ satisfy the three premises W1–W3, then $\mathcal{B}$ is indeed infeasible. To see this, assume, to the contrary, that $\mathcal{B}$ is feasible. Then $\mathcal{B}$ has an infinite computation $\sigma: s_0, s_1, \ldots$, such that for infinitely many states s_i in σ, $s_i \models J$. Then, from premises W2 and W3, there exists an infinite sequence of states over which the ranking function δ decreases and never increases in any other step. Since δ is defined over a well-founded domain, this is clearly impossible. The completeness of rule WELL is stated in the following claim:

Claim 2. Let $\mathcal{B}: \langle V, \Theta, \rho, \mathcal{J} : \{J\}, \mathcal{C} : \emptyset \rangle$ be a BDS. If $\mathcal{B}$ is infeasible, then there exist an assertion φ, a well founded domain $(\mathcal{W}, \prec)$ and a ranking function $\delta: \Sigma_V \mapsto \mathcal{W}$ satisfying the premises of rule WELL.

Proof (sketch)*:* To prove the claim, we have to find both an assertion φ and a ranking function δ which satisfy the premises W1–W3 of rule WELL. The proof of existence of an assertion φ characterizing the set of all reachable states of a BDS is presented in [MP91b] (Section 2.5). The existence of a well founded domain $(\mathcal{W}, \prec)$ and ranking function δ satisfying the premises W1–W3, is shown in [Var91], based on [LPS81] and [GFMdR85]. □

Let $\mathcal{D}$ be an FDS and ψ be a temporal property such that $\mathcal{D} \models \psi$. Based on claims 1 and 2, we can identify an assertion and a ranking function, which satisfy the three premises of rule WELL for the BDS $\mathcal{B}_{(\mathcal{D}, \neg\psi)}$. We denote these assertion and ranking function by Φ and Δ, respectively.

6 Finitary Abstraction of a BDS

In this section, we present a general methodology for *data abstraction* of a BDS, derived from the notion of abstract interpretation [CC77]. For more details see [KP98b]. Let $\mathcal{B} = \langle V, \Theta, \rho, \mathcal{J}: \{J\}, \mathcal{C}: \emptyset \rangle$ be a BDS, and Σ denote the set of states of $\mathcal{B}$, the *concrete states*. Let $\alpha : \Sigma \mapsto \Sigma_{_A}$ be a mapping of concrete states into *abstract states*. We say that α is a *finitary abstraction mapping*, if $\Sigma_{_A}$ is a finite set. To provide a syntactic representation of the abstraction mapping, we assume a set of *abstract variables* $V_{_A}$ and a set of expressions $\mathcal{E}^\alpha$, such that the equality $V_{_A} = \mathcal{E}^\alpha(V)$ syntactically represents the semantic mapping α. Let $p(V)$ be an assertion. We define the operator α^+, as follows

$$\alpha^+(p(V)): \quad \exists V \Big(V_{_A} = \mathcal{E}^\alpha(V) \quad \wedge \quad p(V) \Big).$$

The assertion $\alpha^+(p)$ holds for an abstract state $S \in \Sigma_{_A}$ iff the assertion p holds for *some* concrete state $s \in \Sigma$ such that $s \in \alpha^{-1}(S)$. This can also be expressed by the inclusion $\|p\| \subseteq \alpha^{-1}(\|\alpha^+(p)\|)$. Let $\mathcal{B} = \langle V, \Theta, \rho, \mathcal{J} = \{J\}, \mathcal{C} = \emptyset \rangle$ be a BDS. We define $\mathcal{B}^\alpha = \langle V_{_A}, \Theta^\alpha, \rho^\alpha, \mathcal{J}^\alpha, \mathcal{C}^\alpha \rangle$, the α-abstracted BDS, as follows:

$$\Theta^\alpha = \alpha^+(\Theta) \qquad \rho^\alpha = \alpha^{++}(\rho) \qquad \mathcal{J}^\alpha = \{\alpha^+(J)\} \qquad \mathcal{C}^\alpha = \emptyset$$

where $\quad \alpha^{++}(\rho): \quad \exists V, V' (V_{_A} = \mathcal{E}^\alpha(V) \ \wedge \ V'_{_A} = \mathcal{E}^\alpha(V') \ \wedge \ \rho(V, V'))$.

Claim 3 (Weak Preservation). $Comp(\mathcal{B}^\alpha) = \emptyset \quad$ implies $\quad Comp(\mathcal{B}) = \emptyset$.

As an example, we consider program BAKERY-2, presented in Fig. 3.

local y_1, y_2 : **natural where** $y_1 = y_2 = 0$

$$\left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \left[\begin{array}{l} \ell_1 : \textbf{NonCritical} \\ \ell_2 : y_1 := y_2 + 1 \\ \ell_3 : \textbf{await } y_2 = 0 \ \vee \ y_1 < y_2 \\ \ell_4 : \textbf{Critical} \\ \ell_5 : y_1 := 0 \end{array}\right] \end{array}\right] \ || \ \left[\begin{array}{l} m_0 : \textbf{loop forever do} \\ \left[\begin{array}{l} m_1 : \textbf{NonCritical} \\ m_2 : y_2 := y_1 + 1 \\ m_3 : \textbf{await } y_1 = 0 \ \vee \ y_2 \leq y_1 \\ m_4 : \textbf{Critical} \\ m_5 : y_2 := 0 \end{array}\right] \end{array}\right]$$

$- P_1 -$ $\qquad\qquad$ $- P_2 -$

Fig. 3. Program BAKERY-2: the Bakery algorithm for two processes.

Program BAKERY-2 is obviously an infinite-state system, since y_1 and y_2 can assume arbitrarily large values. The temporal properties we wish to establish are $\psi_{exc} : \Box \neg (at\_\ell_4 \ \wedge \ at\_m_4)$ and $\psi_{acc} : \Box (at\_\ell_2 \quad \rightarrow \quad \Diamond \, at\_\ell_4)$. The safety property ψ_{exc} requires *mutual exclusion*, guaranteeing that the two processes never co-reside in their respective critical section at the same time. The liveness property ψ_{acc} requires *accessibility* for process P_1, guaranteeing that, whenever P_1 reaches location ℓ_2 it will eventually reach location ℓ_4.

Following [BBM95], we define abstract boolean variables $B_{p_1}, B_{p_2}, \ldots, B_{p_k}$, one for each atomic data formula, where the atomic data formulas for BAKERY-2

are $y_1 = 0$, $y_2 = 0$, and $y_1 < y_2$. The abstract system variables consist of the concrete control variables, which are left unchanged, and a set of abstract boolean variables $B_{p_1}, B_{p_2}, \ldots, B_{p_k}$. The abstraction mapping α is defined by

$$\alpha: \quad \{B_{p_1} = p_1, B_{p_2} = p_2, \ldots, B_{p_k} = p_k\}$$

That is, the boolean variable B_{p_i} has the value *true* in the abstract state iff the assertion p_i holds at the corresponding concrete state. It is straightforward to compute the α-induced abstractions of the initial condition Θ^α and the transition relation ρ^α. In Fig. 4, we present program BAKERY-2 (with a capital B), the α-induced abstraction of program BAKERY-2.

$$\textbf{local } B_{y_1=0}, B_{y_2=0}, B_{y_1<y_2} : \textbf{boolean initially } B_{y_1=0} = B_{y_2=0} = 1, B_{y_1<y_2} = 0$$

$$\left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \left[\begin{array}{l} \ell_1 : \textbf{NonCritical} \\ \ell_2 : (B_{y_1=0}, B_{y_1<y_2}) := (0,0) \\ \ell_3 : \textbf{await } B_{y_2=0} \ \vee \ B_{y_1<y_2} \\ \ell_4 : \textbf{Critical} \\ \ell_5 : (B_{y_1=0}, B_{y_1<y_2}) := (1, \neg B_{y_2=0}) \end{array}\right] \end{array}\right] \ \| \ \left[\begin{array}{l} m_0 : \textbf{loop forever do} \\ \left[\begin{array}{l} m_1 : \textbf{NonCritical} \\ m_2 : (B_{y_2=0}, B_{y_1<y_2}) := (0,1) \\ m_3 : \textbf{await } B_{y_1=0} \ \vee \ \neg B_{y_1<y_2} \\ m_4 : \textbf{Critical} \\ m_5 : (B_{y_2=0}, B_{y_1<y_2}) := (1,0) \end{array}\right] \end{array}\right]$$

$$- P_1 - \qquad\qquad - P_2 -$$

Fig. 4. Program BAKERY-2: the Bakery algorithm for two processes.

Since the properties we wish to verify refer only to the control variables (through the at_ℓ and at_m expressions), they are not affected by the abstraction. Program BAKERY-2 is a finite-state program, and we can apply model checking to verify that it satisfies the two properties of mutual exclusion and accessibility. By Claim 3, we can infer that the original program BAKERY-2 also satisfies these two temporal properties.

6.1 Augmentation by Progress Monitors

Program BAKERY-2 is an example of successful data abstraction. However, there are cases where abstraction alone is inadequate for transforming an infinite-state system satisfying a property into a finite-state abstraction which maintains the property. In the following we illustrate the problem and the proposed solution.

In Fig. 5, we present a simple looping program. The assignment at statement ℓ_2 assigns to y non-deterministically the values $y+1$ or y. The property we wish to verify is that program SUB-ADD always terminates.
A natural abstraction for the variable y is defined by

$$Y = \textbf{if } y = 0 \textbf{ then } \mathit{zero} \textbf{ else if } y = 1 \textbf{ then } \mathit{one} \textbf{ else } \mathit{large},$$

where y is abstracted into the three-valued domain $\{\mathit{zero}, \mathit{one}, \mathit{large}\}$. However, applying this abstraction yields the abstract program SUB-ADD-ABS-1, presented in Fig. 6, where the abstract functions *sub2* and *add1* are defined by

$$\begin{array}{lcl} \mathit{sub2}(Y) & = & \textbf{if } Y = \{\mathit{zero}, \mathit{one}\} \textbf{ then } \mathit{zero} \textbf{ else } \{\mathit{zero}, \mathit{one}, \mathit{large}\}, \\ \mathit{add1}(Y) & = & \textbf{if } Y = \mathit{zero} \textbf{ then } \mathit{one} \textbf{ else } \mathit{large}. \end{array}$$

$$
\begin{array}{l}
y\colon \textbf{natural} \\
\ell_0 : \textbf{while } y > 1 \textbf{ do} \\
\quad \left[\begin{array}{l} \ell_1 : y := y - 2 \\ \ell_2 : y := \{y+1, y\} \\ \ell_3 : \textbf{skip} \end{array}\right] \\
\ell_4 :
\end{array}
$$

Fig. 5. Program SUB-ADD .

$$
\begin{array}{l}
Y : \{\mathit{zero}, \mathit{one}, \mathit{large}\} \\
\ell_0 : \textbf{while } Y = \mathit{large} \textbf{ do} \\
\quad \left[\begin{array}{ll} \ell_1 : & Y := \mathit{sub2}(Y) \\ \ell_2 : & Y := \{\mathit{add1}(Y), Y\} \\ \ell_3 : \textbf{skip} & \end{array}\right] \\
\ell_4 :
\end{array}
$$

Fig. 6. Program SUB-ADD-ABS-1 abstracting program SUB-ADD.

Unfortunately, program SUB-ADD-ABS-1 need not terminate, because the function *sub2* can always choose to yield *large* as a result. Termination of programs like program SUB-ADD can always be established by identification of a *progress measure* that never increases and sometimes is guaranteed to decrease. In this case, for example, we can use the progress measure $\delta : y + at\_\ell_2$ which never increases and always decreases on the execution of statement ℓ_1. To obtain a working abstraction, we first compose program SUB-ADD with an additional module, called the *progress monitor* for the measure δ, as shown in Fig. 7.

$$
\begin{array}{c}
y\colon \textbf{natural} \\
\left[\begin{array}{l}
\ell_0 : \textbf{while } y > 1 \textbf{ do} \\
\quad \left[\begin{array}{l} \ell_1 : y := y - 2 \\ \ell_2 : y := \{y+1, y\} \\ \ell_3 : \textbf{skip} \end{array}\right] \\
\ell_4 :
\end{array}\right]
\;\;|||\;\;
\left[\begin{array}{l}
\textbf{define } \delta = y + at\_\ell_2 \\
\mathit{inc} \quad : \{-1, 0, 1\} \\
m_0 : \textbf{always do} \\
\qquad \mathit{inc} := \mathit{diff}(\delta, \delta')
\end{array}\right] \\
\begin{array}{cc}
- \text{ SUB-ADD } - & - \text{ MONITOR } M_\delta -
\end{array}
\end{array}
$$

Fig. 7. Program SUB-ADD composed with a monitor.

The construct **always do** appearing in MONITOR M_δ means that the assignment which is the body of this construct is executed at *every* step. The comparison function $\mathit{diff}(\delta, \delta')$ is defined by

$$\mathit{diff}(\delta, \delta') = \textbf{if } \delta < \delta' \textbf{ then } 1 \textbf{ else if } \delta = \delta' \textbf{ then } 0 \textbf{ else } -1.$$

The presentation of the monitor module M_δ in Fig. 7 is only for illustration purposes. The precise definition of this module is given by the following FDS:

$V : \{V_D, inc : \{-1, 0, 1\}\}$ $\quad \Theta : \mathrm{T}$
$\rho : inc' = diff(\delta, \delta')$ $\quad \mathcal{J} : \emptyset$ $\quad \mathcal{C} : \{(inc < 0, inc > 0)\}$

Thus, at every step of the computation, module M_δ compares the new value of δ with the current value, and sets variable *inc* to -1, 0, or 1, according to whether the value of δ has decreased, stayed the same, or increased, respectively. This FDS has no justice requirements but has the single compassion requirement $(inc < 0, inc > 0)$ stating that δ cannot decrease infinitely many times without also increasing infinitely many times. This requirement is a direct consequence of the fact that δ ranges over the well-founded domain of the natural numbers, which does not allow an infinitely descreasing sequence.

It is possible to represent this composition as (almost) equivalent to the sequential program presented in Fig. 8, where we have conjoined the repeated assignment of module M_δ with every assignment of process SUB-ADD.

$$
\begin{array}{l}
\qquad\qquad\quad y \;\; : \textbf{natural} \\
\qquad\qquad\quad inc : \{-1, 0, 1\} \\
\ell_0 : \textbf{while } y > 0 \textbf{ do} \\
\qquad \left[\begin{array}{lrl}
\ell_1 : (y, inc) & := & (y - 2,\ diff(\delta, \delta')) \\
\ell_2 : (y, inc) & := & (\{y + 1, y\},\ diff(\delta, \delta')) \\
\ell_3 : \quad inc & := & diff(\delta, \delta')
\end{array}\right] \\
\ell_4 :
\end{array}
$$

Fig. 8. A sequential equivalent of the monitored program.

The abstraction of the program of Fig. 8 will abstract y into a variable Y ranging over $\{zero, one, large\}$. The variable *inc* is not abstracted. The resulting abstraction is presented in Fig. 9.

$$
\begin{array}{l}
\qquad Y \qquad\qquad : \{zero, one, large\} \\
\qquad inc \qquad\quad : \{-1, 0, 1\} \\
\qquad \textbf{compassion } (inc < 0, inc > 0) \\
\ell_0 : \textbf{while } Y = large \textbf{ do} \\
\qquad \left[\begin{array}{lrl}
\ell_1 : (Y, inc) & := & (sub2(Y),\ -1) \\
\ell_2 : (Y, inc) & := & (\{add1(Y), Y\},\ \{0, -1\}) \\
\ell_3 : \quad inc & := & 0
\end{array}\right] \\
\ell_4 :
\end{array}
$$

Fig. 9. Abstracted version of the monitored- Program SUB-ADD-ABS-2.

The program SUB-ADD-ABS-2 (Fig. 9) differs from program SUB-ADD-ABS-1 (Fig 6) by the additional compassion requirement $(inc < 0, inc > 0)$. It is this additional requirement which forces program SUB-ADD-ABS-2 to terminate. This

is because a run in which *sub1* always yields *large* as a result is a run in which *inc* is negative infinitely many times (on every visit to ℓ_1) and is never positive beyond the first state. The fact that SUB-ADD-ABS-2 always terminates can now be successfully model-checked.

The extension to the case that the progress measure ranges not over the naturals but over lexicographic tuples of naturals is straightforward.

6.2 The General Structure of a Progress Monitor

We proceed to define the general structure of a progress monitor and show that its augmentation to a verified system is safe. A *well-founded domain* $(\mathcal{W}, \prec)$ consists of a set $\mathcal{W}$ and a total ordering relation $\prec$ over $\mathcal{W}$ such that there does not exist an infinitely descending sequence, i.e., a sequence of the form

$$a_0 \succ a_1 \succ a_2 \succ \cdots ,$$

A *ranking function* for an FDS $\mathcal{D}$ is a function δ mapping the states of $\mathcal{D}$ into a well-founded domain. A *progress monitor* for a ranking function δ is an FDS M_δ of the following form:

$$M_\delta = \left\langle \begin{array}{ll} V : V_{\mathcal{D}}, inc : \{-1, 0, 1\}\}, & \Theta : true, \\ \rho : \ inc' = diff\,(\delta(V_{\mathcal{D}}), \delta(V'_{\mathcal{D}})), & \mathcal{J} : \emptyset, \quad \mathcal{C} : \{(inc < 0, inc > 0)\} \end{array} \right\rangle$$

The following claim states that augmentation to a verified system is safe:

Claim 4. $\quad Comp(\mathcal{D} \| M_\delta) \Downarrow_{V_{\mathcal{D}}} = Comp(\mathcal{D})$

7 Verification by Augmented Finitary Abstraction

Let $\mathcal{B}$ be an infeasible BDS. Let α be an abstraction mapping and δ be a ranking function for $\mathcal{B}$. We say that $\langle \alpha, \delta \rangle$ is an *adequate augmented abstraction* for $\mathcal{B}$ if α is finitary and $Comp((\mathcal{B} \| M_\delta)^\alpha) = \emptyset$.

Let $\mathcal{D}$ be an FDS, ψ be a temporal property such that $\mathcal{D} \models \psi$, and α be a finitary abstraction mapping. Let Δ be a ranking function for $\mathcal{B}_{(\mathcal{D},\neg\psi)}$. From the definition of adequate augmented abstraction and Claim 1, we can say that $\langle \alpha, \Delta \rangle$ is an adequate augmented abstraction for $\langle \mathcal{D}, \psi \rangle$ iff it is an adequate augmented abstraction for $\mathcal{B}_{(\mathcal{D},\neg\psi)}$, i.e., $Comp((\mathcal{B}_{(\mathcal{D},\neg\psi)} \| M_\Delta)^\alpha) = \emptyset$.

We can now formulate the method of *verification by augmented finitary abstraction* (VAA) as follows. To verify that ψ is $\mathcal{D}$-valid,

- Construct a tester $T_{\neg\psi}$ for the negated temporal property ψ.
- Construct the synchronous parallel composition $\mathcal{D} \| T_{\neg\psi}$ of the FDS $\mathcal{D}$ representing the verified system and the tester $T_{\neg\psi}$, and transform it into an equivalent BDS $\mathcal{B}_{(\mathcal{D},\neg\psi)}$.
- Identify an appropriate ranking function Δ for $\mathcal{B}_{(\mathcal{D},\neg\psi)}$, and construct the progress monitor M_Δ.
- Construct an FDS of the augmented system $\mathcal{A} : \quad \mathcal{B}_{(\mathcal{D},\neg\psi)} \| M_\Delta$.
- Abstract the augmented system $\mathcal{A}$ into a *finitary abstract* FDS $\mathcal{A}^\alpha$.
- Model-check $Comp(\mathcal{A}^\alpha) = \emptyset$.
- Infer $\mathcal{D} \models \psi$.

Claim 5 (Soundness). $\quad Comp(\mathcal{B}_{(\mathcal{D},\neg\psi)} \| M_\Delta^\alpha) = \emptyset$ implies $\mathcal{D} \models \psi$

8 Completeness of the VAA Method

In the following we prove the completeness of the VAA method. First we introduce the operator α^-, dual to α^+, and establish some useful properties of the abstraction mappings α^+ and α^{++}.

8.1 The α^- Operator

The operator α^-, is dual to α^+. Let $p(V)$ be an assertion. The operator α^- is defined by

$$\alpha^-(p(V)): \quad \forall V \left(V_A = \mathcal{E}^\alpha(V) \quad \rightarrow \quad p(V)\right).$$

The assertion $\alpha^-(p)$ holds for an abstract state $S \in \Sigma_A$ iff the assertion p holds for *all* concrete states $s \in \Sigma$ such that $s \in \alpha^{-1}(S)$. This can also be expressed by the inclusion $\alpha^{-1}(\|\alpha^-(p)\|) \subseteq \|p\|$, where $\|p\|$ and $\|\alpha^-(p)\|$ represent the sets of states which satisfy the assertions, respectively. If $\alpha^-(p)$ is valid, then $\|\alpha^-(p)\| = \Sigma_A$ implying $\alpha^{-1}(\|\alpha^-(p)\|) = \Sigma$ which, by the above inclusion, leads to $\|p\| = \Sigma$ establishing the validity of p.

An abstraction α is said to be *precise with respect to an assertion* p if $\alpha^+(p) \sim \alpha^-(p)$. A sufficient condition for α to be precise w.r.t. p is that the abstract variables include a boolean variable B_p with the definition $B_p = p$.

8.2 Properties of α^+ and α^{++}

Lemma 1. *If α is precise with respect to the assertions $p_1, \ldots, p_n$, then α is precise with respect to any boolean combination of these assertions.*

Lemma 2. *Let $p = p(V, V')$ and $q = q(V)$ be two assertions, such that α is precise with respect to q. Then, the following equivalences hold*

$$\alpha^{++}(p \ \wedge \ q) \sim \alpha^{++}(p) \ \wedge \ \alpha^+(q) \tag{1}$$

$$\alpha^{++}(p \ \wedge \ q') \sim \alpha^{++}(p) \ \wedge \ \alpha^+(q)' \tag{2}$$

It also follows from the definitions that if $p = p(V)$, then both $\alpha^{++}(p) \sim \alpha^+(p)$ and $\alpha^{++}(p') \sim \alpha^+(p)'$ hold without any precision assumptions about p. Finally, we observe that if an implication is valid, we can apply the abstractions α^+ and α^{++} to both sides of the implication. That is,

$$\models p \ \rightarrow \ q \qquad \text{implies} \qquad \left(\begin{array}{ll} \models & \alpha^+(p) \rightarrow \alpha^+(q) \qquad \text{and} \\ \models & \alpha^{++}(p) \rightarrow \alpha^{++}(q) \end{array} \right) \tag{3}$$

8.3 The Completeness Statement

Claim 6 (Completeness of VAA*).* Let $\mathcal{B}_{(\mathcal{D},\neg\psi)}$ be an infeasible BDS. Then, there exists an adequate augmented abstraction for $\mathcal{B}_{(\mathcal{D},\neg\psi)}$.

As the ranking function for our augmented abstraction we take Δ (see Section 5). Let H denote all the *homogeneous* atomic state sub-formulas of the invariant assertion Φ and any of the components of $\mathcal{A}$, where $\mathcal{A} : \quad \mathcal{B}_{(\mathcal{D},\neg\psi)} \| M_\Delta$. That is, H contains all the atomic state sub-formulas $f(V)$, such that $f(V)$ or $f(V')$ appears in Φ or any of the components of $\mathcal{A}$. The set H does not include atomic formulas with mixed (primed and unprimed) variables such as $y' = y + 1$.

Let α be a finitary abstraction which is precise with respect to all the assertions appearing in H, and does not abstract the auxiliary variables in X_φ and the variable inc, where $inc : \mathit{diff}(\Delta, \Delta')$. In the following, we show that for this choice of ranking function and abstraction mapping, $\mathit{Comp}(\mathcal{A}^\alpha) = \emptyset$, that is, the abstracted augmented system is indeed infeasible.

Abstracting the Premises of Rule WELL.

The proof is based on the abstraction of premises W1–W3 of rule WELL (Section 5), applied to the BDS $\mathcal{B}_{(\mathcal{D},\neg\psi)}: \langle V, \Theta, \rho, \mathcal{J} = \{J\}, \mathcal{C} = \emptyset\rangle$. These three premises are known to be valid for our choice of Φ and Δ. Since $\mathcal{A} = \mathcal{B}_{(\mathcal{D},\neg\psi)} \| M_\Delta$, then from the definition of M_Δ, the components of $\mathcal{A}$ are given by

$$\Theta_\mathcal{A} : \Theta \qquad \rho_\mathcal{A} : \rho \wedge \underbrace{inc' = \mathit{diff}(\Delta, \delta')}_{\rho_{M_\Delta}} \qquad \mathcal{J}_\mathcal{A} : \mathcal{J} \qquad \mathcal{C}_\mathcal{A} : \{(inc < 0, inc > 0)\}$$

From the implication

$$inc' = \mathit{diff}(\Delta, \Delta') \quad \rightarrow \quad \Big(\Delta' \preceq \Delta \rightarrow inc' \leq 0 \quad \wedge \quad \Delta' \prec \Delta \rightarrow inc' < 0\Big)$$

and the three premises of rule WELL applied to $\mathcal{B}_{(\mathcal{D},\neg\psi)}$, we can obtain the following three valid implications:

$$\begin{array}{ll} \text{U1. } \Theta_\mathcal{A} & \rightarrow \Phi \\ \text{U2. } \rho_\mathcal{A} \wedge \Phi & \rightarrow \Phi' \wedge inc' \leq 0 \\ \text{U3. } \rho_\mathcal{A} \wedge \Phi \wedge J' & \rightarrow \Phi' \wedge inc' < 0. \end{array}$$

Based on Equation (3), we can apply α^+ to both sides of U1 and apply α^{++} to both sides of U2 and U3. We then simplify the right-hand sides, using the fact that $\alpha^{++}(p') \sim \alpha^+(p)'$, and that α does not abstract inc. Next, we use the fact that α is precise w.r.t. all the atomic formulas appearing in Φ and J, in order to distribute the abstraction over the conjunctions on the left-hand sides of the implications. These transformations and simplifications lead to the following three valid abstract implications:

$$\begin{array}{ll} \text{V1. } \alpha(\Theta_\mathcal{A}) & \rightarrow \alpha(\Phi) \\ \text{V2. } \alpha^{++}(\rho_\mathcal{A}) \wedge \alpha(\Phi) & \rightarrow \alpha(\Phi)' \wedge inc' \leq 0 \\ \text{V3. } \alpha^{++}(\rho_\mathcal{A}) \wedge \alpha(\Phi) \wedge \alpha(J)' & \rightarrow \alpha(\Phi)' \wedge inc' < 0. \end{array}$$

The augmented System $\mathcal{A}^\alpha$ has no computations

We proceed to show that $\mathcal{A}^\alpha$ has no computations ($\mathit{Comp}(\mathcal{A}^\alpha) = \emptyset$). Assume to the contrary. Let $\sigma: s_0, s_1, \ldots$ be a computation of $\mathcal{A}$.

First we use the implications V1–V3 to show that the assertion $\alpha(\Phi)$ is an invariant of σ. Since σ is a computation of $\mathcal{A}^\alpha$, the first state of σ satisfies $\alpha(\Theta_\mathcal{A})$ and we conclude by V1 that the first state of σ satisfies $\alpha(\Phi)$. Proceeding from each state s_j of σ to its successor s_{j+1}, which must be an $\alpha^{++}(\rho_\mathcal{A})$-successor of s_j, we see from V2and V3 that $\alpha(\Phi)$ keeps propagating. It follows that $\alpha(\Phi)$ is an invariant of σ, i.e, every state s_i of σ satisfies $\alpha(\Phi)$.

Next, since σ is a computation of $\mathcal{A}^\alpha$, it must contain infinitely many states which satisfy $\alpha(J)$. According to implications V2 and V3, the variable *inc* is never positive, and is negative infinitely many times. Such a behavior contradicts the compassion requirement $(inc < 0, inc > 0)$ associated with $\mathcal{A}^\alpha$. Thus, σ cannot be a computation of $\mathcal{A}^\alpha$, contradicting our initial assumption. This concludes our proof of completeness.

9 Conclusions

We have presented a method for verification by augmented finitary abstraction by which, in order to verify that a (potentially infinite-state) system satisfies a temporal property, one first augments the system with a non-constraining progress monitor and then abstracts the augmented system and the temporal specification into a finite-state verification problem, which can be resolved by model checking. The method has been shown to be sound and complete.

In principle, the established completeness promotes the VAA method to the status of a viable alternative to the verification of infinite-state reactive systems by temporal deduction. Some potential users of formal verification may find the activity of devising good abstraction mappings more tractable (and similar to programming) than the design of auxiliary invariants. However, on a deeper level it is possible to argue that this is only a formal shift and that the same amount of ingenuity and deep understanding of the analyzed system is still required for effective verification as in the practice of temporal deduction methods.

The development of the VAA theory calls for additional research in the implementation of these methods. In particular, there is a strong need for devising heuristics for the automatic generation of effective abstraction mappings and corresponding augmenting monitors.

Acknowledgment: We gratefully acknowledge the many useful discussions and insightful observations by Moshe Vardi which helped us clarify the main issues considered in this paper. We also thank Saddek Bensalem for his helpful comments and for running many of our examples on his automatic abstraction system INVEST.

References

[BBLS92] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Properties preserving simulations. *CAV'92*, *LNCS* 663.

[BBM95] N. Bjørner, I.A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. In 1st *Intl. Conf. on Principles and Practice of Constraint Programming*, *LNCS* 967, 1995.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Abstractions of infinite state systems compositionally and automatically. *CAV'98*, *LNCS* 1427.

[BMS95] I.A. Browne, Z. Manna, and H.B. Sipma. Generalized verification diagrams. In *FSTTSC'95*, *LNCS* 1026.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *POPL'77*.

[CGH94] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *CAV'94*, *LNCS* 818.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Trans. Prog. Lang. Sys.*, 16(5):1512–1542, 1994.

[CGL96] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking. In *Model Checking, Abstraction and Composition*, volume 152 of *Nato ASI Series F*, pages 477–498. Springer-Verlag, 1996.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. *POPL'78*.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Prog. Lang. Sys.*, 19(2), 1997.

[GFMdR85] O. Grumberg, N. Francez, J.A. Makowski, and W.-P. de Roever. A proof rule for fair termination. *Inf. and Comp.*, 66:83–101, 1985.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. *CAV'93*, *LNCS* 697.

[KP98] Y. Kesten and A. Pnueli. Deductive verification of fair discrete systems. Technical report, the Weizmann Institute, 1998.

[KP98b] Y. Kesten and A. Pnueli. Modularization and Abstraction: The Keys to Formal Verification. *MFCS'98*, *LNCS* 1450.

[KPR98] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. *ICALP'98*, *LNCS* 1443.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. *POPL'85*.

[LPS81] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *ICALP'81*, *LNCS* 115.

[MAB+94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, 1994.

[MBSU98] Z. Manna, A. Brown, H. B. Sipma, and T. E. Uribe. Visual abstractions for temporal verification. *AMAST'98*.

[MP91a] Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comp. Sci.*, 83(1):97–130, 1991.

[MP91b] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[MP94] Z. Manna and A. Pnueli. Temporal verification diagrams. In *Theoretical Aspects of Computer Software*, *LNCS* 789, 1994.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[SUM96] H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. *CAV'96*.

[Uri99] T. E. Uribe. *Abstraction-Based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Stanford University, 1999.

[Var91] M. Y. Vardi. Verification of concurrent programs – the automata-theoretic framework. *Annals of Pure Applied Logic*, 51:79–98, 1991.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS'86*.

Signed Interval Logic

Thomas Marthedal Rasmussen*

Department of Information Technology,
Technical University of Denmark, Building 344,
DK-2800 Lyngby, Denmark
tmr@it.dtu.dk

Abstract. Signed Interval Logic (SIL) is an extension of Interval Temporal Logic (ITL) with the introduction of the notion of a *direction* of an interval.
We develop syntax, semantics, and proof system of SIL, and show that this proof system is sound and complete. The proof system of SIL is not more complicated than that of ITL but SIL is (contrary to ITL) capable of specifying *liveness* properties. Other interval logics capable of this (such as Neighbourhood Logic) have more complicated proof systems. We discuss how to define *future intervals* in SIL for the specification of liveness properties.
To characterize the expressive power of SIL we relate SIL to *arrow logic* and *relational algebra*.

Keywords: interval logic, temporal intervals, arrow logic, real-time systems, liveness.

1 Introduction

Interval logics [4,11,17,19,8,5,22,20,12,2,13,14,6,21] are logics of temporal intervals: One can express properties such as "if ϕ holds on this interval then ψ must hold on all subintervals" or "φ must hold on some interval eventually". Interval logics have proven useful in the specification and verification of real-time and safety-critical systems.

In this paper we introduce a new kind of interval logic, called *Signed Interval Logic* (SIL), with the introduction of the notion of a *direction* of an interval. The proof system of SIL turns out to be not more complicated than that of Interval Temporal Logic (ITL) [2] but SIL is (contrary to ITL) capable of specifying *liveness* properties. Other interval logics capable of this (such as Neighbourhood Logic (NL) [21]) have more complicated proof systems.

ITL is one of the most simple interval logics; it has only one *interval modality*, the binary *chop*: $\frown$. The semantics of $\frown$ is given in Fig. 1. In ITL (and NL) intervals are represented by pairs $[b, e]$ (where $b \leq e$) of elements from some totally ordered *temporal domain* of time points.

* Tel: +45 4525 3764, Fax: +45 4593 0074.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 157–171, 1999.

We will refer to m of Fig. 1 as the *chopping point* of $\frown$. The chopping point will always lie *inside* the *current interval* on which we interpret a given formula. In general, modalities with this property are called *contracting*. With contracting modalities it is only possible to specify *safety properties* of a system. This is because once we have chosen the interval we want to observe we are restricted to specifying properties of this interval and its subintervals.

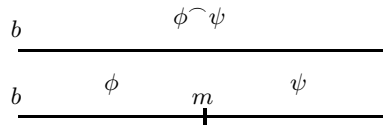


Fig. 1. $\phi \frown \psi$ holds on $[b, e]$ iff there exists $m \in [b, e]$ such that ϕ holds on $[b, m]$ and ψ holds on $[m, e]$

To specify *liveness* properties, we need to reach intervals *outside* the current interval. In general, modalities which can do this are called *expanding*. Neighbourhood Logic (NL) [21] is an example of an interval logic with expanding modalities. NL has two modalities $\Diamond_r$ and $\Diamond_l$ for reaching a *right neighbourhood* and a *left neighbourhood*, respectively, of the current interval. This intuition is made more precise in Fig. 2 in the case of $\Diamond_r$. The case of $\Diamond_l$ is similar.

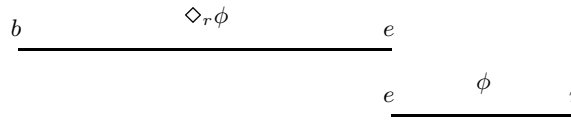


Fig. 2. $\Diamond_r \phi$ holds on $[b, e]$ iff there exists $n \geq e$ such that ϕ holds on $[e, n]$

Both ITL and NL include a special symbol ℓ which intuitively represents the *length* of an interval. This property is not common for all interval logics.

We now turn the attention to the contribution of this paper: SIL is an extension of ITL with the introduction of the notion of a *direction* (which can be either *forward* or *backward*) of an interval. The idea for SIL originates in [3] where an interval logic with such a notion of a direction of an interval was informally developed.

An interval with a direction is in SIL represented by a *signed interval* (b, e). Both the pair (b, e) and the pair (e, b) represent the *same* interval but (e, b) has the opposite direction of (b, e). SIL inherits the special symbol ℓ from ITL. ℓ now gives the *signed length* of an interval. Intuitively, the absolute value of ℓ gives the length of the interval and the sign of ℓ determines the direction.

Like ITL, SIL only has the binary modality $\frown$. But because of the directions of intervals, the semantics is now altered: See Fig. 3. On the figure the direction of an interval is marked with a small arrowhead in either end of the interval. The chopping point can now lie anywhere and not just inside the current interval. This means that $\frown$ of SIL has become an expanding modality, hence SIL can specify liveness properties.

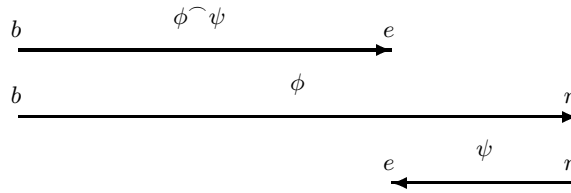


Fig. 3. $\phi^\frown\psi$ holds on (b, e) iff there exists m such that ϕ holds on (b, m) and ψ holds on (m, e)

We will in the following shortly consider related work on interval logic.

In [5] an interval logic with six unary interval modalities is developed, and it is shown that they can express all thirteen possible relations between intervals [1]. In [19] a complete proof system for the interval logic of [5] is given. [20] considers an even more expressive interval logic with three binary modalities instead of six unary. A complete proof system for this logic is also given. Unfortunately, both the proof systems of [19,20] are somewhat complicated due to the fact that they are kept in a propositional setting. In particular, both systems include some complicated inference rules.

NL has also a complete proof system [21]. This proof system is somewhat complicated because of "two-level" axiom schemes: Besides being axiom schemes in terms of formulas, the axioms are also schemes in terms of interval modalities.

It is possible in NL to define the six unary modalities of [5] and the three binary modalities of [20] as abbreviated modalities [21]. This *adequacy* result relies on NL being a first order interval logic with the special length symbol ℓ. In [15] it is shown that SIL is an adequate first order interval logic by defining the modalities of NL as abbreviated modalities in SIL.

The rest of this paper is organized as follows. Sect. 2, Sect. 3 and Sect. 4 give the syntax, semantics and proof system of SIL, respectively. In Sect. 5 we sketch a proof showing that the proof system of SIL is sound and complete with respect to a certain class of models. In Sect. 6 we relate SIL to *arrow logic* [9] and *relational algebra* [18] to characterize the expressive power of SIL. Sect. 7 considers how to define *future intervals* for expressing liveness properties in SIL as this was a motivation for the development of SIL. Finally, Sect. 8 considers further work on SIL.

2 Syntax

The formulas of SIL are constructed from the following sets of symbols:

Var: An infinite set of *variables* $x, y, z, \ldots$.

FSymb: An infinite set of *function symbols* $f^n, g^m, \ldots$ equipped with arities $n, m \geq 0$. If f^n has arity $n = 0$ then f is called a *constant*. Constants will be denoted by $a, b, c, \ldots$.

PSymb: An infinite set of *predicate symbols* $G^n, H^m, \ldots$ equipped with arities $n, m \geq 0$. If G^n has arity $n = 0$ then G is called a *propositional letter*. Propositional letters will be denoted by $p, q, r, \ldots$. The predicate symbols also include the special binary predicate $=$.

A function/predicate symbol is either *rigid* or *flexible*. In particular, $=$ is rigid.

The set of *terms* $\theta, \theta_i \in Terms$ is defined by the following abstract syntax:

$$\theta \quad ::= \quad x \mid a \mid f^n(\theta_1, \ldots, \theta_n) \ .$$

The set of *formulas* $\phi, \psi \in Formulas$ is defined by the following abstract syntax:

$$\phi \quad ::= \quad p \mid G^n(\theta_1, \ldots, \theta_n) \mid \theta_1 = \theta_2 \mid \neg\phi \mid \phi \wedge \psi \mid \phi^\frown\psi \mid (\exists x)\phi \ .$$

We also use $\varphi, \phi_i, \psi_i, \varphi_i$ to denote formulas. A language consisting of variables, function and predicate symbols, and the symbols $(,)$, $=$, $\neg$, $\wedge$, $\frown$, and $\exists$ will be called a *chop-language*.

We use "sentence" as a synonym for "closed formula". If $x_1, \ldots, x_n$ are the free variables of ϕ then we denote $(\forall x_1) \ldots (\forall x_n)\phi$ the *universal closure* of ϕ. A formula is said to be *chop-free* if it does not contain the symbol $\frown$. A formula is said to be *flexible* if it contains a flexible symbol. Otherwise it is said to be *rigid*. For convenience, we use infix notation for binary functional or predicate symbols such as $+$ and $\leq$. We will use the standard abbreviations from first order predicate logic for the symbols $\vee$, $\Rightarrow$, $\Leftrightarrow$ and $\forall$. To avoid excessive use of parentheses we introduce the following precedences: 1. $\neg$, 2. $\frown$, 3. $\vee$, $\wedge$, 4. $\Rightarrow$, $\Leftrightarrow$, $\forall$, $\exists$.

3 Semantics

We start by giving a general Kripke-style possible worlds semantics.

Definition 1. *A model $\mathcal{M}$ for a chop-language is a quadruple (W, R, D, I) where*

- *W is a non-empty set of possible worlds and R is a ternary accessibility relation on W, thus $R \subseteq W \times W \times W$.*
- *D is a non-empty set.*
- *I is a function assigning an interpretation to each function/predicate symbol in each world:*

$$I \in \begin{pmatrix} FSymb \\ \cup \\ PSymb \end{pmatrix} \to W \to \left(\bigcup_{n \in \mathbb{N}} D^n \right) \to \begin{pmatrix} D \\ \cup \\ \{tt, ff\} \end{pmatrix}$$

such that (where $w \in W$):

$$\begin{array}{ll} I(a)(w) \in D, & I(f^n)(w) \in D^n \to D, \\ I(p)(w) \in \{tt, ff\}, & I(G^n)(w) \in D^n \to \{tt, ff\}, \end{array}$$

and such that the interpretation of a rigid symbol is the same in all worlds.

The pair (W, R) is called the *frame* and D the *domain* of the model. Compared to models of classical modal logic [7] the only difference is that the accessibility relation is ternary and not binary.

Given a model $\mathcal{M} = (W, R, D, I)$, an *$\mathcal{M}$-valuation* is a function $\mathcal{V}$ associating an element of the domain with each variable, thus $\mathcal{V} \in \mathit{Var} \to D$.

We denote by $\mathcal{M}, \mathcal{V}, w \models \phi$ that ϕ is *satisfied* in a world $w \in W$ of a model $\mathcal{M} = (W, R, D, I)$ under an $\mathcal{M}$-valuation $\mathcal{V}$. Satisfaction of formulas is inductively defined in a standard way [7]. The only interesting case is that of $^\frown$ (see [15] for the full definition):

$$\mathcal{M}, \mathcal{V}, w \models \phi^\frown\psi \quad \text{iff} \quad \mathcal{M}, \mathcal{V}, w_1 \models \phi \text{ and } \mathcal{M}, \mathcal{V}, w_2 \models \psi \text{ and } R(w_1, w_2, w) \text{ for some } w_1, w_2 \in W \ .$$

Given a set of formulas Γ, we say that $\mathcal{M}$ satisfies Γ if there is a world w of $\mathcal{M}$ and an $\mathcal{M}$-valuation $\mathcal{V}$ such that for every formula $\phi \in \Gamma$, $\mathcal{M}, \mathcal{V}, w \models \phi$. A formula ϕ is *valid in* $\mathcal{M}$ if for any world w of $\mathcal{M}$ and any $\mathcal{M}$-valuation $\mathcal{V}$, $\mathcal{M}, \mathcal{V}, w \models \phi$. ϕ is *valid in a class of models* $\mathcal{C}$ if it is valid in all models of $\mathcal{C}$.

We now proceed by presenting results necessary for defining the class of *signed interval models* which gives the more concrete semantics of SIL.

Definition 2. *A* signed temporal domain *is a non-empty set T.*

Contrary to work on (non-signed) interval logic with a general temporal domain [5,19,20,2,21] we do *not* require T to be totally ordered. In the case of SIL we can have a completeness result without this requirement and therefore choose to define T as general as possible. (But see the remark at the end of Sect. 5.)

Definition 3. *A* signed interval frame (W, R) *on a signed temporal domain T is defined by:*

- $W = T \times T$ *is the set of* signed intervals *on T.*
- $R \subseteq W \times W \times W$ *is the ternary accessibility relation on W defined by:*

$$R((t_1, t_1'), (t_2, t_2'), (t, t')) \quad \textit{iff} \quad t = t_1, t_1' = t_2, t_2' = t' \ .$$

This definition corresponds to the intuition of signed intervals given in the introduction. In particular, R expresses that three signed intervals are related as indicated in Fig. 3, thus $R((b, m), (m, e), (b, e))$.

We want to be able to refer to the *signed length* of a signed interval (c.f. the discussion in the introduction). For this we define the following:

Definition 4. *Given a signed interval frame (W, R) on a signed temporal domain T, a* signed measure *is a function $m \in W \to D$ where D is a set equipped with a binary operator $+$ and a distinguished element $0 \in D$. Furthermore, m has to satisfy the following conditions for any $t, t', u, u' \in T$ and $x, y \in D$:*

M1: *if* $m(t, u) = m(t, u')$ *then* $u = u'$
if $m(u, t) = m(u', t)$ *then* $u = u'$

M2: $m(t, t) = 0$

M3: $m(t, u) + m(u, t') = m(t, t')$

M4: $m(t, t') = x + y$ *iff* $m(t, u'') = x$ *and* $m(u'', t') = y$ *for some* $u'' \in T$

We now define which properties a domain of values to represent signed lengths should have in general (some of these will be implicitly given by the above definition).

Definition 5. *A* signed duration domain *is a group* $(D, +, -, 0)$.[1]

A chop-language which includes the symbols $\ell, +, -$ and 0 (where ℓ is flexible and $+, -, 0$ are rigid) will be called a *signed interval language*.

Definition 6. *Given a signed temporal domain* T*, a signed duration domain* D *and a signed measure* $m \in T \times T \to D$*, a* signed interval model *is a model* $\mathcal{M} = (W, R, D, I)$ *for a signed interval language where:*

- *The frame is the signed interval frame* (W, R) *defined by* T*.*
- *The domain is the signed duration domain* D*.*
- *The interpretation of* $\ell, +, -, 0$ *is such that*

$$I(\ell)(t,t') = m(t,t'), \quad I(0)(t,t') = 0, \quad I(+)(t,t') = +, \quad I(-)(t,t') = -$$

for any signed interval $(t,t') \in W$*.*

The class of all signed interval models will be denoted by $\mathcal{Z}$*. We say that a formula* ϕ *is* SIL-valid *(written* $\models_{\mathrm{SIL}} \phi$*) if it is valid in* $\mathcal{Z}$*.*

The semantics of the chop modality can be reformulated if we assume a signed interval model:

$$\mathcal{M}, \mathcal{V}, (b,e) \models \phi^\frown\psi \quad \text{iff} \quad \mathcal{M}, \mathcal{V}, (b,m) \models \phi \text{ and } \mathcal{M}, \mathcal{V}, (m,e) \models \psi$$

for some $m \in T$. This corresponds to the informal semantics given in Fig. 3.

4 Proof System

The axioms of SIL are:

A1: $((\phi^\frown\psi) \wedge \neg(\phi^\frown\varphi)) \Rightarrow (\phi^\frown(\psi \wedge \neg\varphi))$
$((\phi^\frown\psi) \wedge \neg(\varphi^\frown\psi)) \Rightarrow ((\phi \wedge \neg\varphi)^\frown\psi)$

A2: $((\phi^\frown\psi)^\frown\varphi) \Leftrightarrow (\phi^\frown(\psi^\frown\varphi))$

R: $(\phi^\frown\psi) \Rightarrow \phi$ if ϕ is a rigid formula
$(\phi^\frown\psi) \Rightarrow \psi$ if ψ is a rigid formula

B: $(((\exists x)\phi)^\frown\psi) \Rightarrow ((\exists x)(\phi^\frown\psi))$ if x is not free in ψ
$(\phi^\frown((\exists x)\psi)) \Rightarrow ((\exists x)(\phi^\frown\psi))$ if x is not free in ϕ

L1: $((\ell = x)^\frown\phi) \Rightarrow \neg((\ell = x)^\frown\neg\phi)$
$(\phi^\frown(\ell = x)) \Rightarrow \neg(\neg\phi^\frown(\ell = x))$

L2: $(\ell = x + y) \Leftrightarrow ((\ell = x)^\frown(\ell = y))$

[1] The main binary operator of the group is $+$ and its unary inverse operator is $-$.

L3: $\phi \Rightarrow (\phi^\frown(\ell = 0))$
$\phi \Rightarrow ((\ell = 0)^\frown\phi)$

The inference rules of SIL are:

Modus Ponens (MP): $\dfrac{\phi \quad \phi \Rightarrow \psi}{\psi}$ Generalization (G): $\dfrac{\phi}{(\forall x)\phi}$

Necessitation (N): $\begin{cases} \dfrac{\phi}{\neg(\neg\phi^\frown\psi)} \\ \dfrac{\phi}{\neg(\psi^\frown\neg\phi)} \end{cases}$ Monotonicity (M): $\begin{cases} \dfrac{\phi \Rightarrow \psi}{(\phi^\frown\varphi) \Rightarrow (\psi^\frown\varphi)} \\ \dfrac{\phi \Rightarrow \psi}{(\varphi^\frown\phi) \Rightarrow (\varphi^\frown\psi)} \end{cases}$

Furthermore, SIL contains axioms expressing the properties of a signed duration domain (c.f. Definition 5):

D1: $(\forall x)(\forall y)(\forall z)((x + y) + z = x + (y + z))$

D2: $(\forall x)(x + 0 = x)$
$(\forall x)(0 + x = x)$

D3: $(\forall x)(x + (-x) = 0)$
$(\forall x)((-x) + x = 0)$

Finally, SIL contains axioms of first order predicate logic with equality. Any axiomatic basis can be chosen but one has to be careful when instantiating universally quantified formulas. We can e.g. choose the following two axioms concerning universal quantification:

Q1: $(\forall x)\phi(x) \Rightarrow \phi(\theta)$ if θ is free for x in $\phi(x)$ and $\begin{cases} \theta \text{ is rigid or} \\ \phi(x) \text{ is chop-free} \end{cases}$

Q2: $(\forall x)(\phi \Rightarrow \psi) \Rightarrow (\phi \Rightarrow (\forall x)\psi)$ if x is not free in ϕ

Note the strengthened side condition in Q1 compared to the side condition of first order predicate logic which just reads: "if θ is free for x in $\phi(x)$" [10].

A *proof* of ϕ (in the proof system of SIL) is defined the standard way [10]. We write $\vdash_{\text{SIL}} \phi$ to denote that a proof of ϕ in SIL exists and we say that ϕ is a *theorem* of SIL. Similarly, given a set of formulas Γ, we define *deduction of* ϕ *in SIL from* Γ (written $\Gamma \vdash_{\text{SIL}} \phi$) the standard way. We write $\Gamma, \psi \vdash_{\text{SIL}} \phi$ for $(\Gamma \cup \{\psi\}) \vdash_{\text{SIL}} \phi$.

We end this section by observing that the proof system of SIL is very similar to that of ITL [2]. Only the axioms relating to the duration domain distinguish the two systems: In ITL the axioms D1–D3 are not present; instead five other related axioms are added. This concretize the assertion of the introduction that the proof system of SIL is not more complicated than that of ITL.

5 Soundness and Completeness

In this section we sketch the proof of the completeness result for SIL: The proof system of SIL is sound and complete with respect to the class of all signed

interval models. The completeness proof for SIL is inspired by the completeness proof for ITL [2].

The proof of completeness follows the general structure of a Henkin-style completeness proof [10,7]. The central idea in a Henkin-style proof is the following: Given an arbitrary formula ϕ which is not a theorem, *construct a model* which satisfies $\neg\phi$. This implies the non-validity of ϕ and the completeness follows.

We start by presenting some standard results used in Henkin-style completeness proofs, namely results concerning maximal consistent sets and witnesses.

Definition 7. *Let Γ be a set of sentences of a signed interval language $\mathcal{L}$.*

- *Γ is* consistent *(with respect to SIL) if there is no finite subset $\{\phi_1, \ldots, \phi_n\}$ of Γ such that $\vdash_{\text{SIL}} \neg(\phi_1 \wedge \ldots \wedge \phi_n)$.*
- *Γ is* maximal consistent *if it is consistent and there is no consistent set of sentences Γ' such that $\Gamma \subset \Gamma'$.*

Let $B = \{b_0, b_1, b_2, \ldots\}$ be an infinite, countable set of symbols not occurring in the signed interval language $\mathcal{L}$. Let $\mathcal{L}^+$ denote the signed interval language obtained by adding all symbols of B to $\mathcal{L}$ as rigid constants.

Definition 8. *A set Γ of sentences of $\mathcal{L}^+$ is said to have* witnesses in B *if for every sentence of Γ of the form $(\exists x)\phi(x)$ (where x is the only free variable of $\phi(x)$) there exists a constant $b_i \in B$ such that $\phi(b_i) \in \Gamma$.*

Theorem 1. *If Γ is a consistent set of sentences of $\mathcal{L}$, there is a set Γ^* of sentences of $\mathcal{L}^+$ which satisfies the following:*

$$\Gamma \subseteq \Gamma^*, \qquad \Gamma^* \text{ is maximal consistent}, \qquad \Gamma^* \text{ has witnesses in } B.$$

If Γ_0 is a consistent set of sentences of $\mathcal{L}$, let Γ_0^* be a set of sentences of $\mathcal{L}^+$ which existence is guaranteed by the above theorem.

Given a consistent set Γ_0 of sentences we can now *construct a model* $\mathcal{M}_0 = (W_0, R_0, D_0, I_0)$ where the worlds of W_0 are certain maximal consistent sets of sentences (including Γ_0^*), R_0 is defined by $R_0(\Delta_1, \Delta_2, \Delta)$ iff for any ϕ_1, ϕ_2, if $\phi_1 \in \Delta_1$ and $\phi_1 \in \Delta_2$ then $(\phi_1{}^\frown\phi_2) \in \Delta$, and D_0 is the set of equivalence classes w.r.t. $=$ on B. Finally, I_0 is defined for all symbols in all worlds. For example, in the case of a propositional letter we define $I_0(p)(\Delta) = (p \in \Delta)$, i.e. $\mathcal{M}_0, \mathcal{V}, \Delta \models p \quad \text{iff} \quad p \in \Delta$.

The following theorem generalizes the case of a propositional letter to arbitrary formulas [2].

Theorem 2. $\mathcal{M}_0, \mathcal{V}, \Delta \models \phi \quad$ *iff* $\quad \phi \in \Delta$.

The model $\mathcal{M}_0$ will play a central part in the construction of a satisfying signed interval model. Another important part in this construction will be played by the following proposition.

Proposition 1. *Let* $((\Delta_1, \Delta_2), (\Delta'_1, \Delta'_2)) \in (W_0 \times W_0) \times (W_0 \times W_0)$. *If* $R_0(\Delta_1, \Delta_2, \Gamma_0^*)$ *and* $R_0(\Delta'_1, \Delta'_2, \Gamma_0^*)$ *then there is a unique world* $\Delta \in W_0$ *such that* $R_0(\Delta_1, \Delta, \Delta'_1)$ *and* $R_0(\Delta, \Delta'_2, \Delta_2)$.

The intuition of this proposition can be given in terms of signed intervals: Given a pair of pairs $((\Delta_1, \Delta_2), (\Delta'_1, \Delta'_2))$ of consecutive signed intervals of the current signed interval Γ_0^* there is a unique signed interval Δ lying between the two chopping points of the two pairs of signed intervals. We have sketched this intuition in Fig. 4.

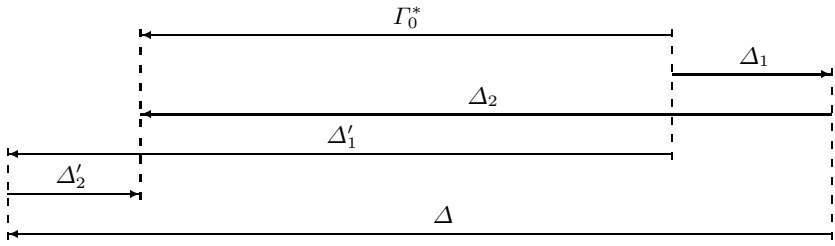


Fig. 4. Possible configuration of the worlds of Proposition 1

We will now start constructing a signed interval model from $\mathcal{M}_0$. For this we need to define a signed temporal domain T (c.f. Definition 6):

$$T = \{\ (\Delta_1, \Delta_2) \in W_0 \times W_0 \mid R_0(\Delta_1, \Delta_2, \Gamma_0^*)\ \}\ .$$

The intuition behind this particular definition of T is the following: If we think of the worlds of W_0 as signed intervals, T is the set of all pairs of consecutive signed intervals of the current signed interval. These pairs define, by means of their chopping points, all the necessary temporal points. See Fig. 5.

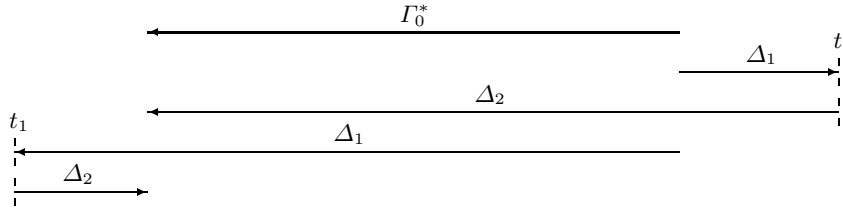


Fig. 5. Intuitively, the points of T are the "chopping points" (marked by t_1 and t_2 on the figure) of the pairs (Δ_1, Δ_2) related by $R_0(\Delta_1, \Delta_2, \Gamma_0^*)$. The figure shows two of the possible pairs (Δ_1, Δ_2).

We have now come to the crucial step in the construction. Intuitively, we want to identify a signed interval given by two points of T with a signed interval of W_0. But this connection is exactly what Proposition 1 gives us. Formally, let $\mu : T \times T \to W_0$ such that $\mu((\Delta_1, \Delta_2), (\Delta'_1, \Delta'_2))$ is the world Δ given by Proposition 1. Revisit Fig. 4 for the intuition.

We are now ready to construct a model $\mathcal{M} = (W, R, D, I)$ on the basis of $\mathcal{M}_0$ as follows:

- The frame (W, R) is the signed interval frame defined by T.
- The domain D is the same as D_0.
- The interpretation function I is given by $I(s)(t,t') = I_0(s)(\mu(t,t'))$ for any symbol s and any signed interval (t,t').

The following proposition shows that $\mathcal{M}$ is indeed a signed interval model.

Proposition 2. *The constructed model $\mathcal{M}$ is a signed interval model.*

Proof. We have to make sure that D of $\mathcal{M}$ is a signed duration domain and that the interpretation is as specified in Definition 6.

The rigid symbols $-$ and $+$ of $\mathcal{L}$ defines a unary and a binary operation we also denote by $-$ and $+$ in D. The interpretation of the rigid constant 0 will be an element of D we also denote by 0. As D1–D3 are valid in $\mathcal{M}_0$ they will by Theorem 3 (see below) also be valid in $\mathcal{M}$, hence $(D, +, -, 0)$ is a signed duration domain. We now only need an appropriate signed measure. But it can be shown [15] that the interpretation of ℓ is already defined such that M1–M4 of Definition 4 are satisfied. Thus, we define the signed measure by $m(t,t') = I(\ell)(t,t')$ for any signed interval $(t,t') \in W$. □

We want to establish a connection between satisfaction of formulas in $\mathcal{M}$ and $\mathcal{M}_0$: We want to show that a formula is satisfied in a world (t,t') of $\mathcal{M}$ iff it is satisfied in the corresponding world $\mu(t,t')$ of $\mathcal{M}_0$. The only difficulty is in the case of chop. For this we need the following two propositions (see [15] for proofs).

Proposition 3. *If $t, t', u \in T$ then $R_0(\mu(t,u), \mu(u,t'), \mu(t,t'))$.*

Proposition 4. *Let $t, t' \in T$ and $\Gamma_1, \Gamma_2 \in W_0$. If $R_0(\Gamma_1, \Gamma_2, \mu(t,t'))$ then $\Gamma_1 = \mu(t,u)$ and $\Gamma_2 = \mu(u,t')$ for some $u \in T$.*

We can now formulate the connection between $\mathcal{M}$ and $\mathcal{M}_0$. We note that since the domains of $\mathcal{M}$ and $\mathcal{M}_0$ are the same, an $\mathcal{M}$-valuation is also an $\mathcal{M}_0$-valuation.

Theorem 3. $\mathcal{M}, \mathcal{V}, (t,t') \models \phi$ *iff* $\mathcal{M}_0, \mathcal{V}, \mu(t,t') \models \phi$.

Proof. The proof is by a straightforward structural induction on ϕ: In the case of ϕ being $\psi^\frown\varphi$ we use Propositions 3 and 4. See [15]. □

We can now establish the main result of this section.

Theorem 4. *If Γ_0 is a consistent set of sentences (with respect to SIL) then we can construct a signed interval model which satisfies Γ_0.*

Proof. We know by Proposition 2 that the constructed model $\mathcal{M}$ is a signed interval model. We are therefore done if we can show that $\mathcal{M}$ satisfies Γ_0.

It is possible to find worlds $\Delta_1, \Delta_2 \in W_0$ such that $R_0(\Delta_1, \Gamma_0^*, \Gamma_0^*)$ and $R_0(\Gamma_0^*, \Delta_2, \Gamma_0^*)$ (see [15]). Thus, both $t = (\Delta_1, \Gamma_0^*)$ and $t' = (\Gamma_0^*, \Delta_2)$ belong to T, hence $(t,t') \in W$. It is now immediate (by definition of μ) that $\mu(t,t') = \Gamma_0^*$. Then, utilizing Theorems 3 and 2, we are done. □

From the above theorem the completeness of SIL now follows easily.

Theorem 5. *A formula ϕ of a signed interval language is valid in $\mathcal{Z}$ (the class of all signed interval models) iff it is a theorem of SIL, thus*

$$\models_{\text{SIL}} \phi \quad \textit{iff} \quad \vdash_{\text{SIL}} \phi \ .$$

Proof. For the *if*-part (soundness) we simply have to check that all axioms of SIL are valid in $\mathcal{Z}$ and that all inference rules of SIL preserve validity. This is straightforward.

For the *only if*-part (completeness) assume ϕ is *not* a theorem of SIL. We now have to show that ϕ is not valid in some signed interval model. Let ϕ' be the universal closure of ϕ; ϕ' is not a theorem either. The set $\{\neg\phi'\}$ will therefore be consistent and we can construct a signed interval model $\mathcal{M}$ which satisfies $\neg\phi'$ (Theorem 4). Since $\neg\phi'$ is satisfied by $\mathcal{M}$, ϕ' is not valid in $\mathcal{M}$ and neither is ϕ. □

Remark. The above completeness result is for a general class of signed interval models with no ordering on the underlying temporal domains. To justify the name "interval logic" one could argue that it would be more natural to require a total ordering on these domains. In [16] it is shown how a completeness result can be established in this case.

6 Arrow Logic and Relational Algebra

In this section we establish results relating SIL to *arrow logic* [9] and *relational algebra* [18]. These results rely on the capability to define an abbreviated unary modality $^{-1}$ in SIL which "reverses" the direction of an interval. We define:

$$\phi^{-1} \quad \widehat{=} \quad (\exists x)(\ (\ell = x) \wedge (\ (\ell = 0) \wedge (\ell = x)^\frown \phi\ )^\frown \text{true}\ ) \ ,$$

where true $\widehat{=}\ 0 = 0$ and x is some variable not free in ϕ. The following proposition can now be proved [15].

Proposition 5. *For any signed interval model $\mathcal{M}$, valuation $\mathcal{V}$ and signed interval (b, e):*

$$\mathcal{M}, \mathcal{V}, (b, e) \models \phi^{-1} \quad \textit{iff} \quad \mathcal{M}, \mathcal{V}, (e, b) \models \phi \ .$$

Arrow logic [9] is a modal logic where the possible worlds are pairs of elements from some set. We see that this corresponds to signed intervals of SIL which makes a comparison interesting.

Arrow logic is equipped with a constant $\iota\delta$, a unary modality $\otimes$ and a binary modality $\circ$. The semantics of $\iota\delta$, $\otimes$ and $\circ$ can informally be given in terms of SIL: $\iota\delta$ corresponds to $(\ell = 0)$, $\otimes$ to $^{-1}$, and $\circ$ to $^\frown$. In arrow logic $\iota\delta$, $\otimes$ and $\circ$ are basic modalities and not abbreviations of some kind. In SIL we can define $^{-1}$ using $^\frown$ and ℓ. Thus, we conclude that SIL can express the same as arrow logic with just the basic modality $^\frown$ (corresponding to $\circ$) and then the special symbol ℓ.

But this expressive power of SIL has a price: Firstly, the introduction of ℓ restricts the set of possible models considerably. Secondly, to define $^{-1}$ it was necessary to use a first order construct to quantify over the value of ℓ. In conclusion we can therefore (rather informally) state that: SIL is a first order arrow logic with the special symbol ℓ.

We now consider another consequence of $^{-1}$: It seems natural to think of signed intervals as *binary relations*, hence $(t, t') \in T \times T$ asserts that t is related to t'. In some signed interval model, a formula ϕ will either be true or false on some signed interval. If we now consider the set of all signed intervals on which ϕ is true we will have a binary relation on T. We will in the following pursue this idea by relating SIL to relational algebra [18].

Definition 9. *A* relational algebra $\mathfrak{RA} = \langle \mathcal{S}, \oplus, \odot, \circ, \ominus, \otimes, \mathbf{0}, \mathbf{1}, \iota\delta \rangle$ *is a non-empty set* $\mathcal{S}$ *equipped with three binary operators* $\oplus, \odot, \circ$*, two unary operators* $\ominus, \otimes$*, and three constants* $\mathbf{0}, \mathbf{1}, \iota\delta$ *such that* $\langle \mathcal{S}, \oplus, \odot, \ominus, \mathbf{0}, \mathbf{1} \rangle$ *is a Boolean algebra and the following axioms are satisfied for all* $x, y, z \in S$*:*

$$
\begin{array}{llll}
RA1 & (x \oplus y) \circ z = (x \circ z) \oplus (y \circ z) & RA5 & x \circ \iota\delta = x \\
RA2 & \otimes(x \oplus y) = \otimes x \oplus \otimes y & RA6 & \otimes\otimes x = x \\
RA3 & (x \circ y) \circ z = x \circ (y \circ z) & RA7 & \otimes(x \circ y) = \otimes y \circ \otimes x \\
RA4 & \ominus(\otimes x \circ \ominus(x \circ y)) \oplus \ominus y = \mathbf{1} & &
\end{array}
$$

We now formally define how to build the above mentioned binary relations.

Definition 10. *Let* $\mathcal{M} = (W, R, D, I)$ *be a signed interval model and* $\mathcal{V}$ *be a* $\mathcal{M}$*-valuation. As* $\mathcal{M}$ *is a signed interval model we will have* $W = T \times T$ *for some set* T*. We now define a* SIL-relation *of* ϕ *(written* $\mathrm{R}_{\mathcal{M},\mathcal{V}}(\phi)$*) by:*

$$\mathrm{R}_{\mathcal{M},\mathcal{V}}(\phi) = \{(t, t') \in T \times T \mid \mathcal{M}, \mathcal{V}, (t, t') \models \phi\} \ .$$

Furthermore, we define the set $\mathcal{R}_{\mathcal{M},\mathcal{V}}$ *of all SIL-relations in a given model and valuation:*

$$\mathcal{R}_{\mathcal{M},\mathcal{V}} = \{\mathrm{R}_{\mathcal{M},\mathcal{V}}(\phi) \mid \phi \in \mathit{Formulas}\} \ .$$

To establish the connection to relational algebra, we associate three binary operators $+, \cdot, ;$, two unary operators $-, \smile$ and three constants $0, 1, 1'$ with $\mathcal{R}_{\mathcal{M},\mathcal{V}}$. The meaning of these operators and constants is given by the following equivalences:

$$
\begin{array}{rclrcl}
1 & = & \mathrm{R}_{\mathcal{M},\mathcal{V}}(\mathrm{true}) & -\mathrm{R}_{\mathcal{M},\mathcal{V}}(\phi) & = & \mathrm{R}_{\mathcal{M},\mathcal{V}}(\neg\phi) \\
0 & = & \mathrm{R}_{\mathcal{M},\mathcal{V}}(\mathrm{false}) & \mathrm{R}_{\mathcal{M},\mathcal{V}}(\phi) + \mathrm{R}_{\mathcal{M},\mathcal{V}}(\psi) & = & \mathrm{R}_{\mathcal{M},\mathcal{V}}(\phi \vee \psi) \\
1' & = & \mathrm{R}_{\mathcal{M},\mathcal{V}}(\ell = 0) & \mathrm{R}_{\mathcal{M},\mathcal{V}}(\phi) \cdot \mathrm{R}_{\mathcal{M},\mathcal{V}}(\psi) & = & \mathrm{R}_{\mathcal{M},\mathcal{V}}(\phi \wedge \psi) \\
\mathrm{R}^{\smile}_{\mathcal{M},\mathcal{V}}(\phi) & = & \mathrm{R}_{\mathcal{M},\mathcal{V}}(\phi^{-1}) & \mathrm{R}_{\mathcal{M},\mathcal{V}}(\phi); \mathrm{R}_{\mathcal{M},\mathcal{V}}(\psi) & = & \mathrm{R}_{\mathcal{M},\mathcal{V}}(\phi^{\frown}\psi)
\end{array}
$$

Any SIL-relation build using any of the three constants and five operators can thus by simple equational reasoning be transformed to a single SIL-relation $\mathrm{R}_{\mathcal{M},\mathcal{V}}(\phi)$ for some formula ϕ.

To show equivalence of SIL-relations we have the following lemma.

Lemma 1. $\models_{\mathrm{SIL}} \phi \Leftrightarrow \psi \quad \textit{implies} \quad \mathrm{R}_{\mathcal{M},\mathcal{V}}(\phi) = \mathrm{R}_{\mathcal{M},\mathcal{V}}(\psi)$.

We can now formulate the following theorem saying that $\mathcal{R}_{\mathcal{M},\mathcal{V}}$ together with the above defined operators and constants is a relational algebra.

Theorem 6. $\mathfrak{R}_{\mathrm{SIL}} = \langle \mathcal{R}_{\mathcal{M},\mathcal{V}}, +, \cdot, ;, -, ^\smile, 0, 1, 1' \rangle$ *is a relational algebra.*

Proof. By Definition 9 we must first show that $\langle \mathcal{R}_{\mathcal{M},\mathcal{V}}, +, \cdot, -, 0, 1 \rangle$ is a Boolean algebra. But this follows easily due to the standard correspondence between propositional logic and Boolean algebra.

We must then show that $\mathfrak{R}_{\mathrm{SIL}}$ satisfies the axioms RA1–RA7 for arbitrary members of $\mathcal{R}_{\mathcal{M},\mathcal{V}}$. For this we use Lemma 1. For example, we can show that $\mathfrak{R}_{\mathrm{SIL}}$ satisfies RA6 by showing $\models_{\mathrm{SIL}} (\phi^\frown\psi)^{-1} \Leftrightarrow \psi^{-1}{}^\frown\phi^{-1}$. But this is not difficult using Proposition 5. See [15] for the full proof of the theorem. □

Theorem 6 gives a nice theoretical characterization of the expressive power of SIL. It is only establishable because $^{-1}$ is definable in SIL.

7 Future Intervals

As discussed in the introduction, SIL has the ability to express liveness properties. An abstract liveness property could e.g. be that some property will hold eventually.

To be able to express such properties concisely in SIL it would be convenient to have modalities saying that a formula will hold on some future interval or on all future intervals with respect to the current interval. But what exactly should we consider to be a future interval? And how should we define abbreviated modalities in SIL expressing this? We will briefly consider these questions in this section. There is a more comprehensive discussion in [15].

Firstly, we have to assume a total ordering $\leq$ on both the signed temporal domain and on the signed duration domain. A completeness result for SIL in this case is established in [16]. A definition of a future interval could then be the following: If the current interval is (b, e) then a future interval is any interval (m, n) with $n \geq e$ and $m \geq b$. This can be illustrated by the following two figures:

We can also consider definitions of future intervals which are independent of the direction of the current interval. We can e.g. define a future interval as any interval (m, n) where both m and n are greater than $\min(b, e)$ where (b, e) is the current interval. This can be illustrated by the following two figures:

Both the above proposals are reflexive and transitive which seems to be very desirable properties for practical use. In [15] other proposals are discussed but they are discarded as they are either not reflexive or not transitive.

According to the above two proposals for future intervals, the goal is now to define two abbreviated modalities $\boxminus$ and $\diamondminus$ such that $\boxminus\phi$ holds on an interval iff ϕ holds on all future intervals and $\diamondminus\phi$ holds on an interval iff ϕ holds on some future interval. The first proposal gives rise to the following two abbreviations:

$$\diamondminus\phi \mathrel{\widehat{=}} (\ell \geq 0)^\frown(\phi^\frown(\ell \leq 0)) \quad \text{and} \quad \boxminus\phi \mathrel{\widehat{=}} \neg\diamondminus(\neg\phi) \ .$$

The second proposal gives rise to the following abbreviations:

$$\begin{array}{lcl} \diamondminus\phi & \mathrel{\widehat{=}} & ((\ell \geq 0)^\frown(\phi^\frown(\ell \leq 0)) \wedge (\ell = 0))^\frown\text{true} \vee \\ & & \text{true}^\frown((\ell \geq 0)^\frown(\phi^\frown(\ell \leq 0)) \wedge (\ell = 0)) \quad \text{and} \\ \boxminus\phi & \mathrel{\widehat{=}} & \neg\diamondminus(\neg\phi) \ . \end{array}$$

Which definition of future intervals to choose of course depends on the particular problem at hand. A reason for choosing the first proposal would be that the corresponding modalities are fairly simple compared to the modalities of the second proposal. On the other hand, in [15] an example is considered where the second proposal is chosen because the first turns out inadequate. This is due to the fact that all subintervals of the current interval are future intervals in the second proposal whereas they are not in the first.

8 Further Work on SIL

As mentioned in the previous section, there is a more comprehensive discussion of future intervals in [15]. Here it is also discussed how to define a contracting chop in SIL.

Two simple modalities $\Diamond$ and $\Box$ are introduced in [15] such that a formula $\Box\phi$ holds on a signed interval iff ϕ holds on all possible signed intervals and $\Diamond\phi$ holds on a signed interval iff ϕ holds on some signed interval. By means of $\Box$ a deduction theorem for SIL is established: If $\Gamma, \phi \vdash_{\text{SIL}} \psi$ then $\Gamma \vdash_{\text{SIL}} \Box\phi \Rightarrow \psi$ (with a sidecondition on free variables of ϕ).

In [15] several proofs in the proof system of SIL are conducted. Various general conventions and results for proof-making in SIL are established.

An extension to SIL called Signed Duration Calculus (SDC) is developed in [15]. The basic idea of this is the same as that in [22,6]. Syntax, semantics, and proof system for SDC is given in [15].

In [15] a very simple example concerning liveness and proof of correctness hereof is considered. We hope to consider a larger case study in SIL/SDC in the future and we also hope to investigate further theoretical results concerning e.g. decidability and proof theory for SIL/SDC.

9 Acknowledgements

I would like to thank Michael R. Hansen and Hans Rischel for valuable comments and suggestions to my work on Signed Interval Logic.

References

1. J.F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983. 159
2. B. Dutertre. Complete Proof Systems for First Order Interval Temporal Logic. In *LICS'95*, pages 36–43. IEEE Press, 1995. 157, 157, 161, 163, 164, 164
3. M. Engel and H. Rischel. Dagstuhl-Seminar Specification Problem - a Duration Calculus Solution. Dept. of Computer Science, Technical University of Denmark – Private Communication, 1994. 158
4. J.Y. Halpern, B. Moszkowski, and Z. Manna. A Hardware Semantics based on Temporal Intervals. In *ICALP'83*, volume 154 of *LNCS*, pages 278–291. Springer, 1983. 157
5. J.Y. Halpern and Y. Shoham. A Propositional Modal Logic of Time Intervals. *Journal of the ACM*, 38(4):935–962, 1991. 157, 159, 159, 159, 161
6. M.R. Hansen and Zhou Chaochen. Duration Calculus: Logical Foundations. *Formal Aspects of Computing*, 9(3):283–330, 1997. 157, 170
7. G.E. Hughes and M.J. Cresswell. *An Introduction to Modal Logic*. Routledge, 1968. 160, 161, 164
8. H.R. Lewis. A Logic of Concrete Time Intervals. In *LICS'90*, pages 380–389. IEEE Press, 1990. 157
9. M. Marx and Y. Venema. *Multi-Dimensional Modal Logic*, volume 4 of *Applied Logic Series*. Kluwer Academic Publishers, 1997. 159, 167, 167
10. E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth & Brooks/Cole Advanced Books & Software, 3. edition, 1987. 163, 163, 164
11. B. Moszkowski. A Temporal Logic for Multilevel Reasoning about Hardware. *IEEE Computer*, 18(2):10–19, 1985. 157
12. B. Moszkowski. Some Very Compositional Temporal Properties. In *Programming Concepts, Methods, and Calculi*, volume A-56 of *IFIP Transactions*, pages 307–326. North Holland, 1994. 157
13. Y.S. Ramakrishna, P.M. Melliar-Smith, L.E. Moser, L.K. Dillon, and G. Kutty. Interval logics and their decision procedures. Part I: An interval logic. *Theoretical Computer Science*, 166:1–47, 1996. 157
14. Y.S. Ramakrishna, P.M. Melliar-Smith, L.E. Moser, L.K. Dillon, and G. Kutty. Interval logics and their decision procedures. Part II: A real-time interval logic. *Theoretical Computer Science*, 170:1–46, 1996. 157
15. T.M. Rasmussen. Signed Interval Logic. Master's thesis, Dept. of Information Technology, Technical University of Denmark, January 1999. 159, 161, 166, 166, 166, 166, 167, 169, 169, 170, 170, 170, 170, 170, 170, 170, 170
16. T.M. Rasmussen. Signed Interval Logic on Totally Ordered Domains. Note, Dept. of Information Technology, Technical University of Denmark, June 1999. 167, 169
17. R. Rosner and A. Pnueli. A Choppy Logic. In *LICS'86*, pages 306–313. IEEE Press, 1986. 157
18. G. Schmidt and T. Ströhlein. *Relations and Graphs*. Springer, 1993. 159, 167, 168
19. Y. Venema. Expressiveness and Completeness of an Interval Tense Logic. *Notre Dame Journal of Formal Logic*, 31(4):529–547, 1990. 157, 159, 159, 161
20. Y. Venema. A Modal Logic for Chopping Intervals. *Journal of Logic and Computation*, 1(4):453–476, 1991. 157, 159, 159, 159, 161
21. Zhou Chaochen and M.R. Hansen. An Adequate First Order Interval Logic. In *COMPOS'97*, volume 1536 of *LNCS*, pages 584–608. Springer, 1998. 157, 157, 158, 159, 159, 161
22. Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40(5):269–276, 1991. 157, 170

Quantitative Temporal Logic

Yoram Hirshfeld and Alexander Rabinovich

Sackler Faculty of Exact Sciences
Tel Aviv University
{yoram, rabino}@math.tau.ac.il

Abstract. We define a quantitative Temporal Logic that is based on a simple modality within the framework of Monadic Predicate Logic. Its canonical model is the real line (and not an ω-sequence of some type). We prove its decidability using general theorems from Logic (and not Automata theory). We show that it is as expressive as any alternative suggested in the literature.

1 Introduction

1.1 Summary of the Results

Temporal Logic (TL) is a convenient framework for reasoning about the evolving of a system in time. This made TL a popular subject in the Computer Science community and it enjoyed extensive research during the last 20 years. In temporal logic the relevant properties of the system are described by *Atomic Propositions* that hold at some points in time and not at others. More complex properties are described by formulas built from the atoms using Boolean connectives and *Modalities* (temporal connectives): a k-place modality C transforms statements $\varphi_1, \ldots, \varphi_k$ on points possibly other than the given point t_0 to a statement $C(\varphi_1, \ldots, \varphi_k)$ on the point t_0. The rule that specifies when is the statement $C(\varphi_1, \ldots, \varphi_k)$ true for the given point is called *Truth Table* in [GHR94]. The choice of the particular modalities with their truth table determines the different temporal logics. The most basic modality is the one place "diamond" modality $\Diamond X$ saying "X holds some time in the future". Its truth table is usually formalized by $\varphi_{\Diamond}(t_0, X) \equiv (\exists t > t_0)X(t)$ [GHR94].

The truth table of $\Diamond$ is a formula of the Monadic Logic of Order (MLO). MLO is a fundamental formalism in Mathematical Logic, part of the general framework of Predicate Logic. Its formulas are built using atomic propositions $X(t)$ (similar to the atoms X of TL), atomic relations between elements $t_1 = t_2$, $t_1 < t_2$ and using Boolean connectives and (first order) quantifiers $\exists t$ and $\forall t$ (occasionally we shall be interested in second order MLO that has also quantifiers $\exists X$ and $\forall X$). Practically all the modalities used in the literature have their truth table defined in MLO and as a result every formula of the temporal logic translates directly into an equivalent formula of MLO. Therefore, the different temporal logics may

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 172–187, 1999.

be considered a convenient way to use fragments of MLO. There is a lot to be gained from adopting this point of view: the rich theory concerning MLO and in particular the decidability results concerning MLO apply to TL. MLO can also serve as a yardstick by which to check the strength of the temporal logic chosen: a temporal logic is *expressively complete* if every formula of MLO with single variable t_0 is equivalent to a temporal formula. An expressively complete temporal logic is as strong as can be expected.

Actually the notion of expressive completeness refers to a temporal logic and to a model (or a class of models) since the question if two formulas are equivalent depends on the domain over which they are evaluated. Any ordered set with monadic predicates is a model for TL and MLO, but the main, *canonical*, intended models are the non-negative integers $\langle N, <, 0\rangle$ for discrete time and the non-negative reals $\langle R^+, <, 0\rangle$ for continuous time. There may be reasons to use other models but they should be spelled out explicitly if the need arises to ignore the natural model. A major result concerning TL is Kamp's proof [Kamp68] (reproved in [GPSS80]) that the pair of modalities X *until* Y and X *since* Y is expressively complete for the two canonical models (but not for less natural models like the rationals). *Note* that since this paper is not concerned with discrete time we mean from now on only the model of non-negative reals R^+ when we speak of the canonical model.

Sometimes, in particular in Computer Science it is natural to restrict the attention to a subclass of unary predicates over R^+, the class of predicates with "finite variability"; i.e - predicates that change from *true* to *false* only finitely often in any finite interval of time. We shall call the standard model with these finite variability predicates the "canonical finite variability model". It is clear that a predicate P is a finite variability predicate if and only if there is an unbounded increasing sequence t_i such that P is constant on any interval (t_i, t_{i+1}).

MLO and its derived temporal logics are not suitable to deal with *quantitative* properties. The most basic quantitative property is "X will happen within one unit of time". This is analogue to the discrete case modality "X will happen at the next step". A natural way to deal with quantitative modalities is by extending MLO to MLO_1-monadic logic of order with the $+1$ function: adding to the language the function $S(t) = t + 1$. The basic modality described above is then described by the formula:

$$\varphi(t_0, X) \equiv \exists t(t_0 < t < t_0 + 1 \wedge X(t)) \tag{1}$$

The canonical model R^+ is also the canonical model for MLO_1 but there may be other models: R^+ with a different time scale (e.g. $S(t) = 2^t$) or any ordered set with a function $+1$ that satisfies some obvious axioms (see Section 3). Unfortunately, MLO_1 is too expressive and it is undecidable over the canonical model (this follows also from [AFH96]). Quite a few formalisms have been suggested in the literature in order to supply TL with the capability to deal with quantitative properties in a decidable way. Many of them are surveyed in [AH92]. We add one that seems to be the simplest and the right one:

a) We take **(1)** as a table for the modality $\diamondsuit_1 X$ and its companion (2) as the table for the modality $\diamondsuit_1 X$ - "X happened during the previous unit of time":

$$\psi(t_0, X) \equiv \exists t(t_0 - 1 < t < t_0 \wedge X(t)) \ . \tag{2}$$

The $t-1$ is interpreted as 0 for $t < 0+1$ and as the standard $t-1$ function for $t \geq 0+1$.

b) We add those two simple modalities to TL obtaining *quantitative temporal logic* QTL. We also identify the corresponding fragment QMLO of MLO_1.

c) We show that these logics are as expressive as any of the previously suggested constructs by showing how these constructs are defined in QMLO.

d) We show that the validity (and satisfiability) problems for this logic is decidable over the canonical model by reducing the problem to decidability of pure MLO and using the decidability results for MLO in [BG85].

e) We show that the satisfiability problem for QTL in the canonical finite variability model is PSPACE complete. In view of (c) the last two items reprove the decidability and the complexity results for logics like MITL without resorting in the proof to automata theory. It may seem surprising that automata theory does not yield any tighter bounds than straightforward translations and general logical considerations.

f) The whole discussion except for the complexity results applies uniformly to the canonical finite variability model and to the canonical general model. No other approach was even able to represent the most natural canonical model.

While QTL is as expressive as any of its rivals in the literature, it can be enriched to stronger yet still decidable logics. For example A. Pnueli's modality in X and Y: $P(t_0, X, Y) \equiv \exists t_1 t_2[t_0 < t_1 < t_2 < t_0 + 1 \wedge X(t_1) \wedge Y(t_2)]$ and similar constructs can be added retaining decidability. However, none of these modalities seem general enough to be officially added to QTL. The search for a natural stronger logic still continues [HR99].

We believe that our formalism is *exactly the right one*: We use the most natural model, we use the most simple pair of quantitative modalities, everything is done within plain mathematical logic with no new or ad-hoc constructs, we prove decidability using mainstream logic and our formalism works just as well for general systems that may vary infinitely often in a finite length of time.

1.2 Comparison with Previous Works

Many formalisms were suggested in the last 15 years to extend temporal logic to deal with quantitative properties, as surveyed in [AH92] and [Wilke94].

> "The number of formalisms that purportedly facilitate the modeling, specifying and proving of timing properties for reactive systems has exploded over the past few years. The authors, who confess to have added to the confusion by advancing a variety of different syntactic and semantic proposals, feel that it would be beneficial to pause for a second - to pause and look back to sort out what has been accomplished and what needs to be done." [AH92].

> "Recent research in reactive and hybrid systems is dominated by a plethora of Concepts, Terminology, Notations. Unfortunately, this background is not free of ad-hoc and ambiguous decision which are liable to misjudgments and to infliction of myths into the area" [Trak99].

So it is not easy to compare our work with the related literature. We shall try to compare our work to the previous results regarding four aspects: the model used, the temporal connectives introduced, the decidability proof technique and the complexity of the method.

Models In [AH92] (section 2.2) there is a classification of all the models leading to "sixteen possible formal semantics of real time sequences". All sixteen models are ω-sequences of some kind. None of them is the real line itself. None of them reflects the real line faithfully and none avails itself to be adjusted to systems without finite variability. In section 7 we shell compare the canonical model with two of the most popular models.

Formal Language Some of the formalisms are obtained from logics or temporal logics by adding programming language constructs which lack the universality of logical notions. Notions such as freeze quantification, clock variables, half-order logics, explicit-clock notations, reset quantifications, position variables and position quantifications, nonstandard reals (which have nothing to do with the non-standard analysis) etc were invented [AH92,BL95,HRS98].

Others use temporal connectives which are definable in a fragment of first or second order Monadic Logic. We will discuss only the latter. Notably among logics with first-order defined connectives is MITL of [AFH96]. All such decidable languages are equal in expressive power to the most basic logic QTL on the finite variability canonical model. On the different ω-sequences models the logics differ in expressive power, causing a proliferation of logics. Thus "ϕ will happen within two units of time" is expressible in the canonical models through the connective "ϕ will happen within one time unit" but not so in the popular model of state sequences (see section 7).

Some modalities suggested in the literature were motivated by pragmatic considerations. Some other were obtained annotating the authors favorite modalities for TL by some time constrains. For example, the operator $\Diamond_{[1,3]}$ is interpreted as "eventually within one to three time units." There is no yardstick by which to measure the appropriateness of a temporal logic.

The most important characteristic of these formalisms are (1) expressiveness, which refers to correctness properties which the formalism can specify and (2) Complexity and decidability - the complexity of the model checking and the satisfiability problems for the formalism. One of the main goals was to formulate the most expressive logic which is still decidable.

Decidability Alur and Dill introduced time automata and proved that the emptiness problem is decidable for the class of these automata [AD94]. The proof

reduces the problem to the emptiness problem for Büchi ω-automata. This result was used in all the previous approaches to prove that the logic is decidable (we suspect that this is the main reason why all the authors ignored the standard model in favor of an ω sequence model of some kind).

The class of languages accepted by timed automata is not closed under complementation. On the other hand logic is closed under negation. Therefore the usual way to show that a logic was decidable was to introduce some variations and restriction of timed automata which is closed under complementation and providing an effective transformation from logical formulae to "equivalent" automata in this class. These classes were given nice descriptive names like "Recursive Event Clock Automata" [HRS98] or "Bounded Two Way Deterministic Timed Automata" [AH92a]. It seems that these classes of automata do not give any additional insight about the corresponding logic.

In contrast our decidability proof remains within the framework of logic first reducing the language to a normal form ("Timer normal form") and then reducing the problem to the known result about the decidability of pure MLO (or TL). In particular our proof applies just as well to the standard model with all the unary predicates (and not just finite variability predicates).

Complexity It may have been hoped that the use of automata theory for decidability would provide a less complex procedure and clearer bounds than decidability proofs based on general logical reductions. We found this not to be the case; our method easily yields the same upper bounds or better ones (see section 6).

2 Monadic Logic of Order (MLO) and Temporal Logic (TL)

We survey the basic facts about these logics introducing TL within the framework of MLO [GHR94].

The syntax of MLO has in its vocabulary *individual (first order) variables* $t_0, t_1, \ldots$, *monadic predicate names* $X_0, X_1, \ldots$, and one binary relation $<$ (the order). *Atomic formulas* are of the form $X(t), t_1 < t_2$ and $t_1 = t_2$. *MLO formulas* are obtained from atomic formulas using the Boolean connectives $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$ and the (first order) quantifiers $\exists t$ and $\forall t$.

Second order Monadic Logic of Order formulas are obtained using also second order quantifiers $\exists X$ and $\forall X$.

Warning on Terminology. In this work we are mainly interested in first-order monadic logic of order, unless explicitly stated otherwise.

As usual if φ is a formula we may write $\varphi(t_1, \ldots t_k; X_1, \ldots, X_m)$ to indicate that the free variables in φ are among $t_1, \ldots t_k$ and the monadic predicate names are among $X_1, \ldots X_m$.

A structure for MLO is a pair $M = \langle A, < \rangle$ where $<$ is a linear order over A. The important examples for us are: $\langle R^+, < \rangle$, and $\langle N, < \rangle$; the non-negative real

line and the non-negative integers. Note that at this stage, we made the choice to deal only with linear order so that trees and other orders are not in the scope of the discussion at present.

We shall not repeat the inductive definition saying when is a formula satisfied. Recall that in order to check if the formula $\varphi(t_1, \dots t_k; X_1, \dots, X_m)$ is true we need to specify which model $M = \langle A, < \rangle$ is intended and what are the elements $\tau_1, \dots, \tau_k$ in A and the predicates (subsets) $P_1 \cdots P_m$ over A which are assigned to the variables and predicate names $t_1, \dots, t_k, X_1 \cdots X_m$. Hence, the notation will usually be

$$\langle M, \tau_1, \dots \tau_k; P_1 \cdots, P_m \rangle \models_{MLO} \varphi(t_1, \cdots t_k; X_1, \dots, X_m)$$

which we also abbreviate to

$$M \models \varphi[\tau_1, \dots, \tau_k; P_1, \dots, P_m]$$

or even to $M \models \varphi[\overline{\tau}, \overline{P}]$ where the bar denotes a tuple of appropriate length. When we define the semantics of a second order formula or when we deal with validity and satisfiability of a first order formula it is necessary to specify over which predicates should the variables X range. In *full MLO* they range over all unary predicates (i.e. – subsets).

A requirement that is often imposed in the literature is that in every bounded time interval a system can change its state only finitely many times. This requirement is called *finite variability* (or non-Zeno) requirement. We consider also finitely variability interpretation of second-order MLO. Under this interpretations monadic predicates range over predicates with finite variability. Observe that in the real model this property of a predicate X can be expresses by the pure first-order formula [Rab98]. It is worth noting that there is no first-order monadic formula that defines finite variability predicates over the rationals, however, the finite variability predicates over the rationals can be defined by a monadic second-order formula.

The syntax of Temporal Logic (TL) has in its vocabulary *Predicate variables* $X_1, X_2, \dots$ and some *modality names* with prescribed arity $O_1^{(k_1)} \cdots O_n^{(k_n)}$ (the arity notation is usually omitted). For example the "sometime in the future" modality $\Diamond$ has arity 1. The "_until_" modality has arity 2.

Atomic formulas are just variables X_i and *temporal formulas* are obtained from the atoms using Boolean connectives and applying the modalities: if $\varphi_1, \dots, \varphi_{k_i}$ are temporal formulas then so is $O^{(k_i)}(\varphi_1, \dots, \varphi_{k_i})$. As usual we write φ *until* ψ instead of *until* (φ, ψ).

Structures for TL are again linear orders $M = \langle A, < \rangle$. Every modality $O^{(k)}$ is interpreted in every structure M as an operator $O_M^{(k)} : [P(M)]^k \to P(M)$ which assigns "the set of points where $O[P_1, \dots, P_k]$ holds" to the k-tuple $\langle P_1, \dots, P_k \rangle$. We consider only modalities which are defined in MLO: we assume that for every modality $O^{(k)}$ there is a formula *(truth table)* $\overline{O}(t_0, X_1, \dots X_k)$ of MLO such that in every structure M:

$$O_M(P_1, \dots, P_k) = \{\tau \mid M \models_{MLO} \overline{O}[\tau; P_1, \dots, P_k]\} .$$

Example (Tables for Modalities)

- The modality $\diamondsuit X$, "X has happened before" is defined by $\varphi(t_0, X) \equiv \exists t < t_0\ X(t)$.
- The modality X *until* Y is defined by $\psi(t_0, X, Y) \equiv \exists t_1(t_0 < t_1 \wedge Y(t_1) \wedge \forall t(t_0 < t < t_1 \rightarrow X(t)))$.
- The modality X *since* Y is defined by $\psi(t_0, X, Y) \equiv \exists t_1(t_0 > t_1 \wedge Y(t_1) \wedge \forall t(t_1 < t < t_0 \rightarrow X(t)))$.

Satisfaction of a formula at a point τ in the model M is defined inductively starting with:

$$\langle M, \tau, P\rangle \models_{TL} X \quad \text{iff} \quad \tau \in P$$

and

$$\langle M, \tau, P_1, \ldots P_k\rangle \models_{TL} O(X_1, \ldots X_k) \quad \text{iff} \quad \langle M, \tau, P_1, \ldots P_k\rangle \models_{MLO} \overline{O}(t_0, X_1, \ldots X_k)\ .$$

This extends easily to:

Proposition. *For every formula $\varphi(X_1, \ldots X_n)$ of TL there is a formula $\overline{\varphi}(t_0, X_1, \ldots X_n)$ of MLO such that for every M, $\tau \in M$ and predicates $P, \ldots P_n$*

$$\langle M, \tau, P_1, \ldots P_n\rangle \models_{TL} \varphi \quad \text{iff} \quad \langle M, \tau, P_1, \ldots P_n\rangle \models_{MLO} \overline{\varphi}\ .$$

MLO supplies us with a yardstick by which to measure if a temporal logic (i.e. a choice of modalities) is as expressive as can be hoped for:

Definition. Let $\mathcal{C}$ be a class of structures and L a temporal logic. L is *expressively complete* with respect to $\mathcal{C}$ if every formula φ of MLO with single first order free variable t_0 is equivalent in $\mathcal{C}$ to some formula φ of the logic. A list of modalities $O_1^{(k_1)} \cdots O_n^{(k_n)}$ is *expressively complete* if the temporal logic with these modalities is expressibly complete.

There are natural choices for expressively complete sets of modalities:

Theorem 1. *a) ([Kamp68], reproved in [GPSS80]): The pair of modalities X until Y* and
X since Y is expressively complete for the canonical structures: the real line and the natural numbers.

b) [GHR94] There is a pair of modalities $X\,until_s\,Y$ and *$X\,since_s\,Y$ ("Stavi's modalities") which together with the since and until is expressively complete for the class of all linear orders.*

3 Quantitative Temporal Logic and Quantitative MLO

The logics MLO and TL are not suitable to deal with quantitative statements like "X will occur within one unit of time". This can be easily remedied in predicate logic by using the function $t + 1$:

Definition. MLO_1, the monadic logic of order with a $+1$ function is the monadic logic built from the primitive relation $<$ *and* one unary function symbol which we denote either by $S(t)$ or by $t+1$. The *standard (canonical) model* for this logic is the non-negative real line with the usual $+1$ function. *General structures* for MLO_1 are ordered structures $M = \langle A, <, 0, S\rangle$ with first element 0 and with a unary function S that satisfies some natural requirements: like strict monotonicity, non-archimedian property.

We will deal here only with the canonical model and leave out the axiomatization of more general models. We use the standard notation $t+1$ for $S(t), n$ for $0+1+\cdots+1$ (n times) and $t-1$ for 0 when $t<1$ and for the unique t_1 such that $t_1+1=t$ for $t \geq 1$.

MLO_1 is a broad language and everything that we do is inside MLO_1. But it is too strong: the problem of validity and satisfiability in the standard model is undecidable for MLO_1 (there is a natural encoding of Turing computations that shows it but it can also be deduced from the undecidability proof in [AFH96]). We shall, therefore, start at the other end: introduce the simplest quantitative modalities to TL and check what the corresponding fragment of MLO_1 is. We then check if the result is expressive enough and if it is decidable.

Definition: Quantitative Temporal Logic (QTL) is the temporal logic constructed from an expressively complete set of modalities for MLO *and* two new modalities $\diamondsuit_1 X$ and $\diamondsuit_{\overleftarrow{1}} X$ defined by the tables (in t_0):

$$\diamondsuit_1 X: \qquad \exists t((t_0 < t < t_0+1) \wedge X(t)) \tag{3}$$

$$\diamondsuit_{\overleftarrow{1}} X: \qquad \exists t((t_0-1 < t < t_0) \wedge X(t))\ . \tag{4}$$

Next we intend to identify the fragment of MLO_1 that corresponds to QTL. This fragment will use the function $t+1$ only in a very restricted form as indicated in (3) and (4). We introduce some syntactical sugar to MLO_1 – the "bounded quantifiers" $(\exists t)^{<t_0+1}_{>t_0}$ and $(\exists t)^{<t_0}_{>t_0-1}$ as follows: if φ is a formula of MLO_1 then we use the shorthand:

$$(\exists t)^{<t_0+1}_{>t_0}\varphi \equiv \exists t(t_0 < t < t_0+1 \wedge \varphi(t)) \tag{5}$$

$$(\exists t)^{<t_0}_{>t_0-1}\varphi \equiv \exists t(t_0-1 < t < t_0 \wedge \varphi(t)) \tag{6}$$

Definition. Quantitative Monadic Logic of Order (QMLO) is the fragment of MLO_1 which is built from the atomic formulas $t_1 < t_2, t_1 = t_2, X(t)$ (t, t_1, t_2 variables) using Boolean connectives, first order quantifiers and the following rule: if $\varphi(t)$ is a formula of QMLO *with t its only first order free variable* then $(\exists t)^{<t_0+1}_{>t_0}\varphi$ and $(\exists t)^{<t_0}_{>t_0-1}\varphi$ are formulas of QMLO.

The following observation characterizes the expressive power of QTL.

Theorem 2. *Let Γ be an expressive complete set of modalities (over the reals) for first-order MLO. For every formula p over the modalities $\{\Gamma, \Diamond_1, \overleftarrow{\Diamond}_1\}$ there is a formula $\phi(t_0)$ of QMLO effectively computable from p such that $\phi(t_0)$ is equivalent (over the reals) to p. For every formula $\phi(t_0)$ of QMLO there is a formula p over the modalities $\{\Gamma, \Diamond_1, \overleftarrow{\Diamond}_1\}$ effectively computable from $\phi(t_0)$ such that $\phi(t_0)$ is equivalent (over the reals) to p.*

Proof. Straightforward induction.

4 The Expressive Power of QMLO

4.1 General Bounded Quantifiers

At first glance the modalities $\overleftarrow{\Diamond}_1$ and $\Diamond_1$ may seem insufficient to express more general modalities like $\Diamond_{[5,7)}$ defined by the table $\exists t.\ t_0 + 5 \leq t < t_0 + 7 \wedge X(t)$. However this is not the case.

We shall first show that one can use more general quantifiers in QMLO: $(\exists t)^{<t_0+n+m}_{>t_0+n}$, $(\exists t)^{<\infty}_{>t_0+n}$ and quantifiers with weak inequality replacing the strict inequality in one or both ends of the interval, where n is an integer and m is a positive natural number.

(a) $(\exists t)^{<t_0+1}_{\geq t_0} X(t) \equiv X(t_0) \vee (\exists t)^{<t_0+1}_{>t_0} X(t)$

(b) $(\exists t)^{\leq t_0+1}_{>t_0} X(t) \equiv (\exists t)^{<t_0+1}_{>t_0} X(t) \vee [\text{First } (t_0, X) \wedge (\forall t_1)^{<t_0+1}_{>t_0} (\exists t)^{<t_1+1}_{>t_1} X(t)]$

when First (t_0, X) says that there is a first point past t_0 for which $X(t)$ holds:

$$\text{First } (t_0, X) \equiv \exists t_1[t_0 < t_1 \wedge X(t_1) \wedge \forall t(t_0 < t < t_1 \rightarrow \neg X(t))] \tag{7}$$

it is not difficult to see that once a first solution to $X(t)$ is granted then the last conjunct in (b) is equivalent to $X(t_0+1)$ (note, however, that $X(t_0+1)$ by itself is not definable in QMLO and its addition would lead to undecidability).

Hence quantifiers $(\exists t)^{<t_0+1}_{\geq t_0}$ and $(\exists t)^{\leq t_0+1}_{>t_0}$ are definable in QMLO.

Let us list some more laws

(c) $(\exists t)^{<\infty}_{>t_0+1} X(t) \equiv (\forall t)^{\leq t_0+1}_{>t_0} \exists t_1 . t_1 > t \wedge X(t_1)$

(d) $(\forall t)^{<t_0+n+1}_{>t_0+n} X(t) \equiv (\forall t)^{<t_0+n}_{>t_0+n-1} (\exists t_2)^{<t+1}_{>t} (\forall t_1)^{<t_2+1}_{>t_2} X(t_1)$ for $n > 0$.

Theorem 3. *The extension L of QMLO by the following rules is expressive equivalent to QMLO over the canonical model.*

if $\phi(t)$ is an L formula with the only free variable t then the following are L formulae:

1. *$(\exists t)^{<t_0+n+m}_{>t_0+n} \phi(t)$, where n is an integer and m a positive natural number.*
2. *$(\exists t > t_0 + n)\phi(t)$ (denoted also by $(\exists t)^{<\infty}_{>t_0+n}\phi(t)$) and $(\exists t < t_0 + n)\phi(t)$ (denoted by $(\exists t)^{<t_0+n}_{>-\infty}\phi(t)$), where n is an integer.*
3. *Similar to (1) or (2) above with weak inequality replacing one or both occurrences of the strong inequality.*

Henceforth, we will freely use these generalized quantifiers which are definable in QMLO and also the corresponding modalities like $\Box_{\leq n}$ which is defined by $(\forall t)^{\leq t_0+n}_{\geq t_0} X_1(t)$ and $\Diamond_{[n,m]}$ which is defined by $(\exists t)^{\leq t_0+m}_{\geq t_0+n} X_1(t)$.

4.2 Modalities in Real Time Logics

Here are the definitions of some modalities that were used before:
(e) Pnueli's $\text{Age}_1(X)$ modality is dual to $\diamondsuit_1$

$$\text{Age}_1(X) \equiv (\forall t)^{<t_0}_{>t_0-1} X(t)$$

(f) Wilke's relative distance construct "the first time $X(t)$ occurs after t_0 is in distance smaller than larger than or equal to n"[Wilke94]

$$d_{<n}(X) \quad \text{is just} \quad \text{First}(t_0, X) \wedge (\exists t)^{\leq t_0+n}_{>t_0} X(t)$$

$$d_{=n}(X) \quad \text{is} \quad (\exists t)^{\leq t_0+n}_{>t_0} X(t) \wedge (\forall t)^{\leq t_0+n}_{>t_0} \neg X(t)$$

(g) The $\triangleright$ modality of [HRS98]: $\triangleright_{(n,n+m)} X$ "there is a first instance of $X(t)$ among the points in the interval $(t_0 + n, t_0 + n + m)$". For $n = 0$ this is $\text{First}(t_0, X) \wedge (\exists t)^{<t_0+m}_{>t_0} X(t)$. For $n > 0$ this is $[(\forall t_1)^{<t_0+n}_{>t_0+n-1} (\exists t)^{<t_1+1}_{>t_1} \text{First}(t, X)] \wedge (\exists t)^{<t_0+n+m}_{>t_0+n} X(t)$.

The logic MITL [AFH96] is based on an infinite set of modalities $until_I$ where I is a non-singular interval with integer endpoints - these modalities are called constrained *until* modalities. The MLO_1 truth table for example for the formula $X\ until_{[5,8)}\ Y$ is $\exists t. t_0 + 5 \leq t < t_0 + 8 \wedge Y(t) \wedge (\forall t\ .\ t_0 < t\ < t \rightarrow X(t\ ))$.

It is shown in [AFH96] that the operators like our $\diamondsuit_1$ are definable in this logic. Unfortunately the fundamental operator $\diamondsuit_1$ is not discussed there and in fact we can show that it cannot be expressed using constrained $until_I$ modalities.

What is called MITL in [HRS98] is the logic based on $until_I$ operators and the since operators $since_I$. The modality $\diamondsuit_1$ is easily expressed using these since operators.

In the sequel we will refer only to the version with both until and since operators.

(h) $\text{until}_{(n,n+m)}$ can be defined as $P\ \text{until}_{(n,n+m)} Q \equiv \Box_{(0,n]}(P \wedge (P\ until\ Q)) \wedge \diamondsuit_{(n,m)} Q$.

Similar definitions work for since and for half open or closed intervals. Hence, MITL and QTL are expressive equivalent.

5 Decidability

We want to show that there is an algorithm which given a formula $\varphi(\overline{Z})$ of QTL determines if φ is valid in $\langle R^+, 0, <, +1\rangle$. We prove the equivalent claim that there is an algorithm which given ψ of QMLO determines if ψ is satisfiable in $\langle R^+, 0, <, +1\rangle$.

Notation. For every n we define the formula:

$$\text{Timer}_n(X_1, \ldots X_n, Y_1, \cdots Y_n) \equiv \bigwedge_i \forall t (Y_i(t) \longleftrightarrow (\forall t_1)^{<t}_{>t-1} X_i(t_1))$$

i.e. each Y_i is a timer that measures if X_i persisted for at least one unit of time.

Theorem 4 (Timer Normal form). *There is an algorithm which associates with any formula $\varphi(t_0, \overline{Z})$ of QMLO variables $X_1, \ldots, X_n, Y_1, \ldots, Y_n$ and a formula $\overline{\varphi}(t_0, X_1, \ldots, X_n, Y_1, \ldots, Y_n, \overline{Z})$ of pure MLO such that $\varphi(t_0, \overline{Z})$ is satisfiable iff the following formula is satisfiable:*

$$\mathrm{Timer}_n(X_1, \ldots, X_n, Y_1, \ldots, Y_n) \wedge \overline{\varphi}(t_0, \overline{X}, \overline{Y}, \overline{Z})$$

The proof is by routine normalization (note that the two formulas are equivalent only with respect to satisfiability).

Definition A formula is said to be in first (second) order **timer normal form** if it has the form

$$\mathrm{Timer}_n(X_1, \ldots, X_n, Y_1, \ldots, Y_n) \wedge \phi,$$

where ϕ is a first (second) order monadic formula.

Next we associate with the formula $\mathrm{Timer}_n(X_1, \ldots X_n, Y_1, \cdots Y_n)$ a formula $\overline{\mathrm{Timer}}_n(X_1, \ldots, X_n, Y_1, \ldots Y_n)$ in pure monadic logic. It is the conjunction of the properties A_i, $B_{i,j}$, C_i below and some technical properties D_i which we omit. We introduce first the following notation: *the duration of X at t_0* is the largest interval (t_1, t_0) such that $X(t)$ holds for every point t in the interval.

A_i : "If $Y_i(t_0)$ holds then the duration of X_i at t_0 is not empty.

$B_{i,j}$:"If $Y_i(t_0)$ holds and if the duration of X_j at t_0 is at least as big as the duration of X_i then $Y_j(t_0)$ holds. Moreover; if the duration of X_j at t_0 is strictly larger than the duration of X_i then $Y_j(t)$ started to hold already at a point before t_0.

C_i : "Y_i has finite variability. This is expressible in R by the pure first order monadic formula that says: Every point is the left end of some open interval on which Y_i does not change its value and every point except 0 is the right end of an open interval on which Y_i does not change its value.

D_i deals with the behavior of Y_i at 0 and when approaching infinity. It also asserts that Y_i holds on a topologically closed set. The definition will appear in the full version of the paper.

The formula Timer of QMLO and the formula $\overline{\mathrm{Timer}}$ of MLO are related by the following main Lemma:

Lemma 5 (Reduction of Quantitative Properties to Pure Monadic Properties). *The predicates $P_1, \ldots, P_n Q_1, \ldots, Q_n$ over $\langle R^+, 0, < \rangle$ satisfy the formula $\overline{\mathrm{Timer}}_n$ if and only if there is an order preserving bijection $\rho : R^+ \to R^+$ such that $P_1\rho, \ldots, P_n\rho,\ Q_1\rho, \ldots, Q_n\rho$ satisfy Timer_n.*

In the above lemma the predicate $P_i\rho$ is obtained from the predicate P_i by stretching according to the bijection ρ, i.e $(P\rho)(\tau)$ iff $P(\rho(\tau))$. Observe that P satisfies a monadic formula $\phi(X)$ iff $P\rho$ satisfies $\phi(X)$. Therefore,

Corollary 6. *For every first-order or second-order monadic formula ϕ $\overline{\mathrm{Timer}}_n \wedge \phi$ is satisfiable iff $\mathrm{Timer}_n \wedge \phi$ is satisfiable.*

We recall

Theorem 7. *1. The satisfiability of monadic first order logic of order over the reals is decidable [BG85].*
2. The satisfiability of monadic second-order logic of order over the reals is undecidable [She75]
3. The satisfiability of monadic second order logic over the real finitely variable interpretation is decidable [Rab98].

Therefore,

Corollary 8. *1. The satisfiability of the formulas in first-order timer normal form in the canonical model is decidable.*
2. The satisfiability of the formulas in the second-order timer normal form under the finite variability interpretation is decidable.

Theorem 9. *Satisfiability in the canonical model is decidable for QMLO and for QTL. Satisfiability in the finite variability canonical model is decidable for QMLO and for QTL.*

6 On Complexity of QTL

In this section we show somewhat informally how one can derive easily the complexity bounds without appealing to automata theoretical techniques.

We observe that the monadic formula $\overline{Timer}$ (see section 5) is expressible in Temporal Logic.

Lemma 10. *There is a TL formula* $timer_n(X_1, \ldots X_n, Y_1, \ldots Y_n)$ *of size* $O(n^2)$ *over the modalities Until and Since such that* $\overline{Timer}_n(X_1, \ldots X_n, Y_1, \ldots Y_n)$ *is equivalent over the canonical model to* $timer_n(X_1, \ldots X_n, Y_1, \ldots Y_n)$.

Proof. Just observe that every clause A_i, $B_{i,j}, C_i$ and D_i in the definition of $\overline{Timer_n}$ (see section 5) is expressible by a temporal logic formula which is independent of n.

Unnesting: We define a process of unnesting by a generic example. Let $Always(X)$ be the modality defined by the table $t_0 = t_0 \wedge \forall t.\ X(t)$. Let OP_1 and OP_2 be any two-place modalities over a linear order A. It is easy to see that that the formulas $OP_1(\neg X_1, OP_2(X_2, X_3 \vee X_4)$ is unsatisfiable over A iff the conjunction of the following formulas is unsatisfiable over A

1. Z
2. $Always(Z \leftrightarrow OP_1(Z_1, Z_2))$
3. $Always(Z_1 \leftrightarrow \neg X_1)$
4. $Always(Z_2 \leftrightarrow OP_2(X_2, Z_3,))$
5. $Always(Z_3 \leftrightarrow (X_3 \vee X_4))$

A simple generalization of this observation shows

Lemma 11. *For every QTL formula ϕ with k metrical quantifiers there is a TL formula ψ and a subset F of $\{1 \ldots k\}$ such that the size of ψ is linear in the size of ϕ and ϕ is satisfiable if and only if the conjunction of ψ with $\bigwedge_{i \in F} Always(X_i \leftrightarrow \Diamond_1 Z_i) \wedge \bigwedge_{i \in \{1,\ldots,k\} \setminus F} Always(X_i \leftrightarrow \Diamond Z_i)$ is satisfiable.*

Lemma 12. *There is a formula over $\{until,\ since,\ \Diamond_1\}$ which is equivalent to $Always(X_i \leftrightarrow \Diamond_1 Z_i)$.*

Lemma 13. *The formula $Always(X_i \leftrightarrow \Diamond_1 Z_i)$ is equivalent to $Timer_1(\neg X_i, \neg Z_i)$.*

The above lemmas and Lemma 5 imply

Theorem 14. *There is a polynomial time algorithm that for every QTL formula ϕ of size m with k metrical modalities constructs a TL formula ψ of size $O(k^2 \times m)$ such that (1) ψ is satisfiable in the canonical model iff ϕ is satisfiable in the canonical model. (2) ψ is satisfiable in the finite variability canonical model iff ϕ is satisfiable in the finite variability canonical model.*

Theorem 15. *[Rab98a] Let Γ be any finite set of modalities definable in monadic first-order logic of order. The satisfiability problem for TL formulas over Γ in the finite variability canonical model is in PSPACE.*

We note here that the proof of the above theorem is automata free and relies on the compositional method.

Open Question: What is the complexity of the satisfiability for TL formulas in the general canonical model.

Corollary 16. *The satisfiability problem for QTL in the finite variability canonical model is in PSPACE.*

Let ϕ be an MITL formula with m boolean and temporal connectives and let c be the largest constant that appears as a subscript of the bounded until or since operator in ϕ. From the laws of Section 4 and an unnesting procedure one can construct a QTL formula ψ of size $O(m \times c)$ such that ψ is satisfiable (in either of the canonical or finite variability canonical models) iff ϕ is satisfiable in that model. Hence, we reprove the result of [AFH96] that satisfiability for MITL is decidable in space polynomial in $m \times c$.

7 About Two Popular Real Time Models

In this paper we are dealing with the canonical model for QTL or with the finitely variable canonical model (in this model predicates are restricted to finitely variable predicates). In the survey [AH92] sixteen models for real time logics are provided. Surprisingly, neither the canonical model nor the finite variability canonical model is among these models.

All the models considered in the literature are ω-sequences of something. Probably because the only technique to show the decidability was the reduction to Alur-Dill timed automata which in turn are reduced to ω-automata. Our proof of decidability is automata free and works for general predicates as well as finitely variable predicates.

We discuss two of these model that gained popularity.

7.1 The State Sequence Model

The most popular model is the *state sequence model*. A state sequence is an ω-sequence $(t_0, \sigma_0), \ldots, (t_n, \sigma_n), \ldots$ where t_i is an unbounded increasing sequence of reals and σ_i gives the values of the monadic predicates at t_i.

The state sequence model does not faithfully reflect the real line. For example the very basic law $\diamondsuit_2 X \leftrightarrow \diamondsuit_1 X \vee \diamondsuit_1 \diamondsuit_1 X$ fails in this model. In fact already at the pure temporal logic level (without the metric) it is clear that sequences can not represent faithfully the canonical model as any formula is satisfiable over real sequences iff it is satisfiable over the discrete model of the natural numbers. Clearly we expect TL to differentiate between the discrete and the continuous model of time. another price paid for the use of state sequences is the proliferation of modal operators; while connectives like $\diamondsuit_{[m,n]}$ can be defined in the canonical model using the connective $\diamondsuit_1$ this is not true for state sequences. Most of the laws from section 4 fail in this model. For example $\diamondsuit_{(0,1]}$ is not expressible from $\diamondsuit_1$.

A remarkable work that achieves impressive results using state sequences is Wilke's [Wilke94]. He introduced metrical properties into second order monadic logic and proved the decidability of an extensive fragment. He embedded all the known decidable formalisms (and much more) into this fragment thus reproving uniformly the decidability.

7.2 The Trace Interval Model

Another popular model is trace interval model. A trace interval is an ω-sequence $(I_0, \sigma_0), \ldots (I_n, \sigma_n) \ldots$ where I_i are disjoint intervals that cover R^+ and I_i precedes I_{i+1}.

A trace interval represents a finite variability structure for MLO. Indeed, every $t \in R^+$ is in exactly one of the intervals I_j and σ_j determines the value of the monadic predicates at t. This clearly defines a finite variability model. However, the same finite variability model has many distinct representations. So the formalisms that distinguish between representations of the same model are unreasonable.

Even though a trace interval encodes all the information about the corresponding finite variability canonical model it may still be misleading if one tries to use it for a different model like the rationals Q. In the words of [AFH96] "MITL cannot distinguish the time domain of the reals from the time domain of the rationals". In fact theorem 2.4.1.4 in [AFH96] says that an MITL formula ϕ is satisfiable in the rational interval traces iff it is satisfiable in the real interval traces. (By the rational interval trace it is meant there an ω-sequence $(I_0, \sigma_0), \ldots (I_n, \sigma_n) \ldots$ where I_i are disjoint intervals of the rationals with the rationals end-points that cover Q^+ and I_i precedes I_{i+1}.)

But the following TL formula is satisfiable in the reals and not in the rationals.

$$(X \wedge \diamondsuit \neg X) \wedge \Box(\neg X \rightarrow \Box \neg X) \wedge ((X \rightarrow \diamondsuit X) \wedge (\neg X \rightarrow \diamondsuit \neg X))$$

(By the first two clauses X holds for a prefix of the model and $\neg X$ holds in a suffix of the model; by the third clause X has no maximal element and $\neg X$ has no minimal element). Does theorem 2.4.1.4 in [AFH96] say that MITL is two weak to express the above formula. Of course not; it only shows that the rationals are not modeled adequately by the rational trace intervals.

8 Conclusion

We believe that we proved the case for the quantitative logic QTL and its predicate logic twin QMLO.

It is easy to modify the theory to deal with more general time lines. For the rational line Q we would need to require that the time unit is Archimedian: the sequence $0, 0+1, 0+1+1, \ldots$ must be unbounded for every $+1$ function. A more general approach will replace the $+1$ function by a relation between pairs of points in time "t_2 is_not_too_far_ahead_of t_1" such that some obvious axioms hold.

Future research must look for *natural* stronger modalities. A. Pnueli noticed that the following modality

$$\varphi(t_0, X, Y) \equiv (\exists t_1 t_2)[(t_0 < t_1 < t_2 < t_0 + 1) \wedge X(t_1) \wedge Y(t_2)]$$

is not definable in QMLO [HR99]. We can show that one may add modalities of the following form without losing decidability

$$(\exists t_1 \cdots t_n)[(t_0 < t_1 < \cdots < t_n < t_0 + 1) \wedge \bigwedge_i \varphi_i(t_i)]$$

(note that φ_i speaks only of t_i). But it is difficult to see from this what should be *the* next modality in the logic.

Acknowledgments

We are indebted to our colleague and teacher B. Trakhtenbrot. His penetrating insight led us through this work. In particular our timer normal form is an adaptation of his "oracle normal form". We would like to thank J.F. Reskin for helpful comments regarding Theorem 3.

References

[AD94] R. Alur, D. Dill A theory of timed Automata. Theoretical Computer Science 126 (1994), 183-235.

[AFH96] R. Alur, T. Feder, T.A. Henzinger. The Benefits of Relaxing Punctuality. Journal of the ACM 43 (1996) 116-146.

[AH92] R. Alur, T.A. Henzinger. Logics and Models of Real Time: a survey. In Real Time: Theory and Practice. Editors de Bakker et al. LNCS 600 (1992) 74-106.

[AH92a] R. Alur, T.A. Henzinger. Back to the future: toward theory of timed regular languages. In 33th FOCS, pp177-186, 1992.

[AH93] R. Alur, T.A. Henzinger. Real Time Logic: Complexity and Expressiveness. Information and Computation 104 (1993) 35-77.

[BL95] A. Bouajjani and Y. Lakhnech. Temporal Logic +Timed Automata: Expressiveness and Decidability. CONCUR95, LNCS 962, pp. 531-547, 1995.

[BG85] J.P. Burges, Y. Gurevich. The Decision Problem for Temporal Logic. Notre Dame J. Formal Logic, 26 (1985) 115-128.

[GHR94] D.M. Gabbay, I. Hodkinson, M. Reynolds. Temporal Logics volume 1. Clarendon Press, Oxford (1994).

[GPSS80] D.M. Gabbay, A. Pnueli, S. Shelah, J. Stavi. On the Temporal Analysis of Fairness. 7th ACM Symposium on Principles of Programming Languages. Las Vegas (1980) 163-173.

[HRS98] T.A. Henzinger, J.F. Raskin, P.Y. Schobbens. The Regular Real Time Languages. 25th ICALP Colloquium (1998).

[HR99] Y. Hirshfeld and A. Rabinovich, A Framework for Decidable Metrical Logics. To appear in ICALP99, LNCS vol.1644, 1999.

[Kamp68] H. Kamp. Tense Logic and the Theory of Linear Order. Ph.D. thesis, University of California L.A. (1968).

[Rab98] A. Rabinovich. On the Decidability of Continuous Time Specification Formalisms. In Journal of Logic and Computation, pp 669-678, 1998.

[Rab98a] A. Rabinovich. On the complexity of TL over the reals. Manuscript 1998.

[She75] S. Shelah. The monadic theory of order. *Ann. of Math.*, **102**, pp 349-419, 1975.

[Trak99] B.A. Trakhtenbrot. Dynamic Systems and their interaction: Definitional Suggestions. TR Tel Aviv University, 1999.

[Wilke94] T. Wilke. Specifying Time State Sequences in Powerful Decidable Logics and Time Automata. In Formal Techniques in Real Time and Fault Tolerance Systems. LNCS 863 (1994), 694-715.

An Expressively Complete Temporal Logic without Past Tense Operators for Mazurkiewicz Traces

Volker Diekert[1] and Paul Gastin[2]

[1] Inst. für Informatik, Universität Stuttgart, Breitwiesenstr. 20-22, D-70565 Stuttgart
[2] LIAFA, Université Paris 7, 2, place Jussieu, F-75251 Paris Cedex 05, France
diekert@informatik.uni-stuttgart.de Paul.Gastin@liafa.jussieu.fr

Abstract. Mazurkiewicz traces are a widely accepted model of concurrent systems. We introduce a linear time temporal logic $\mathrm{LTL}_{\mathrm{f}}$ which has the same expressive power as the first order theory FO(<) of finite (infinite resp.) traces. The main contribution of the paper is that we only use future tense modalities in order to obtain expressive completeness. Our proof is direct using no reduction to words and Kamp's theorem for both finite and infinite words becomes a corollary. This direct approach became possible due to a proof technique of Wilke developed for the case of finite words.
Keywords Temporal logics, Mazurkiewicz traces, concurrency

1 Introduction

The verification of programs is essential for the conception of critical systems, especially for concurrent systems. The model checking approach starts with the abstraction of the actual system into some automata based model. Then the specification to be checked is expressed in some suitable logic, mainly temporal logics. Finally, a tool (model checker) is used to determine whether the system (the automaton) meets its specification (satisfies the formula).
Usually, concurrent systems are reduced to sequential ones by considering all possible linearizations of concurrent behaviors. Then, one can use both the temporal logics for sequences and the existing tools to specify and verify properties of systems. The main problem with this approach is the state explosion induced by these many linearizations. An alternative approach is to use truly concurrent models such as Mazurkiewicz traces and to introduce and study logics over these traces. Work along this line can be found in [1,12,13,14,16,19,20]. See also [3] for the general background of trace theory, and in particular [15] for traces and logic and [8] for infinite traces.
The various linear time temporal logics differ by the kind of modalities allowed: future modalities (next, eventually, until, ...) and past modalities (previous, since, ...). For sequential systems, a crucial result states that linear time temporal logics are expressively complete, i.e., have the same expressive power as the first order theory of words FO(<). This is known now as Kamp's theorem,

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 188–203, 1999.

[9]. The result that we can avoid past tense operators was shown, however, much later; it relies on Gabbay's separation theorem [6].

Since then simplified versions of the proof of Kamp's theorem were found. Let us mention [2], which contains an elegant proof based on Krohn-Rhodes decomposition. The most recent development is due to Wilke [23] who gave an elementary proof based on the classical fact that FO languages are aperiodic and hence given by some counter–free automaton [10,17]. Our approach follows the same lines as Wilke's proof for the corresponding result over finite words.

Kamp's theorem has been generalized by Ebinger [4] to Mazurkiewicz traces, but he used a temporal logic with both past and future modalities, and the proof failed for infinite traces. Then, Thiagarajan and Walukiewicz [21] have introduced a temporal logic LTrL for traces with the usual future modalities and also past tense modalities in the weak form of *previous* constants. They proved (via a reduction to the word case) that this logic is expressively complete, both for finite and infinite traces. It is open whether the fragment of LTrL without the previous constants is still expressively complete. A positive answer to this question was claimed for finite traces in [11], but the proof contained a serious flaw, which has not yet been fixed.

In the present paper we work with a linear time temporal logic for traces having future modalities only. In addition to the classical *next* and *until* modalities, we introduce a new (but natural) operator $\langle B\ \rangle\varphi$ which means that we can reach a configuration satisfying φ by using actions from some given set B only. We can think that the operator works as a *filter* which is reflected in the index f in the notation $\mathrm{LTL_f}$. In the word case, $\langle B\ \rangle\varphi$ means nothing but $\langle B\rangle\top\,\mathcal{U}\,\varphi$ and we can read $\langle B\ \rangle\varphi$ as a simple macro. Hence over words $\mathrm{LTL_f}$ becomes the classical linear time logic LTL. From this we may deduce that the new operator $\langle B\ \rangle\varphi$ can be avoided in the case of direct products of free monoids, but up to now we do not know whether this is possible for traces, in general.

We prove that $\mathrm{LTL_f}$ has the same expressive power as the first order theory of traces FO($<$). The result holds for both finite and infinite traces. This solves an open problem of [4,21] in the sense that we have a temporal logic using future modalities only. We would like to stress that, contrary to previous works, we are not reducing the problem to the word case. We give a direct proof for finite and infinite traces such that, formally, the result for words becomes a corollary.

The hard part of the proof is to find an $\mathrm{LTL_f}(\Sigma)$ formula which is equivalent to some given FO($<$) formula. For this, we use the equivalence between first–order and aperiodic trace languages [5].

All our constructions are effective, and logics like LTrL or our $\mathrm{LTL_f}$ are clearly decidable, but apparently rather complex. The satisfiability problem of FO($<$) is non-elementary in the word and in the trace case, whereas the satisfiability problem of linear time temporal logic is PSPACE-complete in the word case [18]. In the presence of concurrency the situation is even more complex. Walukiewicz [22] has shown that the satisfiability problem for the fragment without previous constants of LTrL is non-elementary over an independence alphabet with four letters. Hence the same statement holds for $\mathrm{LTL_f}$.

2 Preliminaries

Throughout the first part we speak of finite traces only. The infinitary case is treated in Sect. 4. The separation will make it necessary to repeat some parts, which could be treated uniformly otherwise. We have decided to do so for at least two reasons: The result for finite traces is used in order to obtain the corresponding result for infinite traces. Second, the subject is technically involved. So we prefer to postpone the notions needed to deal with infinite traces as long as possible hoping that the first part becomes then more easily accessible.

By (Σ, I) we mean a finite *independence alphabet* where Σ denotes a finite alphabet and $I \subseteq \Sigma \times \Sigma$ is an irreflexive and symmetric relation called the *independence relation*. The complementary relation $D = (\Sigma \times \Sigma) \setminus I$ is called the *dependence relation*. The monoid of *finite traces* $\mathbb{M}(\Sigma, I)$ is defined as a quotient monoid with respect to the congruence relation induced by I, i.e., $\mathbb{M}(\Sigma, I) = \Sigma^* / \{\, ab = ba \mid (a,b) \in I \,\}$. We also write $\mathbb{M}$ instead of $\mathbb{M}(\Sigma, I)$. For $A \subseteq \Sigma$ we denote by $\mathbb{M}_A$ the submonoid of $\mathbb{M}(\Sigma, I)$ generated by A:

$$\mathbb{M}_A = \mathbb{M}(A, I \cap A \times A) = \{x \in \mathbb{M}(\Sigma, I) \mid \mathrm{alph}(x) \subseteq A\}.$$

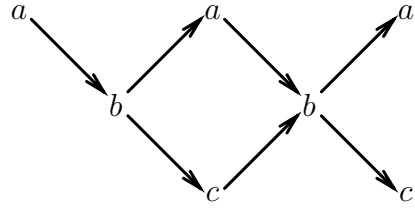
A trace $x \in \mathbb{M}$ is given by a congruence class of a word $a_1 \cdots a_n \in \Sigma^*$ where $a_i \in \Sigma$, $1 \le n$. By abuse of language, but for simplicity we denote a trace x by one of its representing words $a_1 \cdots a_n$. The number n is called the length of x, it is denoted by $|x|$. For $n = 0$ we obtain the *empty* trace, it is denoted by 1. The *alphabet* $\mathrm{alph}(x)$ of a trace x is the set of letters occurring in x. A *trace language* is a subset $L \subseteq \mathbb{M}$. The concatenation is defined as usual:

$$KL = \{xy \in \mathbb{M} \mid x \in K, y \in L\}.$$

The Kleene–star L^* refers to the submonoid of $\mathbb{M}$ which is generated by the set L. We have $L^* = \bigcup_{i \ge 0} L^i$ where $L^0 = \{1\}$ and L^i is the i-fold iteration of L with itself, $i \ge 0$. For $A \subseteq \Sigma$ we have $A^* = \{x \in \mathbb{M} \mid \mathrm{alph}(x) \subseteq A\}$, which is ambiguous, because A^* could also denote the free monoid generated by the set A. It will be clear from the context what we mean.

Every trace $a_1 \cdots a_n \in \mathbb{M}$ can be identified with its dependence graph. This is (an isomorphism class of) a node–labeled, acyclic, directed graph $[V, E, \lambda]$, where $V = \{1, \ldots, n\}$ is a set of vertices, each $i \in V$ is labeled by $\lambda(i) = a_i$, and there is an edge $(i,j) \in E$ if and only if both $i < j$ and $(\lambda(i), \lambda(j)) \in D$. In pictures it is common to draw the Hasse diagram only. Thus, all redundant edges are omitted.

Example 1. Let $(\Sigma, D) = a \text{——} b \text{——} c$, i.e., $I = \{(a,c), (c,a)\}$. Then the trace $x = abcabca$ is given by



By $\min(x)$ and $\max(x)$ we refer to the *minimal* and *maximal letters* in the dependence graph. In the example above $\min(x) = \{a\}$ and $\max(x) = \{a, c\}$. Formally:

$$\min(x) = \{a \in \Sigma \mid x \in a\mathbb{M}\},$$
$$\max(x) = \{a \in \Sigma \mid x \in \mathbb{M}a\}.$$

For $B, C \subseteq \Sigma$ and $\# \in \{\subseteq, =, \supseteq, \neq\}$ we define:

$$I(B) = \{a \in \Sigma \mid \forall b \in B : (a, b) \in I\},$$
$$D(B) = \{a \in \Sigma \mid \exists b \in B : (a, b) \in D\},$$
$$(\mathrm{Min} \# B) = \{x \in \mathbb{M} \mid \min(x) \# B\},$$
$$(\mathrm{Max} \# B) = \{x \in \mathbb{M} \mid \max(x) \# B\},$$
$$[B, C] = \{x \in \mathbb{M} \mid \mathrm{alph}(x) = B,\ \max(x) \subseteq C\}.$$

If in the notations above B (or C) is a singleton, then we usually omit braces and write e.g. $I(b)$ or $\mathrm{Max} = b$.

A trace language of the form $K = B_1 b_1 \cdots B_k b_k$ with $k \geq 0$, is called a *max–filter*, if $B_i = \{b_i, \ldots, b_k\}$ for $1 \leq i \leq k$. In this case, we have $\max(x) = \max(b_1 \cdots b_k)$ for all $x \in K$. The languages above $(\mathrm{Max} \# B)$ and $[B, C]$ are finite unions of max–filters. For example, $[B, C]$ is the finite union over all max–filters $B_1 b_1 \cdots B_k b_k$ such that $B = \{b_1, \ldots, b_k\}$ and $\mathrm{Max}(b_1 \cdots b_k) \subseteq C$.

The syntax of the temporal logic $\mathrm{LTL_f}(\Sigma)$ is defined as follows. There are a constant symbol $\bot$ representing *false*, the logical connectives $\neg$ (not) and $\vee$ (or), for each $a \in \Sigma$ a unary operator $\langle a \rangle$ called *next-a*, for each $B \subseteq \Sigma$ a unary operator $\langle B\ \rangle$ called *B-filter*, and a binary operator $\mathcal{U}$ called *until*. Thus, the syntax is given by:

$$\varphi ::= \bot \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle a \rangle \varphi \mid \langle B\ \rangle \varphi \mid \varphi\, \mathcal{U}\, \varphi,$$

where $a \in \Sigma$ and $B \subseteq \Sigma$.

Usually, the semantics is defined by saying when some formula φ is satisfied by some trace z at some configuration (i.e., prefix) x; hence by defining $(z, x) \models \varphi$. Since our temporal logic uses future modalities only, we have $(z, x) \models \varphi$ if and only if $(y, 1) \models \varphi$, where y is the unique trace satisfying $z = xy$. Therefore, we do not need to introduce configurations and it is enough to say when a trace satisfies a formula at the empty configuration, denoted simply by $z \models \varphi$. This is done inductively on the formula as follows:

$z \not\models \bot$,
$z \models \neg\varphi$ if $z \not\models \varphi$,
$z \models \varphi \vee \psi$ if $z \models \varphi$ or $z \models \psi$,
$z \models \langle a \rangle \varphi$ if $z = ay$ and $y \models \varphi$,
$z \models \langle B\ \rangle \varphi$ if $z = xy$, $x \in \mathbb{M}_B$, and $y \models \varphi$,
$z \models \varphi\, \mathcal{U}\, \psi$ if $z = xy$, $y \in \mathbb{M}$, $y \models \psi$, and $x = x\,x\ ,\ x\ \neq 1$ implies $x\ \ y \models \varphi$.

As usual, we define $\mathrm{L}_{\mathbb{M}}(\varphi) = \{x \in \mathbb{M} \mid x \models \varphi\}$. We say that a trace language $L \subseteq \mathbb{M}$ is *expressible in* $\mathrm{LTL_f}(\Sigma)$, if there exists a formula $\varphi \in \mathrm{LTL_f}(\Sigma)$ such that $L = \mathrm{L}_{\mathbb{M}}(\varphi)$.

Equivalently, we can define inductively the language $\mathrm{L}_{\mathbb{M}}(\varphi)$ as follows:

$$\begin{aligned} \mathrm{L}_{\mathbb{M}}(\bot) &= \emptyset & \mathrm{L}_{\mathbb{M}}(\langle a\rangle\varphi) &= a\mathrm{L}_{\mathbb{M}}(\varphi) \\ \mathrm{L}_{\mathbb{M}}(\neg\varphi) &= \mathbb{M}\setminus \mathrm{L}_{\mathbb{M}}(\varphi) & \mathrm{L}_{\mathbb{M}}(\langle B\ \rangle\varphi) &= \mathbb{M}_B\mathrm{L}_{\mathbb{M}}(\varphi) \\ \mathrm{L}_{\mathbb{M}}(\varphi\vee\psi) &= \mathrm{L}_{\mathbb{M}}(\varphi)\cup\mathrm{L}_{\mathbb{M}}(\psi) & \mathrm{L}_{\mathbb{M}}(\varphi\,\mathcal{U}\,\psi) &= \mathrm{L}_{\mathbb{M}}(\varphi)\,\mathcal{U}\,\mathrm{L}_{\mathbb{M}}(\psi) \end{aligned}$$

where the *until* operator $\mathcal{U}$ is defined on trace languages by

$$L\,\mathcal{U}\,K = \{xy \mid x\in\mathbb{M},\ y\in K,\ \text{and } x = x\ x\ \ , x\ \ \neq 1 \text{ implies } x\ \ y\in L\}.$$

As an easy exercise let us state the following lemma, which will be used frequently.

Lemma 1. *Let $L\subseteq\mathbb{M}$ be expressible in $\mathrm{LTL}_{\mathrm{f}}(\Sigma)$ and $B,C\subseteq\Sigma$. Then the languages* $(\mathrm{Min}\ \#\ B)$, $(\mathrm{Max}\ \#\ B)$, $[B,C]$, *and* $[B,C]L$ *are expressible in* $\mathrm{LTL}_{\mathrm{f}}(\Sigma)$.

The following operators are standard abbreviations. They serve as macros.

$$\begin{aligned} \top &:= \neg\bot & &\textit{true}, \\ \langle B\rangle\varphi &:= \textstyle\bigvee_{b\ \ B}\langle b\rangle\varphi & &\text{for } B\subseteq\Sigma, \\ \mathrm{X}\,\varphi &:= \langle\Sigma\rangle\varphi & &\textit{neXt } \varphi, \\ \mathcal{F}\,\varphi &:= \top\,\mathcal{U}\,\varphi & &\textit{future } (\text{or } \textit{eventually})\ \varphi, \\ \mathcal{G}\,\varphi &:= \neg\,\mathcal{F}\,\neg\varphi & &\textit{globally } \varphi. \end{aligned}$$

Remark 1. For comparison let us mention that the syntax and semantics of the logic LTrL defined in [21] is very similar; the difference is only that instead of the modalities $\langle B\ \rangle\varphi$ there is for each letter $a\in\Sigma$ a constant $\langle a^{-1}\rangle\top$. Since the constant $\langle a^{-1}\rangle\top$ refer to the past, we need to use configurations to define its semantics: $(z,x)\models\langle a^{-1}\rangle\top$ if $a\in\max(x)$. It is not clear whether there is a direct translation of $\mathrm{LTL}_{\mathrm{f}}(\Sigma)$ to LTrL or vice versa.

Remark 2. In the case of words, $I=\emptyset$, the meaning of $\langle B\ \rangle\varphi$ is equivalent to the formula $(\langle B\rangle\top)\,\mathcal{U}\,\varphi$. Thus, over words $\langle B\ \rangle\varphi$ is nothing but a simple macro. For traces this is not the case. We always have $\mathrm{L}_{\mathbb{M}}(\langle B\ \rangle\varphi)\subseteq\mathrm{L}_{\mathbb{M}}((\langle B\rangle\top)\,\mathcal{U}\,\varphi)$, but the reverse inclusion fails in general: Let $B=\{b\}$ and $a\in\Sigma$ such that $(a,b)\in I$. Then

$$\mathrm{L}_{\mathbb{M}}(\langle b\ \rangle\langle b\rangle\neg\,\mathrm{X}\,\top) = b^+ \neq \{a,b\}\ \ b\subseteq\mathbb{M}_{I(b)}b^+ = \mathrm{L}_{\mathbb{M}}((\langle b\rangle\top)\,\mathcal{U}\,(\langle b\rangle\neg\,\mathrm{X}\,\top)).$$

The operator $\langle B\ \rangle$ works like a *filter*, given a formula $\langle B\ \rangle\varphi$ the actions from B may pass (but nothing else) and what remains has to satisfy φ. It is open whether $\langle B\ \rangle\varphi$ can be expressed in terms of the other operators.
There is an operator $\langle B\ \rangle$ for all subsets of Σ, but not all of them are needed. For example, if $\mathbb{M}$ is a direct product of free monoids, then no $\langle B\ \rangle$ is needed at all, see Step 2 in the proof of Thm. 2.
Finally let us mention that the language $\mathrm{L}_{\mathbb{M}}(\langle B\ \rangle\varphi)$ has also a fixed-point definition since it is the unique solution to the equation $Z=\mathrm{L}_{\mathbb{M}}(\varphi)\cup BZ$.

Remark 3. Later we shall perform an induction on the size of Σ leading to formulae $\varphi \in \mathrm{LTL_f}(A)$ for $A \subseteq \Sigma$. We note that the interpretation over $\mathbb{M}(\Sigma, I)$ yields $\mathrm{L}_{\mathbb{M}_A}(\varphi) = \mathrm{L}_{\mathbb{M}}(\varphi) \cap \mathbb{M}_A$. Since $\mathbb{M}_A = \mathrm{L}_{\mathbb{M}}(\neg \mathcal{F}\langle \Sigma \setminus A \rangle \top) = \mathrm{L}_{\mathbb{M}}(\langle A \; \rangle \neg \mathrm{X}\, \top)$, a language $\mathrm{L}_{\mathbb{M}_A}(\varphi)$ is expressible in $\mathrm{LTL_f}(\Sigma)$. Another trivial observation is that if $\psi \in \mathrm{LTL_f}(\Sigma)$, then we can construct a formula $\psi_A \in \mathrm{LTL_f}(A)$ such that $\mathrm{L}_{\mathbb{M}}(\psi) \cap \mathbb{M}_A = \mathrm{L}_{\mathbb{M}_A}(\psi_A)$.

The first order theory of traces FO($<$) is given by the syntax:

$$\varphi ::= P_a(x) \mid x < y \mid \neg\varphi \mid \varphi \vee \varphi \mid (\exists x)\varphi,$$

where $a \in \Sigma$ and $x, y \in \mathrm{Var}$ are first order variables. Given a trace $t = [V, E, \lambda]$ and a valuation of the free variables into the vertices $\sigma : \mathrm{Var} \to V$, the semantics is obtained by interpreting the relation $<$ as the transitive closure of E and the predicate $P_a(x)$ by $\lambda(\sigma(x)) = a$. Then we can say when $(t, \sigma) \models \varphi$. If φ is a closed formula (a sentence), then the valuation σ has an empty domain and we define the language $\mathrm{L}_{\mathbb{M}}(\varphi) = \{t \in \mathbb{M} \mid t \models \varphi\}$. We say that a trace language $L \subseteq \mathbb{M}$ is expressible in FO($<$) if there exists some sentence $\varphi \in \mathrm{FO}(<)$ such that $L = \mathrm{L}_{\mathbb{M}}(\varphi)$.
Passing from a temporal logic formula to an FO($<$) one is not very difficult. It is well-known or belongs to folklore. The transformation relies on the fact that a prefix (configuration) p of a trace t can be defined by its maximal vertices. Such a set of maximal vertices is bounded by the maximal number of pairwise independent letters in Σ. Therefore, a prefix inside a trace can be defined using a bounded number of first order variables.
Our new modality $\langle B \; \rangle$ yields no extra difficulty: For instance, if some $\mathrm{LTL_f}(\Sigma)$–formula φ is equivalent to the FO–formula $\tilde{\varphi}$, then the $\mathrm{LTL_f}(\Sigma)$–formula $\langle B^+ \rangle \varphi$ can be expressed by the FO–formula

$$\exists x_1 \cdots \exists x_k \left(\tilde{\varphi}(x_1, \ldots, x_k) \wedge \forall x \Big(\bigvee_{1 \le i \le k} x \le x_i \Big) \longrightarrow \Big(\bigvee_{b \;\; B} P_b(x) \Big) \right)$$

where $k = |B|$ and $\tilde{\varphi}(x_1, \ldots, x_k)$ is the classical relativization of the formula $\tilde{\varphi}$ to the vertices which are not in the past of $x_1, \ldots, x_k$. Hence, we can state:

Proposition 1. *If a trace language is expressible in* $\mathrm{LTL_f}(\Sigma)$*, then it is expressible in* FO($<$).

As in the case of LTrL, this translation yields a non-elementary decision procedure for the uniform satisfiability problem of $\mathrm{LTL_f}$. (See also [7] for a modular decision procedure based on automata constructions.) For the lower bound, we can use [22], since the lower bound is given there for the fragment of LTrL without the previous constant $\langle a^{-1} \rangle \top$. Putting this together the result of Walukiewicz becomes:

Proposition 2 ([22]). *The satisfiability problem for both logics* LTrL *and* $\mathrm{LTL_f}$ *is non-elementary over Mazurkiewicz traces.*

In the remainder we shall use the well-known equivalence between FO(<)–definability and aperiodic languages. Recall that a finite monoid S is *aperiodic*, if there is some $n \geq 0$ such that $s^n = s^{n+1}$ for all $s \in S$. A trace language $L \subseteq \mathbb{M}$ is *aperiodic*, if there exists a morphism to some finite aperiodic monoid $h : \mathbb{M} \to S$ such that $L = h^{-1}(h(L))$.

Proposition 3 ([4,5]). *A trace language is expressible in* FO(<) *if and only if it is aperiodic.*

3 Kamp's Theorem for Finite Traces

Theorem 1. *A trace language is expressible in* FO(<) *if and only if it is expressible in* $\mathrm{LTL}_{\mathrm{f}}(\Sigma)$.

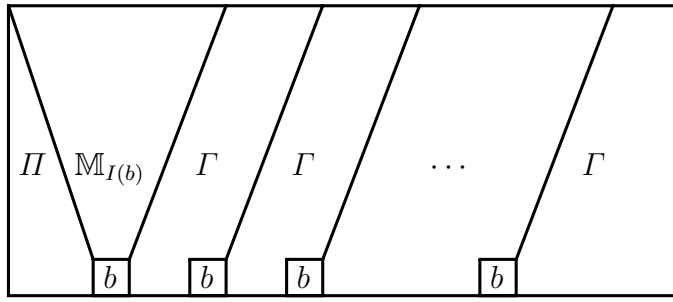
By Props. 1 and 3 it is enough to show that all aperiodic languages in $\mathbb{M}(\Sigma, I)$ are expressible in $\mathrm{LTL}_{\mathrm{f}}(\Sigma)$. This will cover the rest of this section.
In the following $h : \mathbb{M} \to S$ denotes a homomorphism to some finite aperiodic monoid S. A divisor of S is a homomorphic image of some submonoid of S. We shall use two simple facts. First, if $h : S \to G$ is a homomorphism to some group G, then $h(S)$ is trivial. Second, every divisor T of S is itself aperiodic. For a finite set (of states) Q we denote by $\mathrm{Trans}(Q)$ the monoid of mappings from Q to Q. The multiplication is the composition of mappings and the neutral element is the identity id_Q. Every finite monoid can be realized as a submonoid of some $\mathrm{Trans}(Q)$ where $|Q| \leq |S|$. Hence we assume $h : \mathbb{M} \to S \subseteq \mathrm{Trans}(Q)$ and we proceed by induction on $(|Q|, |\Sigma|)$. It is enough to construct a formula for $h^{-1}(s)$ where $s \in S$ and $h^{-1}(s) \neq \emptyset$. If $h(a) = \mathrm{id}_Q$ for all $a \in \Sigma$, which is in particular the case when $|Q| = 1$, then $s = \mathrm{id}_Q$ and $h^{-1}(\mathrm{id}_Q) = \mathbb{M}(\Sigma, I) = \mathrm{L}_{\mathbb{M}}(\top)$.
Hence we may assume that $h(b) \neq \mathrm{id}_Q$ for at least one $b \in \Sigma$ and we fix such a letter $b \in \Sigma$. The crucial observation here is that $h(b)$ is no permutation of Q. Indeed, if $h(b)$ were a permutation, then $h(b)$ would generate a non–trivial subgroup, which is impossible since S is aperiodic. Hence, $h(b)(Q) = Q'$ for some $Q' \subseteq Q$ with $|Q'| < |Q|$.
Define $A = \Sigma \setminus \{b\}$ and let $g : \mathbb{M}_A \to S$ be the restriction of h to the submonoid $\mathbb{M}_A \subset \mathbb{M}$. By induction on $|\Sigma|$ we may assume that $g^{-1}(u)$ is expressible in $\mathrm{LTL}_{\mathrm{f}}(A)$ (and hence in $\mathrm{LTL}_{\mathrm{f}}(\Sigma)$ by Rem. 3) for all $u \in S$. Since $h^{-1}(s) = g^{-1}(s) \cup (h^{-1}(s) \cap \mathbb{M}b\mathbb{M})$, it is enough to construct a formula for $h^{-1}(s) \cap \mathbb{M}b\mathbb{M}$. With respect to the letter b we define two more subsets of $\mathbb{M}$.

$$\Gamma = \{x \in \mathbb{M}_A \mid \min(x) \subseteq D(b)\},$$
$$\Pi = \{x \in \mathbb{M}_A \mid \max(x) \subseteq D(b)\}.$$

The notation Π is chosen since Πb are exactly the *pyramids* of $\mathbb{M}$ where the unique maximal element is b. It should be noted that $(b\Gamma)^*$ and $(\Pi b)^*$ are in fact free submonoids of $\mathbb{M}$, being infinitely generated if $D(b) \neq \{b\}$. We have the following unambiguous decomposition:

$$\mathbb{M}b\mathbb{M} = \Pi\, \mathbb{M}_{I(b)}\, b\, (\Gamma b)^*\, \Gamma.$$

This decomposition is best visualized by the following picture; it is in some sense the guide for the modular construction of the formula $h^{-1}(s) \cap \mathbb{M}b\mathbb{M}$.



Each $s \in h((\Gamma b)^*)$ maps the subset Q to Q. Hence we may define subsets $T, T' \subseteq \mathrm{Trans}(Q)$ by $T = \{s\restriction_{Q'} \mid s \in h(\Gamma b)\}$ and $T' = \{s\restriction_{Q'} \mid s \in h((\Gamma b)^*)\}$. Since $h((\Gamma b)^*)$ is a submonoid of S, the set T' is a monoid and T is a set of generators, the monoid T' is a divisor of S, hence it is aperiodic. By T' we denote the free monoid generated by the set T (here T is viewed as an alphabet). The inclusion $T \subseteq T'$ induces a canonical homomorphism $e : T' \to T$ which is called the *evaluation*. Since T' is a submonoid of $\mathrm{Trans}(Q)$ and $|Q'| < |Q|$, we may use induction (although we might have $|T| > |\Sigma|$). Hence $e^{-1}(t) \subseteq T'$ is expressible in $\mathrm{LTL_f}(T)$ for all $t \in T$. (As a matter of fact, $e^{-1}(t)$ is a language of words in the free monoid T'.)

We need some further notations. The mapping $\sigma : \Gamma b \to T$ defined by $\sigma(x) = h(x)\restriction_{Q'}$ induces a homomorphism $\sigma : (\Gamma b)^* \to T'$ between free monoids. Therefore, we also have the morphism $e \circ \sigma : (\Gamma b)^* \to T$. Note that for all $x \in (\Gamma b)^*$ we have $e \circ \sigma(x) = h(x)\restriction_{Q'}$.

Now, for all $u, v, w \in S$ and $t \in T' \subseteq \mathrm{Trans}(Q)$ the product $uvh(b)tw$ is a well–defined mapping from Q to Q, since $h(b)(Q) = Q$. Hence $uvh(b)tw$ is an element of S. Therefore, using the unambiguous factorization $\mathbb{M}\,b\,\mathbb{M} = \Pi\mathbb{M}_{I(b)}b(\Gamma b)^*\,\Gamma$, the language $h^{-1}(s) \cap \mathbb{M}\,b\,\mathbb{M}$ can be written as the following finite union:

$$\bigcup_{uvh(b)tw=s} \Big(g^{-1}(u) \cap \Pi\Big)\Big(g^{-1}(v) \cap \mathbb{M}_{I(b)}\Big)\, b\, \Big(\sigma^{-1}((e^{-1}(t))\Big)\, \Big(g^{-1}(w) \cap \Gamma\Big).$$

It remains to show that if $L_1, L_2, L_3 \subseteq \mathbb{M}_A$ are trace languages expressible in $\mathrm{LTL_f}(A)$ and if $K = \sigma^{-1}(K')$ for some $K' \subseteq T'$ being expressible in $\mathrm{LTL_f}(T)$, then the language

$$(L_1 \cap \Pi)\,(L_2 \cap \mathbb{M}_{I(b)})\,b\,K\,(L_3 \cap \Gamma)$$

is expressible in $\mathrm{LTL_f}(\Sigma)$.

This is done in the following technical lemmas.

Lemma 2. *Let $\varphi \in \mathrm{LTL_f}(A)$. Then the language $(\mathrm{L}_{\mathbb{M}_A}(\varphi) \cap \Pi)b\mathbb{M}$ is expressible in $\mathrm{LTL_f}(\Sigma)$.*

Proof. The assertion is trivial for $\varphi = \bot$. By structural induction we have to consider formulas of type $\neg\varphi, \varphi \vee \psi, \langle a\rangle\varphi, \langle B\ \rangle\varphi$, and $\varphi\,\mathcal{U}\,\psi$, where $a \in A$ and $B \subseteq A$. We deal with $\langle B\ \rangle\varphi$ and $\varphi\mathcal{U}\,\psi$ only, since the constructions for the other formulas are simpler.

• $\langle B\ \rangle\varphi$: We have

$$(\mathrm{L}_{\mathbb{M}_A}(\langle B\ \rangle\varphi)\cap\Pi)b\mathbb{M} = \bigcup_{B'\subseteq B, C\subseteq A} [B\ , D(C\cup\{b\})]\Big((\mathrm{L}_{\mathbb{M}_A}(\varphi)\cap\Pi)b\mathbb{M}\cap[C, D(b)]b\mathbb{M}\Big).$$

• $\varphi\,\mathcal{U}\,\psi$: We have

$$(\mathrm{L}_{\mathbb{M}_A}(\varphi\,\mathcal{U}\,\psi)\cap\Pi)b\mathbb{M} = ((\mathrm{L}_{\mathbb{M}_A}(\varphi)\cap\Pi)b\mathbb{M}\setminus b\mathbb{M})\,\mathcal{U}\,(\mathrm{L}_{\mathbb{M}_A}(\psi)\cap\Pi)b\mathbb{M}.$$

To see this, consider first $uvbw$ such that $uv \in \Pi, v \in \mathrm{L}_{\mathbb{M}_A}(\psi)$, and where for all $u = u\,u\ , u\ \neq 1$ we have $u\ v \in \mathrm{L}_{\mathbb{M}_A}(\varphi)$. Then $u\ v \in \Pi\setminus\{1\}$ and hence $u\ vbw \notin b\mathbb{M}$. Therefore $uvbw$ is an element of the right–hand side.

Now, let z be an element of the right–hand side. Consider a factorization $z = uy$ with $|u|$ minimal such that $y \in (\mathrm{L}_{\mathbb{M}_A}(\psi)\cap\Pi)b\mathbb{M}$ and for all $u = u\,u\ $ with $u\ \neq 1$, we have $u\ y \in (\mathrm{L}_{\mathbb{M}_A}(\varphi)\cap\Pi)b\mathbb{M}\setminus b\mathbb{M}$. Then, $y = vbw$ with $v \in \mathrm{L}_{\mathbb{M}_A}(\psi)\cap\Pi$ and also $b \notin \mathrm{alph}(u)$.

Consider a factorization $u = u\,u\ $ with $u\ \neq 1$. We have $u\ vbw \in (\mathrm{L}_{\mathbb{M}_A}(\varphi)\cap\Pi)b\mathbb{M}$. We have to show that $u\ v \in \Pi$, because then $u\ v \in \mathrm{L}_{\mathbb{M}_A}(\varphi)$, hence $uv \in \mathrm{L}_{\mathbb{M}_A}(\varphi\mathcal{U}\psi)\cap\Pi$. Let us write $u\ vbw = xbw$ with $x \in \Pi$. Since $b \notin \mathrm{alph}(u\ )$ and $v \in \Pi$ we have $x = x\,v$ and $u\ = x\,x\ $ for some $x\ $ and some $x\ $ independent of v and b. We show that $x\ = 1$. Assume on the contrary that $x\ \neq 1$. Then $|x\ | < |u\ |$ and $z = (u\ x\ )(vbx\ \ w)$ is another possible factorization contradicting the minimality of $|u|$.

Lemma 3. *Assume that $C \times \{b\} \subseteq I$ and let $\varphi \in \mathrm{LTL_f}(C)$. Then the language $\mathrm{L}_{\mathbb{M}_C}(\varphi)(\mathrm{Min}{=}b)$ is expressible in $\mathrm{LTL_f}(\Sigma)$.*

Proof. Again, we proceed by structural induction on φ. Everything is straightforward, up to the until–operator.

• $\varphi\,\mathcal{U}\,\psi$: We have

$$\mathrm{L}_{\mathbb{M}_C}(\varphi\,\mathcal{U}\,\psi)(\mathrm{Min}{=}b)\ =\ \Big(\mathrm{L}_{\mathbb{M}_C}(\varphi)(\mathrm{Min}{=}b)\Big)\,\mathcal{U}\,\Big(\mathrm{L}_{\mathbb{M}_C}(\psi)(\mathrm{Min}{=}b)\Big).$$

The inclusion "$\subseteq$" is trivial. Therefore let z be an element of the right–hand side. Consider a factorization $z = uy$ with $|u|$ minimal such that $y \in \mathrm{L}_{\mathbb{M}_C}(\psi)(\mathrm{Min}{=}b)$ and for all $u = u\,u\ $ with $u\ \neq 1$, we have $u\ y \in \mathrm{L}_{\mathbb{M}_C}(\varphi)(\mathrm{Min}{=}b)$. Then, $y = vw$ with $v \in \mathrm{L}_{\mathbb{M}_C}(\psi)$ and $\mathrm{Min}(w) = \{b\}$.

Assume that $\mathrm{alph}(u) \not\subseteq C$ and let $u = u\,u\ $ with $u\ \in d\mathbb{M}_C$ and $\mathrm{Min}(u\ ) = \{d\}$ for some $d \notin C$. Since $d \in \min(u\ vw) \subseteq C \cup \{b\}$, we must have $b = d$. Hence $u\ = b$ since $C\times\{b\} \subseteq I$. Therefore $z = u\ (vbw)$ is another possible factorization of z contradicting the minimality of $|u|$.

It follows that $\mathrm{alph}(u) \subseteq C$ and for all $u = u\,u\ $ with $u\ \neq 1$, $(u\ v)w$ is the unambiguous factorization in $\mathbb{M}_C(\mathrm{Min}{=}b)$. We deduce that $u\ v \in \mathrm{L}_{\mathbb{M}_C}(\varphi)$ since $u\ vw \in \mathrm{L}_{\mathbb{M}_C}(\varphi)(\mathrm{Min}{=}b)$.

Lemma 4. *Let $L_1, L_2, L_3 \subseteq \mathbb{M}_A$ be trace languages expressible in* $\mathrm{LTL_f}(A)$ *and let $K \subseteq (\Gamma b)^*$ be a trace language such that $K\Gamma$ is expressible in* $\mathrm{LTL_f}(\Sigma)$*. Then the language*

$$(L_1 \cap \Pi)\,(L_2 \cap \mathbb{M}_{I(b)})\,b\,K\,(L_3 \cap \Gamma)$$

is expressible in $\mathrm{LTL_f}(\Sigma)$*.*

Proof. Since the product $\Pi\, b\, \mathbb{M}$ is unambiguous, we can write:

$$\begin{aligned}(L_1 \cap \Pi)(L_2 \cap \mathbb{M}_{I(b)})bK(L_3 \cap \Gamma) &= (L_1 \cap \Pi)b(L_2 \cap \mathbb{M}_{I(b)})K(L_3 \cap \Gamma)\\ &= (L_1 \cap \Pi)b\mathbb{M} \cap \Pi b(L_2 \cap \mathbb{M}_{I(b)})K(L_3 \cap \Gamma).\end{aligned}$$

By Lemma 2 and 1 it is enough to show that $b(L_2 \cap \mathbb{M}_{I(b)})K(L_3 \cap \Gamma)$ is expressible in $\mathrm{LTL_f}(\Sigma)$. Since the product $\mathbb{M}_{I(b)}\,(\mathrm{Min} = b)$ is unambiguous, we have

$$\begin{aligned}b(L_2 \cap \mathbb{M}_{I(b)})K(L_3 \cap \Gamma) &= (L_2 \cap \mathbb{M}_{I(b)})bK(L_3 \cap \Gamma)\\ &= (L_2 \cap \mathbb{M}_{I(b)})(\mathrm{Min} = b) \cap \mathbb{M}_{I(b)}bK(L_3 \cap \Gamma).\end{aligned}$$

By Lemma 3 it is enough to consider $K(L_3 \cap \Gamma)$. Finally, since the product $(\mathrm{Max} \supseteq b)\Gamma$ is unambiguous, we obtain if $1 \in K$ (which is equivalent with $1 \in K\Gamma$)

$$K(L_3 \cap \Gamma) = (L_3 \cap \Gamma) \cup \Big(K\Gamma \cap (\mathrm{Max} \supseteq b)(L_3 \cap \Gamma)\Big).$$

and if $1 \notin K$

$$K(L_3 \cap \Gamma) = K\Gamma \cap (\mathrm{Max} \supseteq b)(L_3 \cap \Gamma).$$

The assertion follows from Lemma 1 since $(\mathrm{Max} \supseteq b)$ is a finite union of max–filters, and Γ is expressible in $\mathrm{LTL_f}(\Sigma)$.

By Lemma 4 it remains to show the next lemma.

Lemma 5. *Let $\varphi \in \mathrm{LTL_f}(T)$. Then the trace language $\sigma^{-1}(L_T(\varphi))\Gamma$ is expressible in* $\mathrm{LTL_f}(\Sigma)$*.*

Proof. For a formula $\varphi \in \mathrm{LTL_f}(T)$ let us denote in this proof by $\sigma^{-1}(\varphi)$ the language $\sigma^{-1}(L_T(\varphi))$. We use structural induction on φ, the basis $\varphi = \bot$ being trivial. Since T^* is a free monoid, it is enough to consider formulas of the form $\neg\varphi$, $\varphi \vee \psi$, $\langle t\rangle\varphi$ where $t \in T$, and $\varphi\,\mathcal{U}\,\psi$.

- $\neg\varphi$: We have $\sigma^{-1}(\neg\varphi)\Gamma = ((\Gamma b)^*\ \Gamma \setminus \sigma^{-1}(\varphi))\Gamma = (\Gamma b)^*\ \Gamma \setminus \sigma^{-1}(\varphi)\Gamma$ since the product $(\Gamma b)^*\ \Gamma$ is unambiguous. Moreover, the language $(\Gamma b)^*\ \Gamma = (\mathrm{Min} \subseteq D(b))$ is expressible in $\mathrm{LTL_f}(\Sigma)$ by Lemma 1.
- $\varphi \vee \psi$: Trivial.
- $\langle t\rangle\varphi$ where $t \in T$: Using the unambiguous decomposition

$$(\Gamma b)(\Gamma b)^*\ \Gamma = (\mathrm{Min} \subseteq D(b)) \cap \Pi\mathbb{M}_{I(b)}b(\Gamma b)^*\ \Gamma$$

we deduce that

$$\sigma^{-1}(\langle t\rangle\varphi)\Gamma = \bigcup_{uvh(b)=t} \Big(\mathrm{Min} \subseteq D(b)\Big) \cap \Big(g^{-1}(u) \cap \Pi\Big)\Big(g^{-1}(v) \cap \mathbb{M}_{I(b)}\Big)b\sigma^{-1}(\varphi)\Gamma.$$

By induction, $\sigma^{-1}(\varphi)\Gamma$ is expressible in $\mathrm{LTL_f}(\Sigma)$. We may apply Lemma 4 to conclude this case.
• $\varphi\,\mathcal{U}\,\psi$: An until–formula $\varphi\,\mathcal{U}\,\psi$ over words is equivalent with $\psi \vee \varphi \wedge \mathrm{X}(\varphi\,\mathcal{U}\,\psi)$. Thus it is enough to consider $\mathrm{X}(\varphi\,\mathcal{U}\,\psi)$. We claim that

$$\sigma^{-1}(\mathrm{X}(\varphi\,\mathcal{U}\,\psi))\Gamma = (\Gamma b)\ \Gamma \cap \left(\left(b\sigma^{-1}(\varphi)\Gamma \cup (\mathrm{Min} \neq b)\right)\,\mathcal{U}\,\left(b\sigma^{-1}(\psi)\Gamma\right)\right).$$

To see the claim, let first $w = u_1 b \cdots u_k b z \in \sigma^{-1}(\mathrm{X}(\varphi\,\mathcal{U}\,\psi))\Gamma$ such that $1 \leq k$, $u_1, \ldots, u_k, z \in \Gamma$, and $t_1 \cdots t_k \in \mathrm{L}_{T^*}(\mathrm{X}(\varphi\,\mathcal{U}\,\psi))$ where $t_i = \sigma(u_i b)$ for $1 \leq i \leq k$. For some $1 \leq j \leq k$ we have $t_{j+1} \cdots t_k \in \mathrm{L}_{T^*}(\psi)$ and for all $2 \leq i \leq j$ we have $t_i \cdots t_k \in \mathrm{L}_{T^*}(\varphi)$. We can write $b u_{j+1} b \cdots u_k b z \in b\sigma^{-1}(\psi)\Gamma$. Consider a factorization $u_1 b \cdots u_{j-1} b u_j = xy$ such that $y \neq 1$. We have to show that $\min(y) = b$ implies $y \in b\sigma^{-1}(\varphi)\Gamma$. However this is clear, because $\min(y) = b$ implies $y = b u_i \cdots b u_j$ for some $2 \leq i \leq j$.
Now, let w be an element of the right–hand side. Since the factorization $(\Gamma b)\ \Gamma$ is unambiguous, there is a unique decomposition $w = u_1 b \cdots u_k b z$ with $u_i,\ z \in \Gamma$. For $1 \leq i \leq k$, we let $t_i = \sigma(u_i b)$. Let $1 \leq j \leq k$ be minimal such that $b u_{j+1} b \cdots u_k b z \in b\sigma^{-1}(\psi)\Gamma$. We must have $b u_i b \cdots u_k b z \in b\sigma^{-1}(\varphi)\Gamma$ for all $2 \leq i \leq j$. It follows that $t_1 \cdots t_k \in \mathrm{L}_{T^*}(\mathrm{X}(\varphi\,\mathcal{U}\,\psi))$.

This finishes the proof of Thm. 1.

4 Kamp's Theorem for Infinite Traces

An *infinite trace* is an infinite dependence graph $[V, E, \lambda]$ such that for all $j \in V$ the set $\downarrow j = \{i \in V \mid i \leq j\}$ is finite. A *real trace* is a finite or infinite trace. The set of real traces is denoted by $\mathbb{R}(\Sigma, I)$ or simply by $\mathbb{R}$. For a real trace $x = [V, E, \lambda]$ the *alphabet at infinity* is defined by the set of letters occurring infinitely many times in x, i.e., $\mathrm{alphinf}(x) = \{a \in \Sigma \mid |\lambda^{-1}(a)| = \infty\}$. We refer to [8] for details about infinite traces.
The aim of this section is to generalize Kamp's theorem for infinite words (i.e., for ω-words) to real traces. The expressiveness result for ω-words is shown in [6]. It is worth mentioning that we do not use this fact on ω-words, which becomes again a formal corollary.
We shall use the syntax of $\mathrm{LTL_f}(\Sigma)$. The semantics of $\mathrm{LTL_f}(\Sigma)$ is defined exactly as in the finitary case. For each $\varphi \in \mathrm{LTL_f}(\Sigma)$ there is a language $\mathrm{L}_{\mathbb{R}}(\varphi)$ and we have $\mathrm{L}_{\mathbb{M}}(\varphi) = \mathrm{L}_{\mathbb{R}}(\varphi) \cap \mathbb{M}$. Moreover, we can express $\mathbb{M}$ as the language $\mathrm{L}_{\mathbb{R}}(\mathcal{F} \neg \mathrm{X}\, \top)$.
Recall also that $x \in L\,\mathcal{U}\,K$, $L, K \subseteq \mathbb{R}$ means that there are y, z with $y \in \mathbb{M}$ such that $z \in K$ and for all $y = y\ y\ ,\ y\ \neq 1$ we have $y\ z \in L$. Clearly, as in the finitary case $\mathrm{L}_{\mathbb{R}}(\varphi\,\mathcal{U}\,\psi) = \mathrm{L}_{\mathbb{R}}(\varphi)\,\mathcal{U}\,\mathrm{L}_{\mathbb{R}}(\psi)$.

Theorem 2. *A language over real traces $L \subseteq \mathbb{R}(\Sigma, I)$ is expressible in* $\mathrm{FO}(<)$ *if and only if it is expressible in* $\mathrm{LTL_f}(\Sigma)$.

As for finite traces, we can pass from an $\mathrm{LTL_f}(\Sigma)$ formula to a $\mathrm{FO}(<)$ formula. It is also well-known that a language of real traces is $\mathrm{FO}(<)$ definable if and only if it is aperiodic [5]. Let us recall the notions of recognizable and of aperiodic real trace language.

Let $h = \mathbb{M} \to S$ be a morphism into some finite monoid S. For $x, y \in \mathbb{R}$, we say that x and y are h-similar, denoted by $x \sim_h y$ if we can find infinite factorizations $x = x_1 x_2 \cdots$ and $y = y_1 y_2 \cdots$ into finite traces such that $h(x_i) = h(y_i)$ for all $i \geq 0$. A real trace language $L \subseteq \mathbb{R}$ is *recognized* by h if it is saturated by $\sim_h$. A real trace language $L \subseteq \mathbb{R}$ is *aperiodic* if it is recognized by some morphism into some finite aperiodic monoid.

We denote by $\approx_h$ the transitive closure of $\sim_h$ which is therefore an equivalence relation. It is well-known (by a Ramsey-type argument) that there are only finitely many equivalence classes $[x]_{\approx_h} = \{y \in \mathbb{R} \mid y \approx_h x\}$ and they form a finite partition of $\mathbb{R}$.

Remark 4. Working with real traces it is convenient to consider languages L which contain simultaneously finite and infinite traces. We allow to write a finite trace $x \in \mathbb{M}$ as an infinite product $x = x_1 x_2 \cdots$ where $x_i = 1$ for almost all i. Thus, it may happen that $x \sim_h y$ where x is finite and y is infinite.

This convention is a matter of taste, but it is quite natural in the presence of concurrency where we have independent components. Some of them may stop and other may run forever.

The remaining of this section consists in the proof that for each aperiodic trace language $L \subseteq \mathbb{R}$, there exists a formula φ in $\mathrm{LTL_f}(\Sigma)$ such that $L = \mathrm{L}_{\mathbb{R}}(\varphi)$. Clearly, this will show Thm. 2. Let $h = \mathbb{M} \to S$ be a morphism to some finite aperiodic monoid S. We can realize S as a submonoid of some transformation monoid $\mathrm{Trans}(Q)$. We show by induction on $(|Q|, |\Sigma|)$ that every language $L \subseteq \mathbb{R}$ recognized by h is expressible in $\mathrm{LTL_f}(\Sigma)$. We use that for each $s \in S$ the language $h^{-1}(s)$ is expressible in $\mathrm{LTL_f}(\Sigma)$ (Prop. 3, Thm. 1).

Step 1: Assume that $h(a) = \mathrm{id}_Q$ for all $a \in \Sigma$ (this is in particular the case when $|Q| = 1$). Then h recognizes only two languages: $\mathbb{R}$ and $\emptyset$ which are both expressible in $\mathrm{LTL_f}(\Sigma)$ by $\top$ and $\bot$ respectively.

Step 2: Assume that $\Sigma = \Sigma_1 \cup \Sigma_2$ where Σ_1, Σ_2 are nonempty subsets of Σ such that $\Sigma_1 \times \Sigma_2 \subseteq I$. For $i = 1, 2$ consider $\mathbb{M}_i = \mathbb{M}(\Sigma_i, I \cap \Sigma_i \times \Sigma_i)$, $\mathbb{R}_i = \mathbb{R}(\Sigma_i, I \cap \Sigma_i \times \Sigma_i)$, and $h_i : \mathbb{M}_i \to S$ the restriction of h to $\mathbb{M}_i$. We claim that L is a finite union of products of the form $L_1 L_2$ where L_1, L_2 are recognized by h_1, h_2 respectively.

Let $x \in L$, we can write $x = x_1 x_2$ with $x_1 \in \mathbb{R}_1$ and $x_2 \in \mathbb{R}_2$. For $i = 1, 2$, let $L_i = [x_i]_{\approx_{h_i}}$. Clearly, L_i is recognized by h_i and $x \in L_1 L_2$. Now, for $i = 1, 2$, let $y_i, z_i \in \mathbb{R}_i$ be such that $y_i \sim_{h_i} z_i$. Since $\Sigma_1 \times \Sigma_2 \subseteq I$, it is easy to see that $y_1 y_2 \sim_h z_1 z_2$. We deduce that $L_1 L_2 \subseteq L$. The claim follows since there are only finitely many equivalence classes under $\approx_{h_1}$ and $\approx_{h_2}$.

Now, we can conclude the second case. Using the induction on $|\Sigma|$ we know that L_1 and L_2 are expressible in $\mathrm{LTL_f}(\Sigma_1)$ and $\mathrm{LTL_f}(\Sigma_2)$ respectively. We deduce that L is expressible in $\mathrm{LTL_f}(\Sigma)$ using Lemma 6.

Lemma 6. *For $i = 1, 2$, let $L_i \subseteq \mathbb{R}_i$ be a trace language expressible in* $\mathrm{LTL_f}(\Sigma_i)$. *Then the language $L_1 L_2$ is expressible in* $\mathrm{LTL_f}(\Sigma)$.

Proof. The product $\mathbb{R}_1\mathbb{R}_2$ is unambiguous and we have $L_1L_2 = (L_1\mathbb{R}_2) \cap (\mathbb{R}_1 L_2)$. The result follow since we can show by structural induction that for each formula φ in $\mathrm{LTL_f}(\Sigma_1)$ we have $\mathrm{L}_{\mathbb{R}}(\varphi) = \mathrm{L}_{\mathbb{R}_1}(\varphi)\mathbb{R}_2$.

Step 3: For a subset $A \subseteq \Sigma$ we denote by $\overline{A}$ its complement, $\overline{A} = \Sigma \setminus A$, and for $\# \in \{\subseteq, =\}$, we define $(\mathrm{Alphinf}\ \#\ A) = \{x \in \mathbb{R} \mid \mathrm{alphinf}(x)\ \#\ A\}$. We show that if A is a proper subset of Σ then $L \cap (\mathrm{Alphinf} \subseteq A)$ is expressible in $\mathrm{LTL_f}(\Sigma)$. For $s \in S$ we let $L(s) = \{y \in \mathbb{R}_A \mid h^{-1}(s) \cdot y \cap L \neq \emptyset\}$. We first show that

$$L \cap (\mathrm{Alphinf} \subseteq A) = \bigcup_{s\ S} (h^{-1}(s) \cap (\mathrm{Max} \subseteq \overline{A}))L(s).$$

Let $z \in L \cap (\mathrm{Alphinf} \subseteq A)$. Using the unambiguous decomposition $(\mathrm{Alphinf} \subseteq A) = (\mathrm{Max} \subseteq \overline{A})\mathbb{R}_A$ we can write $z = xy$ with $x \in (\mathrm{Max} \subseteq \overline{A})$ and $y \in \mathbb{R}_A$. Let $s = h(x)$, we obtain $z = xy \in (h^{-1}(s) \cap (\mathrm{Max} \subseteq \overline{A}))L(s)$.
Conversely, let $s \in S$, $x \in h^{-1}(s) \cap (\mathrm{Max} \subseteq \overline{A})$ and $y \in L(s)$. Let $x\ \in h^{-1}(s)$ be such that $x\ y \in L$. Since $xy \sim_h x\ y$ and L is recognized by h we deduce that $xy \in L$. Moreover, $\mathrm{alphinf}(xy) \subseteq \mathrm{alph}(y) \subseteq A$.
Then, we show that $L(s)$ is recognized by the restriction h_A of h to $\mathbb{M}_A$. Indeed, let $y, z \in \mathbb{R}_A$ be such that $y \sim_{h_A} z$. Note that this implies $y \sim_h z$. Assume that $y \in L(s)$, then there exists $x \in h^{-1}(s)$ such that $xy \in L$. Since $xy \sim_h xz$, we deduce that $xz \in L$ and then $z \in L(s)$.
From the finitary case we know that each language $h^{-1}(s)$ is expressible in $\mathrm{LTL_f}(\Sigma)$. So is the language $(\mathrm{Max} \subseteq \overline{A})$ as a finite union of max–filters. Using the induction on $|\Sigma|$, we deduce that the language $L(s)$ is expressible in $\mathrm{LTL_f}(A)$. From Lemma 7 we conclude that $L \cap (\mathrm{Alphinf} \subseteq A)$ is expressible in $\mathrm{LTL_f}(\Sigma)$.

Lemma 7. *Let $A \subseteq \Sigma$ and let $K \subseteq \mathbb{M}$ and $L \subseteq \mathbb{R}_A$ be trace languages expressible in* $\mathrm{LTL_f}(\Sigma)$ *and* $\mathrm{LTL_f}(A)$ *respectively. Then the language $(K \cap (\mathrm{Max} \subseteq \overline{A}))L$ is expressible in* $\mathrm{LTL_f}(\Sigma)$ *as well.*

Proof. First the product $(\mathrm{Max} \subseteq \overline{A})\mathbb{R}_A$ is unambiguous and we have

$$(K \cap (\mathrm{Max} \subseteq \overline{A}))L = (K \cap (\mathrm{Max} \subseteq \overline{A}))\mathbb{R}_A \cap (\mathrm{Max} \subseteq \overline{A})L.$$

The language $(\mathrm{Max} \subseteq \overline{A})$ is a union of max–filters. Hence, by Lemma 1, the language $(\mathrm{Max} \subseteq \overline{A})L$ is expressible in $\mathrm{LTL_f}(\Sigma)$.
We show now by induction on φ that the language $(\mathrm{L}_{\mathbb{M}}(\varphi) \cap (\mathrm{Max} \subseteq \overline{A}))\mathbb{R}_A$ is expressible in $\mathrm{LTL_f}(\Sigma)$. The proof is similar to that of Lemma 2.
- The cases $\bot$ and $\varphi \vee \psi$ are trivial.
- $\neg\varphi$: Since the product $(\mathrm{Max} \subseteq \overline{A})\mathbb{R}_A$ is unambiguous, we have

$$(\mathrm{L}_{\mathbb{M}}(\neg\varphi) \cap (\mathrm{Max} \subseteq \overline{A}))\mathbb{R}_A = (\mathrm{Max} \subseteq \overline{A})\mathbb{R}_A \setminus (\mathrm{L}_{\mathbb{M}}(\varphi) \cap (\mathrm{Max} \subseteq \overline{A}))\mathbb{R}_A.$$

Moreover $(\mathrm{Max} \subseteq \overline{A})\mathbb{R}_A$ is the set of traces such that the alphabet at infinity is contained in A and is clearly expressible in $\mathrm{LTL_f}(\Sigma)$.

• $\langle a\rangle\varphi$: Follows from Lemma 1 by writing $(\mathrm{L}_{\mathbb{M}}(\langle a\rangle\varphi)\cap(\mathrm{Max}\subseteq\overline{A}))\mathbb{R}_A$ as

$$\bigcup_{\{C\subseteq\Sigma\mid a\ \ \overline{A}\cup D(C)\}} a\Big((\mathrm{L}_{\mathbb{M}}(\varphi)\cap(\mathrm{Max}\subseteq\overline{A}))\mathbb{R}_A\cap[C,\overline{A}]\mathbb{R}_A\Big).$$

• $\langle B\ \rangle\varphi$: Similarly, we can write $(\mathrm{L}_{\mathbb{M}}(\langle B\ \rangle\varphi)\cap(\mathrm{Max}\subseteq\overline{A}))\mathbb{R}_A$ as

$$\bigcup_{B'\subseteq B, C\subseteq\Sigma}[B\ ,\overline{A}\cup D(C)]\Big((\mathrm{L}_{\mathbb{M}}(\varphi)\cap(\mathrm{Max}\subseteq\overline{A}))\mathbb{R}_A\cap[C,\overline{A}]\mathbb{R}_A\Big).$$

• $\varphi\,\mathcal{U}\,\psi$: We have

$$(\mathrm{L}_{\mathbb{M}}(\varphi\mathcal{U}\psi)\cap(\mathrm{Max}\subseteq\overline{A}))\mathbb{R}_A=(\mathrm{L}_{\mathbb{M}}(\varphi)\cap(\mathrm{Max}\subseteq\overline{A}))\mathbb{R}_A\mathcal{U}(\mathrm{L}_{\mathbb{M}}(\psi)\cap(\mathrm{Max}\subseteq\overline{A}))\mathbb{R}_A.$$

Step 4: We may now assume that the alphabet Σ is connected and we show that for some language L expressible in $\mathrm{LTL}_{\mathrm{f}}(\Sigma)$ we have

$$L\cap(\mathrm{Alphinf}=\Sigma)\subseteq L\ \subseteq L.$$

We proceed as for finite traces by choosing a letter $b\in\Sigma$ such that $h(b)\neq\mathrm{id}_Q$. We know that $Q\ =h(b)(Q)$ is a proper subset of Q. We let $A=\Sigma\setminus\{b\}$, $\Pi=\{x\in\mathbb{M}_A\mid\max(x)\subseteq D(b)\}$ and $\Gamma=\{x\in\mathbb{M}_A\mid\min(x)\subseteq D(b)\}$. For $s\in S$ we define $L(s)=\{y\in\mathbb{R}\mid h^{-1}(s)\cdot y\cap L\neq\emptyset\}$. Since L is recognized by h, we deduce easily that $L(s)$ is recognized by h and $h^{-1}(s)L(s)\subseteq L$ for all $s\in S$.
Now we claim that $L\cap(\mathrm{Alphinf}=\Sigma)\subseteq L\ \subseteq L$ with

$$L\ =\bigcup_{u,v\ \ S}(h^{-1}(u)\cap\Pi)(h^{-1}(v)\cap\mathbb{M}_{I(b)})(L(uv)\cap(b\Gamma)^\infty).$$

Since Σ is connected and by the factorization $(\mathrm{Alphinf}=\Sigma)\subseteq\Pi\mathbb{M}_{I(b)}(b\Gamma)^\infty$ which is unambiguous, the first inclusion $L\cap(\mathrm{Alphinf}=\Sigma)\subseteq L\ $ follows easily. The second inclusion is clear since $h^{-1}(s)L(s)\subseteq L$ for all $s\in S$.

We use the definitions and notations introduced earlier for the sets $T,T\ \subseteq\mathrm{Trans}(Q\ )$, the morphisms $e:T\ \to T\ $ and $\sigma:(\Gamma b)\ \to T\ $ and we extend $\sigma:(\Gamma b)^\infty\to T^\infty$ naturally.
For $K\subseteq\mathbb{R}$ recognized by h we define

$$K\ =\{\sigma(x)\mid bx\in K\cap b(\Gamma b)^\infty\}\subseteq T^\infty.$$

We show that
1) $K\cap b(\Gamma b)^\infty=b\sigma^{-1}(K\ )$
2) $K\ $ is recognized by the morphism $e:T\ \to T$.
For the first point, one inclusion is clear. Conversely, let $y=y_1by_2b\cdots\in\sigma^{-1}(K\ )$ with $y_i\in\Gamma$ and let $bx\in K\cap b(\Gamma b)^\infty$ be such that $\sigma(x)=\sigma(y)$. We write $x=x_1bx_2b\cdots$ with $x_i\in\Gamma$. For all i we have $h(x_ib)\restriction_{Q'}=\sigma(x_ib)=\sigma(y_ib)=h(y_ib)\restriction_{Q'}$ and we deduce that $h(bx_ib)=h(by_ib)$. It follows that $bx\sim_h bz\sim_h by$ with $z=x_1by_2bx_3by_4b\cdots$. Since $bx\in K$ and $bx\approx_h by$, we deduce that $by\in K$ and we have proved the converse inclusion since $by\in b(\Gamma b)^\infty$.

For the second point, let $u, v \in T^\infty$ be such that $u \sim_e v$. We write $u = u_1 u_2 \cdots$ and $v = v_1 v_2 \cdots$ with $e(u_i) = e(v_i)$ for all i. Assume that $u \in K$ and let $bx = bx_1 bx_2 b \cdots \in K$ with $x_i b \in (\Gamma b)$ and $\sigma(x_i b) = u_i$. Let $y = y_1 by_2 b \cdots$ with $y_i b \in (\Gamma b)$ and $\sigma(y_i b) = v_i$. Then, for all i we have $h(x_i b)\restriction_{Q'} = e(u_i) = e(v_i) = h(y_i b)\restriction_{Q'}$ and therefore $h(bx_i b) = h(by_i b)$. Hence, we have $bx \sim_h bz \sim_h by$ with $z = x_1 by_2 bx_3 by_4 b \cdots$ and therefore $bx \approx_h by$. Since $bx \in K$ which is recognized by h, it follows that $by \in K$ and $v = \sigma(y) \in K$.

We have proved that K is recognized by the morphism $e : T \to T \subseteq \mathrm{Trans}(Q)$. We deduce by induction on $|Q|$ that K is expressible in $\mathrm{LTL_f}(T)$. Using Lemma 9 (below) we deduce that $K \cap b(\Gamma b)^\infty = b\sigma^{-1}(K)$ is expressible in $\mathrm{LTL_f}(\Sigma)$. Finally, using Lemma 8, we deduce that L is expressible in $\mathrm{LTL_f}(\Sigma)$.

Lemma 8. *Let $L_1, L_2 \subseteq \mathbb{M}$ and $L_3 \subseteq b(\Gamma b)^\infty$ be expressible in $\mathrm{LTL_f}(\Sigma)$. Then the language $(L_1 \cap \Pi)(L_2 \cap \mathbb{M}_{I(b)})L_3$ is expressible in $\mathrm{LTL_f}(\Sigma)$.*

Proof. Similar to that of Lemma 4. We use the decomposition

$$(L_1 \cap \Pi)(L_2 \cap \mathbb{M}_{I(b)})L_3 = (L_1 \cap \Pi)b\mathbb{R} \cap \Pi(L_2 \cap \mathbb{M}_{I(b)})(\mathrm{Min} = b) \cap \mathbb{M}_A L_3.$$

Lemma 9. *Let $K \subseteq T^\infty$ be a language expressible in $\mathrm{LTL_f}(T)$. Then the language $\sigma^{-1}(K)$ is expressible in $\mathrm{LTL_f}(\Sigma)$.*

Proof. Similar to that of Lemma 5 and therefore omitted for lack of space.

References

1. R. Alur, D. Peled, and W. Penczek. Model-checking of causality properties. In *Proceedings of LICS'95*, pages 90–100, 1995.
2. J. Cohen, D. Perrin, and J. E. Pin. On the expressive power of temporal logic. *Journal of Computer and System Sciences*, 46:271–295, 1993.
3. V. Diekert and G. Rozenberg, editors. *The Book of Traces.* World Scientific, Singapore, 1995.
4. W. Ebinger. *Charakterisierung von Sprachklassen unendlicher Spuren durch Logiken.* Dissertation, Institut für Informatik, Universität Stuttgart, 1994.
5. W. Ebinger and A. Muscholl. Logical definability on infinite traces. *Theoretical Computer Science*, 154:67–84, 1996. A preliminary version appeared in Proceedings of the 20th International Colloquium on Automata, Languages and Programming (ICALP'93), Lund (Sweden) 1993, Lecture Notes in Computer Science 700, 1993.
6. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Conference Record of the 12th ACM Symposium on Principles of Programming Languages*, pages 163–173, Las Vegas, Nev., 1980.
7. P. Gastin, R. Meyer, and A. Petit. A (non-elementary) modular decision procedure for ltrl. In L. Brim, F. Gruska, and J. Zlatuska, editors, *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS'98)*, number 1450 in Lecture Notes in Computer Science, pages 356–365. Springer, 1998.

8. P. Gastin and A. Petit. Infinite traces. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*, chapter 11, pages 393–486. World Scientific, Singapore, 1995.
9. J. A. W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, Calif., 1968.
10. R. McNaughton and S. Papert. *Counter-free Automata*. MIT Press, 1971.
11. R. Meyer and A. Petit. Expressive completeness of LTrL on finite traces: an algebraic proof. In *Proceedings of STACS'98*, number 1373 in Lecture Notes in Computer Science, pages 533–543, 1998.
12. M. Mukund and P. S. Thiagarajan. Linear time temporal logics over Mazurkiewicz traces. In *Proceedings of the 21th MFCS, 1996*, number 1113 in Lecture Notes in Computer Science, pages 62–92. Springer, 1996.
13. P. Niebert. A ν-calculus with local views for sequential agents. In *Proceedings of the 20th MFCS, 1995*, number 969 in Lecture Notes in Computer Science, pages 563–573. Springer, 1995.
14. W. Penczek. Temporal logics for trace systems: On automated verification. *International Journal of Foundations of Computer Science*, 4:31–67, 1993.
15. W. Penczek and R. Kuiper. Traces and logic. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*, chapter 10, pages 307–390. World Scientific, Singapore, 1995.
16. R. Ramanujam. Locally linear time temporal logic. In *Proceedings of LICS'96*, Lecture Notes in Computer Science, pages 118–128, 1996.
17. M. P. Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8:190–194, 1965.
18. A. Sistla and E. Clarke. The complexity of propositional linear time logic. *J. ACM*, 32:733–749, 1985.
19. P. S. Thiagarajan. A trace based extension of linear time temporal logic. In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science (LICS'94)*, pages 438–447, 1994.
20. P. S. Thiagarajan. A trace consistent subset of PTL. In *Proceedings of CONCUR'95*, number 962 in Lecture Notes in Computer Science, pages 438–452, 1995.
21. P. S. Thiagarajan and I. Walukiewicz. An expressively complete linear time temporal logic for Mazurkiewicz traces. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, 1997.
22. I. Walukiewicz. Difficult configurations – on the complexity of LTrL. In Kim G. Larsen et al., editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP'98), Aalborg (Denmark) 1998*, number 1443 in Lecture Notes in Computer Science, pages 140–151, Berlin-Heidelberg-New York, 1998. Springer.
23. Th. Wilke. Classifying discrete temporal properties. In Chr. Meinel and S. Tison, editors, *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS'99), Trier 1999*, number 1443 in Lecture Notes in Computer Science, pages 32–46, Berlin-Heidelberg-New York, 1999. Springer. Invited Lecture.

Using Fields and Explicit Substitutions to Implement Objects and Functions in a de Bruijn Setting

Eduardo Bonelli

[1] Laboratoire de Recherche en Informatique, Bât 490, Université de Paris-Sud, 91405, Orsay Cedex, France, email: bonelli@lri.fr
[2] Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Argentina.

Abstract. We propose a calculus of explicit substitutions with de Bruijn indices for implementing objects and functions which is confluent and preserves strong normalization. We start from Abadi and Cardelli's ς-calculus [1] for the object calculus and from the λ_υ-calculus [20] for the functional calculus. The de Bruijn setting poses problems when encoding the λ_υ-calculus within the ς-calculus following the style proposed in [1]. We introduce fields as a primitive construct in the target calculus in order to deal with these difficulties. The solution obtained greatly simplifies the one proposed in [17] in a named variable setting. We also eliminate the conditional rules present in the latter calculus obtaining in this way a full non-conditional first order system.

1 Introduction

The object oriented paradigm is heavily used in the software engineering process. The simplicity of the underlying ideas makes it especially suited for resolving complex tasks. However, since no widespread consensus on its theoretical foundations has been reached, rigorous reasoning is difficult to achieve. In fact due to its success in software development the rapid evolution of object oriented languages has converted the task of formulating a formal calculus capturing the general principals of the paradigm into an interesting problem. In this direction, the calculi introduced by Abadi and Cardelli [1] constitute a simple yet powerful formalism.

The core untyped calculus presented in [1] is called the ς-calculus. This calculus defines objects as collections of methods and supports *method update*, thus providing mecanisms for inheritance by embedding. It also captures the notion of *self*, a name which allows a method to refer to its host object. These primitive constructs allow the representation of a vast amount of object oriented features, including classes, traits, and multiple inheritance. Furthermore, it may be extended into a typed setting.

Evaluation in the ς-calculus is accomplished by means of reduction rules and *substitution*. As in the lambda calculus, substitution is defined as an *atomic*

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 204–219, 1999.

operation which does not form part of the calculus. Therefore, any implementation has to deal with its computation. This is not a trivial task, in particular in a setting where variables are represented by names (as is usually done). Thus inevitably, a gap arises between theory and implementation. Calculi of explicit substitution eliminate this gap by decomposing the substitution process into more atomical parts and incorporating the behaviour of these parts as new operators in the calculus. This has the added benefit that we obtain a finer grained control on the computation of the substitution, providing for example tools for studying refinement of proofs in type theory or the theory of abstract evaluation machines.

Explicit substitution calculi arise with the study of pioneering calculus λ_σ [2]. The idea is simple: the notion of substitution used to define β-reduction in the lambda calculus takes place at a meta-level, explicit substitution calculi add new operators, and reduction rules for these new operators, so that substitutions may be computed at the object-level. Explicit substitution constructors thus implement substitution within the calculus, drawing the theory closer to the implementation level. Abadi, Cardelli, Curien and Lévy used indices, as introduced by de Bruijn in [8], to represent variables and introduced also a typed version of λ_σ. Other calculi of explicit substitutions are $\lambda_{\sigma_\Uparrow}$ [13], λ_v[20], λ_s [14], λ_d [11], λ_ζ [23], λ_χ [21], and λ_x [26]. All but the last two of these calculi have been formulated with de Bruijn indices, λ_χ uses de Bruijn levels and λ_x uses variable names. They have all been studied in the setting of the lambda calculus. Attempts to study explicit substitutions in a general setting are the Explicit CRS [5], based on the higher order rewriting formalism CRS [18], and the eXplicit Reduction Systems [24]. These formalisms although defined in a higher order rewriting setting deal with a fixed "built in" explicit substitution calculus (Σ in Explicit CRS and $\sigma_\Uparrow$ in XRS). This rises naturally the question of the generality of these formalisms as theories of explicit substitution. In particular, we shall see below that the calculus of explicit substitutions implementing the ς object oriented language fits in neither of these schemes.

In this paper we provide an implementation language for object oriented programming as formalized by the ς-calculus. We introduce the untyped ς_{DBES}-calculus, an explicit substitution calculus in a de Bruijn indice setting which is confluent and preserves strong normalization. Abadi and Cardelli's ς-calculus allows the execution of lambda calculus expressions by means of an elegant translation which requires the use of fields (see section 4). Although it does not provide fields as primitive constructions they can be simulated. A brief analysis, as discussed in section 5, shows that this simulation is not well adapted when variable names are replaced by de Bruijn indices and explicit substitutions are incorporated. The ς_{DBES}-calculus introduces fields as *primitive constructs* in the language, thus allowing to merge both the object oriented language and the functional lambda calculus in the spirit of [1].

The use of de Bruijn indices by encoding variable names with numbers avoids having to deal with α-conversion, thus simplifying the associated reduction relation.

Most importantly, the analysis pertaining to the merging of the object oriented language and the functional lambda calculus while retaining the spirit of the aforementioned translation revealed [17] that two different notions of substitution were necessary: Ordinary Substitution and Invoke Substitution. Ordinary Substitution is used to perform evaluation of methods and may be related to the usual notion of substitution which is made explicit in calculi for the lambda calculus. Whereas Invoke Substitution is used to implement functions as objects and reports a different behaviour. Also, some interaction between both types of substitutions must be specified. Therefore, higher order rewriting formalisms such as CRS [18], Explicit CRS [5] or XRS [24] do not cater for this difference. Consequently, the ς_{DBES}-calculus is not an instance of any of these formalisms.

The work reported here is very much in the spirit of [17]. There a calculus of explicit substitutions in a *named variable* setting for the ς-calculus, called ς_{ES}, is defined. The interaction between Ordinary and Invoke substitution is easier to express since one may specify conditions on free variables naturally. Whereas in a *de Bruijn setting* the situation is more complex since conditions on free variables imply adjustments on indices. The solution we have adopted by incorporating field constructs allows us, in contrast to [17], to do away with the conditions on free variables, thus obtaining a non-conditional interaction rule. Also, the calculus obtained here is a first order calculus. No binding operators are needed. As remarked above, just as the ς_{ES} calculus is not an instance of Explicit CRS, the ς_{DBES}-calculus is not an instance of an XRS.

This paper is organized as follows. Section 2 recalls the main concepts and definitions of the ς-calculus. Section 3 introduces the ς-calculus with de Bruijn indices, called ς_{DB}-calculus. Section 4 is devoted to the ς-calculus with de Bruijn indices and fields, the $\varsigma^{\bullet}_{DB}$-calculus. Here we introduce the syntax, we prove confluence and finally we show how it relates to the ς_{DB}-calculus. Also the invoke substitution is defined. Section 5 defines de $\varsigma^{\bullet}_{DB}$-calculus with explicit substitutions, called ς_{DBES}-calculus. The following section deals with the encoding of lambda calculus with explicit substitutions in the ς_{DBES}-calculus. Section 7 proves the main properties of ς_{DBES}: confluence and preservation of strong normalization. Finally, we conclude and suggest future research directions.

2 The ς-Calculus

This section presents the ς-calculus as defined in [1]. We have at our disposal an infinite list of variables denoted $x, y, z, \ldots$, and an infinite list of labels denoted $l, l_i, l', \ldots$. The labels shall be used to reference methods. An object is represented as a collection of methods denoted $l_i = \varsigma(x_i).a_i$. Each method has a reference or method name l_i and a method body $\varsigma(x_i).a_i$. The labels of an object's methods are assumed to be all distinct. Operations allowed on objects are *method invocation* and *method update*. A method invocation of the method l_j in an object $[l_i = \varsigma(x_i).a_i{}^{i\in 1..n}]$ is represented by the term $[l_i = \varsigma(x_i).a_i{}^{i\in 1..n}].l_j$. As a result of a method invocation, not only the corresponding method body is returned but also, this method body is supplied with a copy of its host object.

Thus method bodies are represented as $\varsigma(x_i).a_i$ where ς is a binder that binds the variable x_i in a_i. This variable called *self* will be replaced by the host object when the associated method is invoked. It is this notion of *self* captured by the ς-calculus that makes it so versatile. The other valid operation on objects is *method update*. A method $l_j = \varsigma(x_j).a_j$ in an object o may be replaced by a new method $l'_j = \varsigma(x'_j).a'_j$, thus resulting in a new object o'.

The terms of the ς-calculus, denoted $\mathcal{T}_\varsigma$, is given by the following grammar $a ::= x \mid a.l \mid a \lhd \langle l = \varsigma(x).a \rangle \mid [l_i = \varsigma(x_i).a_i{}^{i\in 1..n}]$.

A variable convention similar to the one present in λ-calculus is adopted: terms differing only in the names of their bound variables (i.e. α-equivalent) are considered identical.

We say that x is a *variable*, $a.l$ is a *method invocation*, $a \lhd \langle l = \varsigma(x).a \rangle$ is a *method update* and $[l_i = \varsigma(x_i).a_i{}^{i\in 1..n}]$ is an *object*.

In order to introduce reduction between terms the notions of free variables and substitution are defined as in [1]. The result of substituting a free variable x in a term a for a term b shall be denoted $a\{x \leftarrow b\}$.

The semantics of the ς-calculus, referred to as the *primitive semantics* in [1], is defined by the following rewrite rules:

$$\begin{array}{lll} o.l_j & \longrightarrow_\varsigma a_j\{x_j \leftarrow o\} & j \in 1..n \\ o \lhd \langle l_j = \varsigma(x).a \rangle & \longrightarrow_\varsigma [l_j = \varsigma(x).a,\ l_i = \varsigma(x_i).a_i{}^{i\in 1..n, i\neq j}] & j \in 1..n \end{array}$$

where $o \equiv [l_i = \varsigma(x_i).a_i{}^{i\in 1..n}]$.

The first rule defines the semantics of *method invocation*. The result of invoking the method l_j (a "call" to method $\varsigma(x_j).a_j$) is the body of the method a_j where the self variable has been replaced by a copy of the host object. The second rule defines the semantics of *method update*. Note that the substitution operator is not part of the ς-calculus but rather a meta-operation.

As regards the expressive power of this calculus, it is shown in [1] that lambda terms can be encoded as objects and that β-reduction can be simulated by ς-reduction.

Definition 1. *The translation* $\prec\!\prec . \succ\!\succ$ *from* λ*-terms to* $\mathcal{T}_\varsigma$ *is defined as:*

$$\begin{array}{lll} \prec\!\prec x \succ\!\succ & =_{def} & x \\ \prec\!\prec \lambda x.a \succ\!\succ & =_{def} & [arg = \varsigma(z).z.arg, val = \varsigma(x).\ \prec\!\prec a \succ\!\succ \{x \leftarrow x.arg\}] \\ \prec\!\prec ab \succ\!\succ & =_{def} & \prec\!\prec a \succ\!\succ \bullet \prec\!\prec b \succ\!\succ \\ & & where\ c \bullet d\ =_{def} (c \lhd \langle arg = \varsigma(y).d \rangle).val\ with\ y \notin FV(d) \end{array}$$

It is then proved for λ-terms a and b that if $a \longrightarrow_\beta b$ then $\prec\!\prec a \succ\!\succ \overset{*}{\longrightarrow}_\varsigma \prec\!\prec b \succ\!\succ$.

3 The ς-Calculus à la de Bruijn (ς_{DB}-Calculus)

Here we introduce the ς-calculus in a de Bruijn setting. N.G.de Bruijn introduced a notation for lambda terms which deals with the problem of having to rename bound variables when implementing mechanized provers [8].

Instead of labelling bound variables with names (as above) variables are labelled with natural numbers. This number is usually referred to as a de Bruijn index. If a term is viewed as a tree, an index n stands for a variable bound by the n-th binder starting from the position of the index. For example, the term $[l_1 = \varsigma(x_1).[l_2 = \varsigma(y_1).x_1, l_3 = \varsigma(z_1).z_1], l_4 = \varsigma(x_2).y_2]$ is represented as $[l_1 = \varsigma([l_2 = \varsigma(2), l_3 = \varsigma(1)]), l_4 = \varsigma(2)]$. Note that free variables are represented by indices greater than the number of binders above it, thus a variable assigned an index n that has m sigmas above it refers to the $(n-m)$-th free variable (in a preestablished ordering on the set of variables). The advantage attained is that there is no longer any need to perform renaming of bound variables. Nevertheless we must take care of index adjustments: if a substitution drags a term under a binder, its indices must be adjusted in order to avoid unwanted capture of indices.

The terms of the ς-calculus à la de Bruijn (the ς_{DB}-calculus), denoted $\mathcal{T}_{\varsigma_{DB}}$, are characterized by the grammar $a ::= \underline{p} \mid a.l \mid a \lhd \langle l = \varsigma(a) \rangle \mid [l_i = \varsigma(a_i)^{\ i\in 1..n}]$ where p is a natural number ($\mathbb{N}$) greater than zero. We shall use underlined natural numbers for indices.

Definition 2 (Ordinary Substitution). *Let a and b be pure terms and $n \geq 1$. The substitution of a by b at level n is defined as follows:*

$$
\begin{array}{ll}
[l_i = \varsigma(a_i)^{\ i\in 1..m}]\{n \leftarrow b\} & =_{def}\ [l_i = \varsigma(a_i\{n+1 \leftarrow b\})^{\ i\in 1..m}] \\
d.l\{n \leftarrow b\} & =_{def}\ d\{n \leftarrow b\}.l \\
d \lhd \langle l = \varsigma(c) \rangle \{n \leftarrow b\} & =_{def}\ d\{n \leftarrow b\} \lhd \langle l = \varsigma(c\{n+1 \leftarrow b\}) \rangle \\
\underline{p}\{n \leftarrow b\} & =_{def} \begin{cases} \underline{p-1} & if\ p > n \\ U_0^n(b) & if\ p = n \\ \underline{p} & if\ p < n \end{cases}
\end{array}
$$

where for every $i \geq 0$ and $n \geq 1$, $U_i^n(.)$ is an updating function from terms in $\mathcal{T}_{\varsigma_{DB}}$ to terms in $\mathcal{T}_{\varsigma_{DB}}$ defined as follows:

$$
\begin{array}{ll}
U_i^n([l_i = \varsigma(a_i)^{\ i\in 1..m}]) & =_{def}\ [l_i = \varsigma(U_{i+1}^n(a_i))^{\ i\in 1..m}] \\
U_i^n(a.l) & =_{def}\ U_i^n(a).l \\
U_i^n(a \lhd \langle l = \varsigma(c) \rangle) & =_{def}\ U_i^n(a) \lhd \langle l = \varsigma(U_{i+1}^n(c)) \rangle \\
U_i^n(\underline{p}) & =_{def} \begin{cases} \underline{p+n-1} & if\ p > i \\ \underline{p} & if\ p \leq i \end{cases}
\end{array}
$$

We now define the appropriate reduction rules using the notion of substitution defined above.

Definition 3 (Reduction in the ς_{DB}-calculus). *Reduction in the ς_{DB}-calculus is defined by the following rewrite rules:*

$$
\begin{array}{ll}
[l_i = \varsigma(b_i)^{\ i\in 1..n}].l_j & \longrightarrow_{\varsigma_{DB}}\ b_j\{1 \leftarrow [l_i = \varsigma(b_i)^{\ i\in 1..n}]\} \\
[l_i = \varsigma(b_i)^{\ i\in 1..n}] \lhd \langle l_j = \varsigma(c) \rangle & \longrightarrow_{\varsigma_{DB}}\ [l_j = \varsigma(c),\ l_i = \varsigma(b_i)^{\ i\in 1..n, i\neq j}]
\end{array}
$$

Notice that substitution is still a meta-operation in this calculus, completely external to the reduction rules of the formalism.

4 The ς-Calculus à la de Bruijn with Fields ($\varsigma^{\bullet}_{DB}$-Calculus)

The $\varsigma^{\bullet}_{DB}$-calculus is a straightforward extension of ς_{DB}-calculus. It is formulated in preparation for the introduction of explicit substitutions in Section 5 and shall also be used for proving some properties of this calculus of explicit substitutions.

From a general standpoint an object may be regarded as an entity encapsulating state (fields) and behaviour (methods) in an object-oriented language . These methods allow the object to modify its local state as well as interact with other objects. Let us concentrate on fields. Consider an object `calculator` that possesses a field which allows the user (another object) to store some intermediate result. For this the object interface includes a method `save(n)` where `n` is the number to be stored. Also, in order to retrieve this value it includes a method `recall`. Thus one would expect the equation `calculator.save(n).recall=n` to be true. This is characteristic of the behaviour of fields. As mentioned in [1] the ς-calculus does not include field contructs as primitive. Nevertheless, methods that do not use the self variable may be regarded as fields. Indeed, let b be a term in the ς-calculus such that it has no occurrence of a variable x. Then we have $[l = \varsigma(x).b].l \longrightarrow_{\varsigma} b\{x \leftarrow [l = \varsigma(x).b]\} \equiv b$ Thus we obtain exactly b, the body of the method $l = \varsigma(x).b$.

Now consider the setting where variables are represented no longer by variable names but by de Bruijn indices. Then we could attempt to proceed as above. Consider a term b in the ς_{DB}-calculus such that $1 \notin FV(b)$. Then we have, $[l = \varsigma(b)].l \longrightarrow_{\varsigma_{DB}} b\{1 \leftarrow [l = \varsigma(b)]\} \equiv b^{-}$ where b^{-} represents b with free indices decremented in one unit. The result obtained is not the same as the body of the method $l = \varsigma(b)$.

Thus we may simulate fields in ς_{DB}-calculus by representing them as methods $l = \varsigma(b^{+})$ where b^{+} represents b where all free indices are incremented in one unit. Nevertheless, we shall introduce fields as primitive constructs in the language. The reason for doing so is that when explicit substitutions are introduced into the calculus and the translation of (an explicit substitution version of) the λ-calculus into this extension studied, field simulation may no longer be performed (c.f. Section 5).

Therefore in our de Bruijn setting we incorporate, as a primitive notion, that of a field. The terms of the ς-calculus à la de Bruijn with fields (hereafter the $\varsigma^{\bullet}_{DB}$-calculus), denoted $\mathcal{T}_{\varsigma^{\bullet}_{DB}}$, are called *pure terms* and are characterized by the following grammar:

$$\begin{array}{lll} a & ::= & \underline{p} \mid a.l \mid a \lhd \langle m \rangle \mid [m_i{}^{i \in 1..n}] \\ m & ::= & l = g \mid l := a \\ g & ::= & \varsigma(a) \end{array}$$

where p is a natural number greater than zero.

An object is constructed by a list of methods and fields. A method is denoted "$l = g$" where l is its label and g its body. A field is denoted "$l := a$" where l is its label and a its body. Note that we may override a method with a field and viceversa.

We now define the appropriate reduction rules using the notion of substitution defined above.

Definition 4 (Reduction in the $\varsigma^{\bullet}_{DB}$-calculus). *Reduction in the $\varsigma^{\bullet}_{DB}$-calculus is defined adding the following rewrite rules to the rewrite rules of Definition 3:*

$$[l_j := a, {m_i}^{i\in 1..n, i\neq j}].l_j \longrightarrow_{\varsigma^{\bullet}_{DB}} a$$
$$[{m_i}^{i\in 1..n}] \lhd \langle l_j := a\rangle \longrightarrow_{\varsigma^{\bullet}_{DB}} [l_j := a,\ {m_i}^{i\in 1..n, i\neq j}]$$

Notice that substitution is still a meta-operation in this calculus, completely external to the reduction rules of the formalism.

The $\varsigma^{\bullet}_{DB}$-calculus is confluent. This may be proved using the proof technique presented in [27], a variation of the Tait-and-Martin Löf technique. Also, via a translation function that adjusts appropriately the indices of the bodies of fields it may be proved that the ς_{DB}-calculus can simulate the $\varsigma^{\bullet}_{DB}$-calculus. For details the reader is referred to [6].

5 Fields and Explicit Substitutions

The ς-calculus with explicit substitutions and de Bruijn indices which we shall hereafter refer to as the ς_{DBES}-calculus is presented in this section. This calculus introduces two forms of substitution into the object language: ordinary substitution and invoke substitution. Also, the need for using explicit fields is explained.

5.1 The ς_{DBES}-Calculus

The set of terms of the ς_{DBES}-calculus, denoted $\mathcal{T}_{\varsigma_{DBES}}$, consists of terms of sort `Term` and terms of sort `Subst`. These are defined by the following grammar (sort `Term` to the left and sort `Subst` to the right)

$$a ::= \underline{p} \mid a.l \mid a \lhd \langle m\rangle \mid [{m_i}^{i\in\ 1..n}] \mid a[s]$$
$$m ::= \overline{l} = g \mid l := a$$
$$g ::= \varsigma(a) \mid g[s]$$
$$s ::= a/ \mid @l \mid \Uparrow(s) \mid \uparrow$$

where p is a natural number greater than zero.

Unless otherwise stated when we say that "a is a term in $\mathcal{T}_{\varsigma_{DBES}}$" we mean "$a$ is a term in $\mathcal{T}_{\varsigma_{DBES}}$ of sort `Term`". A *closure* is a term of the form $a[s]$. A term that does not contain occurrences of closures as subterms is called a *pure term*. A term $a[s]$ may be regarded as the term a with pending substitution s. The substitution operator .[.] is part of the calculus (at the object-level). A substitution s with an occurrence of $a/$ is called an *ordinary substitution* whereas a substitution s with an occurrence of $@l$ is called an *invoke substitution*. More on invoke substitutions shall be said in Section 7. Note that if we erase the grammar rules generating closures then we obtain the set $\mathcal{T}_{\varsigma^{\bullet}_{DB}}$.

The substitution grammar (and substitution subcalculus) for ordinary substitution is based on the calculus of explicit substitution for the lambda calculus, λ_v [20].

We shall frequently use the notation $\Uparrow^i (s)$ and $a[s]^i$ defined inductively as:

$$\begin{array}{ll} \Uparrow^0 (s) \;\; =_{def}\; s & a[s]^0 \;\; =_{def}\; a \\ \Uparrow^{i+1} (s) =_{def} \Uparrow (\Uparrow^i (s)) & a[s]^{i+1} =_{def} a[s]^i[s] \end{array}$$

The semantics of the ς_{DBES}-calculus is defined by the following rewrite rules:

$$\begin{array}{lll}
[l_j = \varsigma(a), m_i{}^{i\in 1..n, i\neq j}].l_j & \longrightarrow_{MI} & a[[l_j = \varsigma(a), m_i{}^{i\in 1..n, i\neq j}]/] \\
[l_j := a, m_i{}^{i\in 1..n, i\neq j}].l_j & \longrightarrow_{FI} & a \\
[m_i{}^{i\,\in\, 1..n}] \lhd \langle l_j = g\rangle & \longrightarrow_{MO} & [l_j = g,\ m_i^{i\in 1..n, i\neq j}] \quad j \in 1..n \\
[m_i{}^{i\,\in\, 1..n}] \lhd \langle l_j := a\rangle & \longrightarrow_{FO} & [l_j := a,\ m_i^{i\in 1..n, i\neq j}] \quad j \in 1..n \\
(\varsigma(c))[s] & \longrightarrow_{SM} & \varsigma(c[\Uparrow (s)]) \\
[m_i{}^{i\in\, 1..n}][s] & \longrightarrow_{SO} & [m_i[s]\ {}^{i\in\, 1..n}] \\
(l := a)[s] & \longrightarrow_{SF} & l := a[s] \\
(l = g)[s] & \longrightarrow_{SB} & l = g[s] \\
a.l[s] & \longrightarrow_{SI} & a[s].l \\
a \lhd \langle m\rangle[s] & \longrightarrow_{SU} & a[s] \lhd \langle m[s]\rangle \\
\underline{1}[a/] & \longrightarrow_{FVar} & a \\
\underline{p+1}[a/] & \longrightarrow_{RVar} & \underline{p} \\
\underline{1}[@l] & \longrightarrow_{FInv} & \underline{1}.l \\
\underline{p+1}[@l] & \longrightarrow_{RInv} & \underline{p+1} \\
\underline{1}[\Uparrow (s)] & \longrightarrow_{FVarLift} & \underline{1} \\
\underline{p+1}[\Uparrow (s)] & \longrightarrow_{RVarLift} & \underline{p}[s][\uparrow] \\
\underline{p}[\uparrow] & \longrightarrow_{VarShift} & \underline{p+1} \\
a[\Uparrow^i (@l_j)][\Uparrow^i ([l_j := b, m_i{}^{i\in 1..n, i\neq j}]/)] & \longrightarrow_{CO} & a[\Uparrow^i (b/)] \\
a[\Uparrow^i (@l)][\Uparrow^k (s)] & \longrightarrow_{SW} & a[\Uparrow^k (s)][\Uparrow^i (@l)] \qquad k > i
\end{array}$$

The rule *MI* activates a method invocation. The rule *FI* activates a field invocation. The rules MO, FO activate method override and field override respectively. Rules SM, SO, SF, SB, SI, SU allow the propagation of the substitution operator through method body, object, field, method, invocation and override constructors. Rules $FVar, RVar, Finv, RInv, FVarLift, RVarLift, VarShift$ allow the computation of substitutions on indices. Finally, the rule *CO* expresses a form of interaction of substitutions, and *SW* expresses a (weak) form of commutation or switching of substitutions. These two rules will be used in simulating λ_v in the ς_{DBES}-calculus.

It is interesting to compare rules *RVar* and *RInv*. The creation of a substitution of the form $b/$ is accompanied by the elimination of a binder (see rule *MI*). Hence all "free" indices should be decremented in one unit. Whereas in the case of the invoke substitution operator "@" no such adjustment is made.

The ς_{DBES}-calculus without the rules MI, MO, FI and FO is referred to as the *ESDB* rewriting system. Note that *ESDB* is not locally confluent since for example the term $\underline{1}[@l_1][[l_1 := b]/]$ reduces to two different terms by the rules

FInv and *CO* respectively, and requires *FI* to close the diagram. The *ESDB* rewriting system is responsible for performing or discarding the substitution operators and additionally allows for some interaction between ordinary and invoke substitution operators. The rewriting system obtained by eliminating the rules for substitution interaction (rules *CO* and *SW*) is called the *BES* (Basic Explicit Substitution) rewriting system. This system suffices for executing substitutions; the interaction rules shall be needed when simulating λ_υ.

5.2 The Need for Explicit Fields

In Section 4 we saw that although the $\varsigma^\bullet_{DB}$-calculus incorporated fields as primitive constructs this is not strictly necessary as fields may be simulated in the ς_{DB}-calculus in a rather natural way. This situation no longer holds when explicit substitutions are introduced and when we attempt to translate the λ_υ-calculus into the $\varsigma^\bullet_{DBES}$-calculus using the translation in [1] (recalled in Section 2) for the λ-calculus.

Let us ignore fields as a primitive construct in the language for the moment and return to our simulation of fields as discussed in Section 4. A field b is represented as the method $l = \varsigma(b^+)$. The ς_{DBES}-calculus is then reduced to the, say, $\varsigma^\bullet_{DBES}$, where rules *FI*, *FO*, *SF* and *CO* have been eliminated.

Now when we attempt to translate the λ_υ-calculus into the $\varsigma^\bullet_{DBES}$-calculus in the style of $\prec\!\prec$. $\succ\!\succ$ we arrive naturally to the translation function k:

$$
\begin{array}{lll}
k(a/) & =_{def}\ k(a)/ & \qquad k(\underline{p}) =_{def} \underline{p} \\
k(\Uparrow(s)) & =_{def}\ \Uparrow(k(s)) & \qquad k(\uparrow) =_{def} \uparrow \\
k(a[s]) & =_{def}\ k(a)[k(s)] & \\
k(ab) & =_{def}\ (k(a) \lhd \langle arg = \varsigma(k(b)^+)\rangle).val & \\
k(\lambda a) & =_{def}\ [arg = \varsigma(1.arg), val = \varsigma(k(a)[@arg])] &
\end{array}
$$

But the meaning of $k(b)^+$ is no longer clear since $k(b)$ may have occurrences of the explicit substitution operator (it is no longer a pure term). To remedy this situation the next logical step would be to introduce an "explicit substitution version" of the $.^+$ operator which in fact we already have: the $\uparrow$ (shift) operator. Indeed, it is with the aid of the shift operator that updating is implemented explicitly (cf. Section 7 in [6]). The final clause of the definition of k is now replaced by $k(ab) =_{def} (k(a) \lhd \langle arg = \varsigma(k(b)[\uparrow])\rangle).val$

So now we proceed to verify that the translation is correct (preserves λ_υ-reduction). Consider for example the λ_υ-reduction rule $(\lambda a)b \longrightarrow_{Beta} a[b/]$ (cf. Section 6). Then we must have $k((\lambda a)b) \xrightarrow{*}_{\varsigma^\bullet_{DBES}} k(a[b/])$. We can go as far as:

$$
\begin{array}{ll}
k((\lambda a)b) & =_{def} \\
([arg = \varsigma(1.arg), val = \varsigma(k(a)[@arg])] \lhd \langle arg = \varsigma(k(b)[\uparrow])\rangle).val & \longrightarrow_{MO} \\
[arg = \varsigma(k(b)[\uparrow]), val = \varsigma(k(a)[@arg])].val & \longrightarrow_{MI} \\
k(a)[@arg][[arg = \varsigma(k(b)[\uparrow]), val = \varsigma(k(a)[@arg])]/] &
\end{array}
$$

Thus in order to arrive at $k(a)[k(b)/]$ we are in need of adding to the $\varsigma^\bullet_{DBES}$-calculus a commutation rule of the form: $a[\Uparrow^i (@l_j)][\Uparrow^i ([l_j = \varsigma(b[\uparrow$

$]), {m_i}^{i\in 1..n, i\neq j}]/)] \longrightarrow_{COM} a[\Uparrow^i (b/)]$ (taking $i = 0$ suffices for our example). But adding a rule like *COM* clearly introduces confluence problems.

A variant could be the rule $a[\Uparrow^i (@l_j)][\Uparrow^i ([l_j = \varsigma(b), {m_i}^{i\in 1..n, i\neq j}]/)] \longrightarrow_{COM'} a[\Uparrow^i (c/)]$ where $b =_{BES} c[\uparrow]$. The major drawbacks are then the fact that the rule is conditional and (computationally) expensive checking on the equational substitution theory is required (this resembles problems studied when dealing with η-contraction in explicit substitution calculi ([7],[25], [16])).

These problems stem from the fact that the formulation of rules which are subject to restrictions on the free variables in a de Bruijn index setting and in the presence of explicit substitutions is non trivial. Here, we have solved these issues by a minor change in the syntax so as to represent fields as primitive operators. In fact, the rewrite rule *CO* of the *named* ς_{ES}-calculus presented in [17] is conditional, whereas the *CO* rule presented in this work, in a *de Bruijn index* setting, is actually simpler since no condition is present.

6 Encoding λ_υ-Terms in the ς_{DBES}-Calculus

In this section we will show how to simulate the explicit substitution calculus for the lambda calculus λ_υ[20] in the ς_{DBES}-calculus. We start by augmenting the grammar productions for the terms of the ς_{DBES}-calculus in order to allow abstractions and applications as legal terms. We then define a translation from terms in the λ_υ-calculus into this augmented set of terms which preserves reduction. We recall the main definitions of the λ_υ-calculus. Terms are defined by the following grammars $t ::= \underline{p} \mid tt \mid \lambda t \mid t[s]$ and $s ::= \uparrow \mid t/ \mid \Uparrow (s)$. We recall the rules below.

$$
\begin{array}{llll}
(\lambda a)b & \longrightarrow_{Beta} & a[b/] & \underline{p+1}[a/] \longrightarrow_{RVar} \underline{p} \\
(a\ b)[s] & \longrightarrow_{app} & a[s]b[s] & \underline{1}[\Uparrow (s)] \longrightarrow_{FVarLift} \underline{1} \\
\lambda a[s] & \longrightarrow_{abs} & \lambda(a[\Uparrow (s)]) & \underline{p+1}[\Uparrow (s)] \longrightarrow_{RVarLift} \underline{p}[s][\uparrow] \\
\underline{1}[a/] & \longrightarrow_{FVar} & a & \underline{p}[\uparrow] \longrightarrow_{VarShift} \underline{p+1}
\end{array}
$$

The mixed set of terms, which we shall call $\mathcal{T}_{\lambda\varsigma_{DBES}}$ consists of the terms of sort `Term` and terms of sort `Subst` (which remain unaltered). The terms of sort `Term` are defined by the following grammar:

$$
\begin{array}{l}
a ::= \underline{p} \mid a.l \mid a \lhd \langle m \rangle \mid [{m_i}^{i\in\ 1..n}] \mid a[s] \mid \lambda a \mid (a\ a) \\
m ::= \overline{l} = g \mid l := a \\
g ::= \varsigma(a) \mid g[s]
\end{array}
$$

where p is any natural number greater than zero.

The rewrite rules of the $\lambda\varsigma_{DBES}$-calculus consists of the rewrite rules of the ς_{DBES}-calculus together with the rules *Beta*, *abs* and *app* of the λ_υ-calculus (note that the remaining rules of λ_υ already belong to the ς_{DBES}-calculus). The resulting system may be proved confluent using the interpretation technique [12] and the fact that the corresponding system with meta-level substitutions is an orthogonal rewrite system.

The encoding of λ_υ-terms into $\lambda\varsigma_{DBES}$-terms makes use of the invoke explicit substitution operator "@" and fields.

Definition 5 (Translation of $\lambda\varsigma_{DBES}$-terms into terms in $\mathcal{T}_{\varsigma_{DBES}}$). *The translation $\prec . \succ$ from $\lambda\varsigma_{DBES}$-terms into terms in $\mathcal{T}_{\varsigma_{DBES}}$ is defined as*

$$
\begin{array}{llll}
\prec \underline{p} \succ & =_{def} \underline{p} & \prec \varsigma(a) \succ & =_{def} \varsigma(\prec a \succ) \\
\prec a.l \succ & =_{def} \prec a \succ .l & \prec a/ \succ & =_{def} \prec a \succ / \\
\prec a \lhd \langle m \rangle \succ & =_{def} \prec a \succ \lhd \langle \prec m \succ \rangle & \prec \Uparrow (s) \succ & =_{def} \Uparrow (\prec s \succ) \\
\prec [m_i{}^{\,i \in 1..n}] \succ & =_{def} [\prec m_i \succ {}^{\,i \in 1..n}] & \prec \uparrow \succ & =_{def} \uparrow \\
\prec l = g \succ & =_{def} l =\prec g \succ & \prec @l \succ & =_{def} @l \\
\prec l := a \succ & =_{def} l :=\prec a \succ & \prec \lambda a \succ & =_{def} c \\
\prec a[s] \succ & =_{def} \prec a \succ [\prec s \succ] & \prec ab \succ & =_{def} \prec a \succ \bullet \prec b \succ
\end{array}
$$

where c is $[arg = \varsigma(1.arg), val = \varsigma(\prec a \succ [@arg])]$ and $p \bullet q \ =_{def} (p \lhd \langle arg := q \rangle).val$

The translation interprets the lambda expressions abstraction and application into objects leaving the rest of the constructions without modifications. The translation of an abstraction introduces the invoke substitution. Note that the index level 1 (to which the invoke substitution applies) is bound. This reveals a difference as regards the behaviour of ordinary and invoke substitutions, as discussed above. Ordinary substitution is of no use since its index adjusting mechanism does not exhibit the desired behaviour.

The principal motivation behind the introduction of the rules describing the interaction of ordinary substitution and invoke substitution lies in the following proposition.

Proposition 1 (ς_{DBES} simulates λ_v). *If $a \longrightarrow_{\lambda v} b$ then $\prec a \succ \xrightarrow{*}_{\varsigma_{DBES}} \prec b \succ$.*

Proof. The proof is done by structural induction on the λ_v-term. The key cases are $\prec (\lambda a)b \succ \xrightarrow{*}_{\varsigma_{DBES}} \prec a[b/] \succ$ (Case 1) and $\prec \lambda a[s] \succ \xrightarrow{*}_{\varsigma_{DBES}} \prec \lambda(a[\Uparrow (s)]) \succ$ (Case 2).

Case 1.

$$
\begin{array}{ll}
\prec (\lambda a)b \succ & =_{def} \\
([arg = \varsigma(1.arg), val = \varsigma(\prec a \succ [@arg])] \lhd \langle arg :=\prec b \succ \rangle).val & \longrightarrow_{MO} \\
[arg :=\prec b \succ, val = \varsigma(\prec a \succ [@arg])].val & \longrightarrow_{MI} \\
\prec a \succ [@arg][[arg :=\prec b \succ, val = \varsigma(\prec a \succ [@arg])]/] & \longrightarrow_{CO} \\
\prec a \succ [\prec b \succ /] & =_{def} \\
\prec a[b/] \succ &
\end{array}
$$

Case 2.

$$
\begin{array}{ll}
\prec (\lambda a)[s] \succ & =_{def} \\
[arg = \varsigma(1.arg), val = \varsigma(\prec a \succ [@arg])][\prec s \succ] & \longrightarrow_{SO} \\
[arg = (\varsigma(1.arg))[\prec s \succ], val = (\varsigma(\prec a \succ [@arg]))[\prec s \succ]] & \xrightarrow{*}_{BES} \\
[arg = \varsigma(1.arg[\Uparrow (\prec s \succ)]), val = \varsigma(\prec a \succ [@arg][\Uparrow (\prec s \succ)])] & \xrightarrow{*}_{BES} \\
[arg = \varsigma(1.arg), val = \varsigma(\prec a \succ [@arg][\Uparrow (\prec s \succ)])] & \longrightarrow_{SW} \\
[arg = \varsigma(1.arg), val = \varsigma(\prec a \succ [\Uparrow (\prec s \succ)][@arg])] & =_{def} \\
\prec \lambda(a[\Uparrow (s)]) \succ &
\end{array}
$$

We may therefore conclude that λ_v-derivations may be translated into ς_{DBES}-reductions sequences, thereby implementing objects and functions at the same time.

7 Confluence and PSN of the ς_{DBES}-Calculus

When dealing with calculi of explicit substitutions some basic properties have to be considered, namely, strong normalization of the substitution calculus ($ESDB$), confluence of the full calculus and preservation of strong normalization, that is, that every strongly normalizing term in $\varsigma^{\bullet}_{DB}$-calculus must also be strongly normalizing in the ς_{DBES}-calculus. The history of calculi of explicit substitution has revealed that this last property is by no means trivial. One of the first calculi of explicit substitution for the λ-calculus, called λ_σ [2] introduced in 1991 was long believed to satisfy the aforementioned property. Surprisingly in 1995 Melliès provided a counterexample [22], exhibiting a (pure typable) term that is strongly β-normalizing yet admits an infinite λ_σ-reduction sequence. Since we allow some interaction between substitutions this property is essential in our current setting.

Strong normalization of the substitution calculus is obtained by the polynomial interpretation technique. As for the confluence of the ς_{DBES}-calculus we have the following result which is proved using the Interpretation Method [12].

Proposition 2. *The ς_{DBES}-calculus is confluent.*

Proving preservation of strong normalization is more complicated. We shall obtain the desired result by using a technique introduced by Bloo and Geuvers in [4]. As remarked before this property is an essential ingredient in any explicit substitution implementation of a calculus, more so if there is some form of interaction between substitutions as is our case.

Definition 6 (Strongly normalising pure terms of $\mathcal{T}_{\varsigma_{DBES}}$). *Let $SN_{\varsigma^{\bullet}_{DB}}$ denote the set of all the $\varsigma^{\bullet}_{DB}$-strongly normalizing pure terms of $\mathcal{T}_{\varsigma_{DBES}}$. Then we may define $\mathcal{F}$ as $\mathcal{F} = \{a \in \mathcal{T}_{\varsigma_{DBES}} |$ for all $b \subseteq a$ of sort $\mathtt{Term}, BES(b) \in SN_{\varsigma^{\bullet}_{DB}}\}$.*

The notation $b \subseteq a$ is used to denote that b is a subterm of a. Next we show that $\mathcal{F}$ is closed with respect to reduction in the ς_{DBES}-calculus.

Lemma 1. *Let $a, b \in \mathcal{T}_{\varsigma^{\bullet}_{DB}}$. If $a \in \mathcal{F}$ and $a \longrightarrow_{\varsigma_{DBES}} b$ then $b \in \mathcal{F}$.*

Proof. We show that for every $e \subseteq b$ we have $BES(e) \in SN_{\varsigma^{\bullet}_{DB}}$. The proof is by induction on a.

Definition 7 (Labelled terms). *We define the set of terms $\mathcal{T}_l$ over the alphabet $\mathcal{A} = \{\star, \circ, ..{}_n., \lhd(.,.), . < . >_n, .[\![.]\!]_n, \varsigma(.), [.], =, :=\}$ and for n a natural number greater or equal to zero, by the following grammar:*

$$
\begin{array}{l}
t ::= \star \mid t.{}_n\circ \mid t < t >_n \mid t[\![\circ]\!]_n \mid \lhd(t,u) \mid [{u_i}^{i \in 1..n}] \\
u ::= \circ = f \mid \circ := t \\
f ::= \varsigma(t) \mid f < t >_n
\end{array}
$$

Definition 8 (Translation from $\mathcal{F}$ to $\mathcal{T}_l$). *The translation $\mathcal{S}(.) : \mathcal{F} \to \mathcal{T}_l$ is defined as follows:*

$$
\begin{array}{lll}
\mathcal{S}(\underline{p}) & =_{def} \star & \\
\mathcal{S}([\overline{m_i^{\,i\,\in 1..n}}]) & =_{def} [\mathcal{S}(m_i)^{i\,\in 1..n}] & \\
\mathcal{S}(l = g) & =_{def} \mathcal{S}(l) = \mathcal{S}(g) & \\
\mathcal{S}(l := a) & =_{def} \mathcal{S}(l) := \mathcal{S}(a) & \\
\mathcal{S}(\varsigma(a)) & =_{def} \varsigma(\mathcal{S}(a)) & \\
\mathcal{S}(a.l) & =_{def} \mathcal{S}(a).{}_n\mathcal{S}(l) & \text{where } n = maxred_{\varsigma^{\bullet}_{DB}}(BES(a.l)) \\
\mathcal{S}(a \lhd \langle m \rangle) & =_{def} \lhd(\mathcal{S}(a), \mathcal{S}(m)) & \\
\mathcal{S}(a[\Uparrow^i (b/)]) & =_{def} \mathcal{S}(a) < \mathcal{S}(b) >_n & \text{where } n = maxred_{\varsigma^{\bullet}_{DB}}(BES(a[\Uparrow^i (b/)])) \\
\mathcal{S}(a[\Uparrow^i (\uparrow)]) & =_{def} \mathcal{S}(a) & \\
\mathcal{S}(a[\Uparrow^i (@l)]) & =_{def} \mathcal{S}(a)[\![\mathcal{S}(l)]\!]_n & \text{where } n = maxred_{\varsigma^{\bullet}_{DB}}(BES(a[\Uparrow^i (@l)]))
\end{array}
$$

where $\mathcal{S}(l) = \circ$.

We define a precedence (partial ordering) on the set of operators of $\mathcal{A}$ as follows: $..{}_{n+1}. \gg . < . >_n \gg .[\![.]\!]_n \gg ..{}_n. \gg \lhd(.,.) \gg \varsigma(.), =, :=, [], \star, \circ$. Then since $\gg$ is well-founded the induced Recursive Path Ordering (RPO) '$\succ_{\mathcal{T}_l}$' defined below is well-founded on $\mathcal{T}_l$ [10].

Lemma 2. *Let $a \in \mathcal{F}$. Then $a \longrightarrow_R a'$ implies $\mathcal{S}(a) \succ_{\mathcal{T}_l} \mathcal{S}(a')$ where $R = \{MI, FI, MO, FO\}$ and $\mathcal{S}(a) \succeq_{\mathcal{T}_l} \mathcal{S}(a')$ if $R = \varsigma_{DBES} - \{MI, FI, MO, FO\}$.*

Proof. The proof is by structural induction on a using lemmas 1 and additional technical lemmata (see [6]).

We may now prove the main proposition of this section, namely, the proposition of preservation of strong normalization for the ς_{DBES}-calculus.

Proposition 3 (PSN of the ς_{DBES}-calculus). *The ς_{DBES}-calculus preserves strong normalization.*

Proof. Suppose that the ς_{DBES}-calculus does not preserve strong normalization. Thus there is a pure term a which is strongly $\varsigma^{\bullet}_{DB}$-normalizing but which possesses an infinite reduction sequence in the ς_{DBES}-calculus. Since the rewriting system $S \equiv ESDB \cup \{MO, FO, FI\}$ is strongly normalizing [6] this reduction sequence must have the form $a \equiv a_1 \xrightarrow{*}_S a_2 \longrightarrow_{MI} a_3 \xrightarrow{*}_S a_4 \longrightarrow_{MI} a_5 \ldots$ where the reductions $a_{2k} \longrightarrow_{MI} a_{2k+1}$ for $k \geq 1$ occur infinitely many times. Now since a is in $\mathcal{F}$, and since by lemma 1 the set $\mathcal{F}$ is closed under reduction in $\varsigma^{\bullet}_{DB}$ we obtain an infinite sequence

$$\mathcal{S}(a) \equiv \mathcal{S}(a_1) \succeq_{\mathcal{T}_l} \mathcal{S}(a_2) \succ_{\mathcal{T}_l} \mathcal{S}(a_3) \succeq_{\mathcal{T}_l} \mathcal{S}(a_4) \succ_{\mathcal{T}_l} \mathcal{S}(a_5) \ldots$$

This contradicts the well-foundedness of the recusive path ordering $\succ_{\mathcal{T}_l}$.

8 Conclusions and Future Work

We have proposed a first order calculus based on de Bruijn indices and explicit substitutions for implementing objects and functions. The encoding of functions

as objects in the spirit of [1] has led us to consider fields as primitive constructs in the language. The resulting calculus has been shown to correctly simulate the object calculus (ς) and the function calculus (λ_v), and also that it satisfies the properties of confluence and preservation of strong normalization. As in [17] two different forms of substitution are present in the calculus: ordinary substitution and invoke substitution. In the named variable calculus presented in [17], called ς_{ES}, this distinction is based on constraints associated with types (in an invoke substitution the type of the method invocation $x.l$ to be substituted for x differ) and the free variable property. In fact, since ς_{ES} is untyped the type constraint may be minimized. In contrast, and as already hinted in [17], in the ς_{DBES}-calculus this distinction is based on different index adjusting mecanisms, thus fully justifying the need for different substitution operators.

Interaction between substitutions such as composition or permutation of substitutions usually renders the property of preservation of strong normalization non trivial. Indeed since a weak form of interaction between both forms of substitutions is present in the ς_{DBES}-calculus the proof of the property of preservation of strong normalization has resulted a key issue.

Finally, rules possessing conditions on free variables in a named variable setting generally pose problems when expressed in a de Bruijn indice setting as may be seen for example when dealing with η-reduction ([7], [25], [16]). The use of fields as a primitive construct has allowed us to replace the conditional rules present in the ς_{ES} with non-conditional rules, thus simplifying the resulting calculus.

As already discussed the ς_{DBES}-calculus is not an instance of the de Bruijn index based higher order rewriting formalism XRS[1] [24]. XRSs provide a fixed substitution calculus ($\sigma_{\Uparrow}$) for computing ordinary substitutions. Thus an interesting approach is to generalize this framework to a formalism where various forms of substitution may be defined with possible interaction between them.

Also, in view of the importance of the typing discipline the consideration of type systems for the ς_{DBES}-calculus is required.

Acknowledgements. I would like to thank Delia Kesner, Alejandro Ríos and Pablo E. Martínez López for valuable discussions, advice and encouragement, and the anonymous referees for their comments.

References

1. M.Abadi and L.Cardelli. *A Theory of Objects.* Springer Verlag, 1996.
2. M.Abadi, L.Cardelli, P.-L.Curien, and J.-J.Lévy. *Explicit Substitutions.* Journal of Functional Programming, 4(1):375-416, 1991.
3. R.Bloo. *Preservation of Termination for Explicit Substitution.* PhD. Thesis, Eindhoven University, 1997.

[1] Strictly speaking it is a first order rewriting formalism but allows to express binding mecanisms.

4. R.Bloo and H.Geuvers. *Explicit Substitution: on the edge of strong normalization.* Theoretical Computer Science, 204, 1998.
5. R.Bloo, K.Rose. *Combinatory Reduction Systems with explicit substitution that preserve strong normalization.* In RTA'96, LNCS 1103, 1996.
6. E. Bonelli. *Using fields and explicit substitutions to implement objects and functions in a de Bruijn setting.* Full version obtainable by ftp at `ftp://ftp.lri.fr/LRI/articles/bonelli/objectdbfull.ps.gz`.
7. D.Briaud. *An explicit eta rewrite rule.* In M.Dezani Ed., Int. Conference on Typed Lambda Calculus and Applications, LNCS vol. 902, 1995.
8. N.G.de Bruijn. *Lambda calculus notation with nameless dummies, a tool for automatic formual manipulation with application to the Church-Rosser theorem.* Koninklijke Nederlandse Akademie van Wetenschappen, Series A, Mathematical Sciences, 75:381-392, 1972.
9. P.-L.Curien, T.Hardin, and J.-J.Lévy. *Confluence properties of weak and strong calculi of explicit substitutions.* Technical Report, Centre d'Etudes et de Recherche en Informatique, CNAM, 1991.
10. N. Dershowitz. *Orderings for term rewriting systems.* Theoretical Computer Science, 17(3):279-301, 1982.
11. M.Ferreira, D.Kesner. and L.Puel. *Lambda-calculi with explicit substitutions and composition which preserve beta-strong normalization.* Proceedings of the 5th International Conference on Algebraic and Logic Programming (ALP'96), LNCS 1139, 1996.
12. T.Hardin. *Résultats de confluence pour les règles fortes de la logique combinatoire catégorique et liens avec les lambda-calculs.* Thèse de doctorat, Université de Paris VII, 1987.
13. T.Hardin and J-J.Lévy. *A confluent calculus of substitutions.* In France-Japan Artificial Intelligence and Computer Science Symposium, 1989.
14. F. Kamareddine and A.Ríos. *A lambda calculus a la de Bruijn with Explicit Substitutions.* Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'95), LNCS 982, 1995.
15. F. Kamareddine and A.Ríos. *Extending a λ-calculus with Explicit Substitutions which Preserves Strong Normalization into a Confluent Calculus on Open Terms.* In Journal of Functional Programming, Vol.7 No.4 , 1997.
16. D.Kesner. *Confluence properties of extensional and non-extensional lambda-calculi with explicit substitutions.* Proceedings of the 7th International Conference on Rewriting Techniques and Applications (RTA'96), LNCS 1103, 1996.
17. D.Kesner, P.E.Martínez López. *Explicit Substitutions for Objects and Functions.* Proceedings of the Joint International Symposiums: Programming Languages, Implementations, Logics and Program (PLILP'98) and Algebraic and Logic Programming (ALP), LNCS 1490,pp. 195-212, Sept. 1998.
18. J.W.Klop. *Combinatory Reduction Systems.* PhD Thesis, University of Utrecht, 1980.
19. J.W.Klop. *Term Rewriting Systems.* In S. Abramsky, D. Gabbay, and T.Maibaum, editors, Handbook of Logic in Computer Science, Volume II, Oxford University Press, 1992.
20. P. Lescanne. *From λ_σ to λ_υ, a journey through calculi of explicit substitutions.* In Ann. ACM Symp. on Principles of Programming Languages (POPL), pp. 60-69. ACM, 1994.
21. P. Lescanne and J.Rouyer Degli. *Explicit substitutions with de Bruijn's levels.* In P.Lescanne editor, RTA'95, LNCS 914, 1995.

22. P.A.Melliès. *Typed λ-calculi with explicit substitutions may not terminate.* In TLCA'95, LNCS 902, 1995.
23. C.A.Muñoz. *Confluence and Preservation of Strong Normalisation in an Explicit Substitutions Calculus.* Proceedings of the Eleven Annual IEEE Symposium on Logic in Computer Science, 1996.
24. B.Pagano. *Des calculs de substitution explicit et leur application à la compilation des langages fonctionnels.* Thèse de doctorat, Université de Paris VII, 1998.
25. A. Ríos. *Contribution à l'étude des λ-calculs avec substitutions explicites.* Ph.D Thesis, Université de Paris VII, 1993.
26. K.Rose. *Explicit Cyclic Substitutions.* In CTRS'92, LNCS 656, 1992.
27. M. Takahashi. *Parallel reduction in the λ-calculus.* Journal of Symbolic Computation, 7:113-123, 1989.

Closed Reductions in the λ-Calculus (Extended Abstract)

Maribel Fernández[1] and Ian Mackie[2]

[1] LIENS (CNRS UMR 8548), École Normale Supérieure
45 Rue d'Ulm, 75005 Paris, France
maribel@dmi.ens.fr
[2] CNRS-LIX (UMR 7650), École Polytechnique
91128 Palaiseau Cedex, France
mackie@lix.polytechnique.fr

Abstract. Closed reductions in the λ-calculus is a strategy for a calculus of explicit substitutions which overcomes many of the usual syntactical problems of substitution. This is achieved by only moving closed substitutions through certain constructs, which gives a weak form of reduction, but is rich enough to capture the usual strategies in the λ-calculus (call-by-value, call-by-need, etc.) and is adequate for the evaluation of programs. An interesting point is that the calculus permits substitutions to move through abstractions, and reductions are allowed under abstractions, if certain conditions hold. The calculus naturally provides an efficient notion of reduction (with a high degree of sharing), which can easily be implemented.

1 Introduction

It is well known that substitution in the -calculus is a meta-operation, de ned outside of the syntax of the system. Equally, it is well known that the substitution process is a very delicate operation that may require the use of -conversion, and it may cause terms, redexes and potential redexes to be duplicated or erased.

In recent years a whole range of explicit substitution calculi have been proposed, starting from the -calculus [1], with the general aim of making the substitution process exist at the same level as -reduction. In general, the main motivation for such calculi is to have a handle on the process of substitution, and to be able to control it in various ways. There are however two distinct applications for these calculi. First, from a term rewriting perspective, the goal is to capture -reduction as a rst order term rewriting system. Here simulation of -reduction, preservation of termination and con uence seem to be the main issues. Secondly, from an implementation perspective, the goal is to explain in low level terms the process of -reduction which can provide a basis for abstract machines and implementations of functional programming languages. Although the second point is used as a motivating factor in most studies of explicit substitutions, we nd that the rst point has had the most attention. In this paper we

J. Flum and M. Rodr guez-Artalejo (Eds.): CSL 99, LNCS 1683, pp. 220 234, 1999.

focus on the second aspect. In particular we are not directly interested in simulating □-reduction in full generality, but rather to use the substitution mechanism to provide a simple and e□ cient form of reduction.

Our point of departure is a calculus of explicit substitutions with names, which simply adds the meta-level operation as a collection of conditional rewrite rules. We then proceed in a thorough-going fashion to systematically remove all the problematic and expensive rules of this system by giving a strategy for reduction. It turns out that this can be achieved by permitting certain reductions to take place only when a sub-term is *closed* (no free variables). In particular, this eliminates variable clash and capture problems, and removes the necessity of generating fresh variable names during reduction, which overcomes the main objection for explicit substitution calculi with names.

Almost all evaluators for the □-calculus are based on reduction to weak head normal form, which is characterized by rejecting reduction under an abstraction. There are many ways to obtain such normal forms, the most common are call-by-need and call-by-value reduction. In the language of explicit substitutions, this weak form of reduction is often interpreted as not pushing substitutions through an abstraction [3]. This form of weak reduction has the convenience that the most awkward part of the substitution process is removed from the system (prohibiting substitution through an abstraction avoids name clashes and □-conversions). However this bene□t is achieved at a price because terms with substitutions (closures) may be copied, which can cause redexes to be duplicated. In our calculus we address this problem by allowing closed substitutions to be made, and moreover we never copy a term (or closure) which contains a free variable.

The □-calculus, in addition to substitution, lacks explicit information about sharing and evaluation orders. This point becomes more subtle when one considers it in the framework of explicit substitutions, since the order in which substitutions are performed can have dramatic consequences on the e□ ciency of the reduction process. To ensure that we have a tight control over the way substitutions are performed, we also make explicit the copying and erasing phases of substitution, which are inspired by various calculi for linear logic. This will also allow us to control (and avoid) the issues of duplicating and erasing free variables in the substitution process.

In summary, we present a calculus of explicit substitutions with explicit resource management, where the emphasis is on obtaining answers in a simple and e□ cient way. The key aspect of the reduction strategy used is that of closed reduction, which provides a mechanism to allow easily: reduction under, and substitution through, an abstraction; avoidance of duplication of free variables; and clean garbage collection. An implementation of closed reduction using interaction nets is presented in [8] together with benchmarks and empirical comparisons with some other implementations of the □-calculus.

Related Work. Our work builds upon the use of explicit substitutions for controlling substitutions in the □-calculus, with an emphasis on implementation, for instance the call-by-need □-calculus of Ariola et al. [2] and calculi with shared envi-

ronments of Yoshida [13]. Also oriented towards implementation are [5] and [12]. The notion of closed reduction for the □-calculus was inspired by a strategy for cut-elimination in linear logic, used in a proof of soundness of the geometry of interaction by Girard [4].

Overview. In the following section we motivate our work by looking at the problematic issues in explicit substitution calculi. In Section 3 we present the closed reduction calculus, called $\Box_c$. In Section 4 we study several properties of the calculus. Section 5 gives a type system for the calculus and we show subject reduction and termination. The relation with □-reduction and strategies in the □-calculus is given in Section 6. Section 7 gives an alternative presentation of our calculus inspired by director strings. We conclude the paper in Section 8.

2 Motivation: Calculi and Strategies

The □rst version of the calculus is obtained by simply making the meta-operation of substitution part of the syntax, and adding explicit conditional rules for the propagation of substitutions. These rules are inspired by the original de□nition of substitution given by Church. By analyzing these rules for substitution from a very syntactic perspective we will identify a series of re□nements that will lead to the □nal calculus that we call $\Box_c$. We shall use explicit substitution calculi with names throughout, but most of what we have to say can be formulated in terms of the de Bruijn notation.

De□nition 1 (□-calculus with names and explicit substitutions). *Terms are built from the grammar:* $t ::= x \mid \Box x.t \mid (tt) \mid t[t/x]$ *with* $x^c\, y^c\, z$ *ranging over variables, and* $t^c\, u^c\, v$ *ranging over terms. We have the following conditional reduction rules:*

Name	*Reduction*		*Condition*
Beta	$(\Box x.t)u$	$\rightsquigarrow t[u/x]$	
Var1	$x[v/x]$	$\rightsquigarrow v$	
Var2	$y[v/x]$	$\rightsquigarrow y$	$x \neq y$
App	$(tu)[v/x]$	$\rightsquigarrow (t[v/x])(u[v/x])$	
Lam1	$(\Box y.t)[v/y]$	$\rightsquigarrow (\Box y.t)$	
Lam2	$(\Box y.t)[v/x]$	$\rightsquigarrow \Box y.t[v/x]$	$x \notin \mathsf{fv}(t) \vee y \notin \mathsf{fv}(v)^c\, x \neq y$
Lam3	$(\Box y.t)[v/x]$	$\rightsquigarrow \Box z.(t[z/y])[v/x]$	$x \notin \mathsf{fv}(t)^c\, y \vee \mathsf{fv}(v)^c\, x \neq y^c\, z$ *fresh*
Comp	$(t[u/x])[v/y]$	$\rightsquigarrow (t[v/y])[u[v/y]/x]$	

Five rules cause the real computational work: *Var2* and *Lam1* discard the term v, *App* and *Comp* duplicate the term v, and *Lam3* requires an additional substitution (renaming) to avoid variable capture and variable clash. We now look at ways of reducing these overheads.

Our □rst improvement is to direct substitutions so that terms are only propagated to the places where they are actually required, avoiding making copies

that will later be discarded. For instance, the *App* rule can be replaced by the rules:

$$\begin{array}{ll}(tu)[v\lhd x] \rightsquigarrow (t[v\lhd x])u & x \in \mathsf{fv}(t) \wedge x \notin \mathsf{fv}(u)\\ (tu)[v\lhd x] \rightsquigarrow t(u[v\lhd x]) & x \in \mathsf{fv}(u) \wedge x \notin \mathsf{fv}(t)\\ (tu)[v\lhd x] \rightsquigarrow (tu) & x \notin \mathsf{fv}(tu)\\ (tu)[v\lhd x] \rightsquigarrow (t[v\lhd x])(u[v\lhd x]) & x \in \mathsf{fv}(u) \wedge x \in \mathsf{fv}(u)\end{array}$$

Factoring out the linear and the non-linear occurrences of the variables can be done at the syntactical level, which eliminates the last two rules. If $x \notin \mathsf{fv}(t)$, then we can make this explicit using the erase construct: $E_x(t)$. The associated reduction rule must preserve this notation, by erasing the substitution, and creating explicit erasing constructs for each of the free variables of the substitution:

$$E_x(t)[v\lhd x] \rightsquigarrow E_{\vec{x}}(t)$$

where $\vec{x} = \mathsf{fv}(v)$, and the notation $E_{\vec{x}}(t)$ can be thought of as an abbreviation of $E_{x_1}(\lll E_{x_n}(t) \lll)$. Remark that if $\mathsf{fv}(v) = \emptyset$ then the rule is simply: $E_x(t)[v\lhd x] \rightsquigarrow t$.

If x occurs twice in a term u, then renaming one occurrence to y and the other to z and using the $C_x^{y\frown z}(u)$ construct makes the copying explicit, as given by the following associated rule:

$$C_x^{y\frown z}(t)[v\lhd x] \rightsquigarrow C_{\vec{x}}^{\vec{y}\frown\vec{z}}((t[v[\vec{y}/\vec{x}]\lhd y])[v[\vec{z}/\vec{x}]\lhd z])$$

where $\vec{x} = \mathsf{fv}(v)$, and renamings of the free variables of each copy of v are introduced to preserve the notation. Remark that if $\mathsf{fv}(v) = \emptyset$ then the rule is simply: $C_x^{y\frown z}(t)[v\lhd x] \rightsquigarrow (t[v\lhd y])[v\lhd z]$. Copying terms with free variables can cause the duplication of redexes that might be created later during reduction. An obvious solution to this would simply be to only copy *closed* terms (i.e. without free variables), which avoids the renaming substitutions in the rule. Note that we no longer need the rules *Var2* and *Lam1* with these constructs.

Similarly to the *App* rule, the *Comp* rule now splits into:

$$\begin{array}{ll}(t[u\lhd x])[v\lhd y] \rightsquigarrow t[u[v\lhd y]\lhd x] & y \in \mathsf{fv}(u)\\ (t[u\lhd x])[v\lhd y] \rightsquigarrow (t[v\lhd y])[u\lhd x] & y \in \mathsf{fv}(t)\end{array}$$

The first rule is useful, since it allows the substitution process to move towards completion. However, the second rule is a clear candidate for non-termination and is not essential in the calculus. We drop it from the system, thus eliminating the usual termination problems of explicit substitution calculi, which we refer the reader to [9] for examples.

There is an important issue with respect to the order of the application of the revised *Comp* rule. Consider the term: $t_0[t_1\lhd x_0][t_2\lhd x_1] \lll [t_n\lhd x_{n-1}]$ with $\mathsf{fv}(t_i) = \{x_i\} \wedge i < n$, and $\mathsf{fv}(t_n) = \emptyset$. There is a choice between innermost and outermost application of this rule. Working innermost requires $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ applications of the rule, which is $O(n^2)$, whereas the outermost sequence requires exactly n steps. A sufficient condition to get an outermost strategy is to require the substitution to be *closed* in the *Comp* rule.

Our next refinement is motivated by pushing substitutions through an abstraction efficiently. This operation is quite a complex task, as indicated by the *Lam3* rule: variables must be renamed to avoid variable capture, and thus the operation requires an additional substitution. Avoiding substitution through an abstraction is one solution to this problem, but may also cause duplication of work later. To obtain more sharing, substitution (and reduction) under an abstraction should be allowed. To overcome this clash of interests, one can remark that the rule *Lam3* can be eliminated if the substitution v is *closed*. In this way there is no risk of variable clash or variable capture: $(\lambda x.t)[v/y] \rightsquigarrow \lambda x.t[v/y]$, if $\mathsf{fv}(v) = \emptyset$. Note that the *Comp* rule is essential to group substitutions together so that a single closed substitution can be pushed through an abstraction. Not only does this give the shortest reduction path, but at the same time eliminates the problems of variable capture and clash.

Our last rule of study is the *Beta* rule. Consider the term $((\lambda y.t)u)[v/x]$, where $x \in \mathsf{fv}(t)$, and $\mathsf{fv}(v) = \emptyset$. If we first perform the *Beta* rule we obtain $(t[u/y])[v/x]$, however if we perform the *Comp* rule first we obtain $(t[v/x])[u/y]$. Since we have eliminated one of the cases for the *Comp* rule there is a potential confluence problem. We can avoid it by only allowing β-reduction to take place when the function is *closed*: $(\lambda x.t)u \rightsquigarrow t[u/x]$, if $\mathsf{fv}(\lambda x.t) = \emptyset$. This resolves the above problem by cutting out the possibility of performing the *Beta* rule first. In other work on explicit substitutions it is interesting to note that it is the application of the *Comp* rule first that is ruled out, see for instance [10].

This completes our refinements, we can now clean up the rules that we have discussed up until now, obtaining what we call closed reduction.

3 Closed Reduction

Putting the previous ideas together suggests a strategy for implementing the β-calculus. The strategy is clearly a weak one since it will not always be the case that substitutions will be closed, but we show that it is adequate for the evaluation of programs (i.e. closed terms of base type). The following is the definition of the β_c-calculus which we use for the rest of this paper.

Definition 2 (β_c-terms). *The following table summarizes the terms, variable constraints and the free variables of the terms.*

Name	*Term*	*Variable Constraint*	*Free Variables*
Variable	x	$\otimes$	$\P x\diamondsuit$
Abstraction	$\lambda x.t$	$x \in \mathsf{fv}(t)$	$\mathsf{fv}(t) \otimes \P x\diamondsuit$
Application	(tu)	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \emptyset$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$
Erase	$E_x(t)$	$x \in \mathsf{fv}(t)$	$\mathsf{fv}(t) \setminus \P x\diamondsuit$
Copy	$C_x^{y,z}(t)$	$x \in \mathsf{fv}(t), y \parallel z, \P y, z\diamondsuit \cap \mathsf{fv}(t)$	$\mathsf{fv}(t) \otimes \P y, z\diamondsuit \cup \P x\diamondsuit$
Substitution	$t[u/x]$	$x \in \mathsf{fv}(t), \mathsf{fv}(t) \otimes \P x\diamondsuit \cap \mathsf{fv}(u) = \emptyset$	$\mathsf{fv}(t) \otimes \P x\diamondsuit \cup \mathsf{fv}(u)$

Definition 3 (Closed Reduction). *Let t, u be β_c-terms, $t \rightsquigarrow_w u$ is called a closed reduction and given by the following conditional rewrite system (variables*

of λ_c-terms are treated as constants in the rewrite system, thus $x \not\equiv x'$, etc.).

Name	*Reduction*		*Condition*
Beta	$(\lambda x.t)u$	$\leadsto_w t[u/x]$	$\mathsf{fv}(\lambda x.t) = \emptyset$
Var	$x[v/x]$	$\leadsto_w v$	$\otimes$
App1	$(tu)[v/x]$	$\leadsto_w (t[v/x])u$	$x \notin \mathsf{fv}(t)$
App2	$(tu)[v/x]$	$\leadsto_w t(u[v/x])$	$x \notin \mathsf{fv}(u)$
Lam	$(\lambda y.t)[v/x]$	$\leadsto_w \lambda y.t[v/x]$	$\mathsf{fv}(v) = \emptyset$
Copy1	$C_x^{y,z}(t)[v/x]$	$\leadsto_w (t[v/y])[v/z]$	$\mathsf{fv}(v) = \emptyset$
Copy2	$C_{x'}^{y,z}(t)[v/x]$	$\leadsto_w C_{x'}^{y,z}(t[v/x])$	$\mathsf{fv}(v) = \emptyset$
Erase1	$E_x(t)[v/x]$	$\leadsto_w t$	$\mathsf{fv}(v) = \emptyset$
Erase2	$E_{x'}(t)[v/x]$	$\leadsto_w E_{x'}(t[v/x])$	$\otimes$
Comp	$(t[w/y])[v/x]$	$\leadsto_w t[w[v/x]/y]$	$x \notin \mathsf{fv}(w)$

As usual, we write $\leadsto_w^*$ as the transitive reflexive closure of $\leadsto_w$. The subscript w is used to indicate that it is a weak calculus. These reduction steps can be applied in every context, in particular within substitutions and under abstractions.

The conditions on the rules are motivated from both an efficiency point of view with respect to minimizing the number of β-reduction and substitution steps, and from a simplification point of view for the substitution calculus.

Remark 1. There are a number of variants of the reduction rules, which do not change the basic results of this paper. For instance, in a simply typed framework, the *Copy1* rule can reduce the substituted term v to normal form before copying, thus avoiding redex duplication. The *App1*, *App2* and *Comp* rules could also require v to be closed, giving a weaker but more directed strategy (with the shortest reduction paths). We shall make use of this variant in Section 7.

Note that since we only do closed reductions, we do not need to represent substitutions by lists, as it is done in most calculi of explicit substitutions. In this way we avoid the introduction of constructors for lists and the operation of concatenation with associativity.

Definition 4 (Compilation). *Let t be a λ-term with $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$. Then compilation into λ_c is defined as: $[x_1] \triangleright \cdots \triangleright [x_n]t^\circ$ with $(\ )^\circ$ given by: $x^\circ = x$, $(tu)^\circ = t^\circ u^\circ$, and $(\lambda x.t)^\circ = \lambda x \triangleright [x]t^\circ$ if $x \in \mathsf{fv}(t)$, otherwise $(\lambda x.t)^\circ = \lambda x \triangleright E_x(t^\circ)$. We define $[\ ]$ as:*

$$
\begin{array}{lll}
[x]x & = x & \\
[x](\lambda y.t) & = \lambda y \triangleright [x]t & \\
[x](tu) & = C_x^{x', x''}([x'](t^{\{x \mapsto x'\}})[x''](u^{\{x \mapsto x''\}})) & x \in \mathsf{fv}(t), x \in \mathsf{fv}(u) \\
 & = ([x]t)u & x \in \mathsf{fv}(t), x \notin \mathsf{fv}(u) \\
 & = t([x]u) & x \notin \mathsf{fv}(u), x \in \mathsf{fv}(t) \\
[x]E_y(t) & = E_y([x]t) & \\
[x]C_y^{y', y''}(t) & = C_y^{y', y''}([x]t) &
\end{array}
$$

where the substitution $t^{\{x \mapsto u\}}$ is the usual (implicit) notion, and the variables x' and x'' above are assumed to be fresh.

Remark 2. Terms built from the compilation are *pure* terms (they do not contain substitutions). The compilation functions are simply counting variables, placing erasing operations outermost, and copying operations innermost. In particular, remark that the compilation only introduces an erasing construct immediately after an abstraction: $\lambda x.E_x(t)$. If we are only interested in terms arising from the compilation, we can abbreviate this to $\lambda.t$, together with a new reduction rule *BetaErase*: $(\lambda.t)u \rightsquigarrow_w t$, if $\mathsf{fv}(u) = \emptyset$ in place of the rules *Erase1*, *Erase2*.

Proposition 1. *If t is a λ-term with free variables $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$, then:*

1. $\mathsf{fv}([x_1] \lll [x_n]t^\circ) = \mathsf{fv}(t)$.
2. $[x_1] \lll [x_n]t^\circ$ *is a valid λ_c-term (satisfying the variable constraints).*

There are valid λ_c-terms which are not derived from the translation of λ-terms, such as: $\lambda y.(xy)[\lambda x.x]$. Here a *Beta* rule must have been applied when the function was not closed. In this paper we will always assume that we are dealing with λ_c-terms derived from the translation and their reducts, and as shown below, reduction preserves the variable constraints. Some of the constraints of the rewriting system can be eliminated with these assumptions, but we keep them for clarity. We can recover a λ-term from a λ_c-term by simply erasing the additional constructs and completing the substitutions:

Definition 5 (Read-back).

$$\begin{array}{llll} x^* & = x & (E_x(t))^* & = t^* \\ (tu)^* & = t^* u^* & (C_x^{y,z}(t))^* & = t^{*\{y \mapsto x\}\{z \mapsto x\}} \\ (\lambda x.t)^* & = \lambda x.t^* & (t[u/x])^* & = t^{*\{x \mapsto u^\circ\}} \end{array}$$

One can easily show that if $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$ then $([x_1] \lll [x_n]t^\circ)^* = t$.

Example 1. We give several example terms in this calculus:

$$\begin{array}{lll} \mathbf{I} & = (\lambda x.x)^\circ & = \lambda x.x \\ \mathbf{K} & = (\lambda xy.x)^\circ & = \lambda xy.E_y(x) \\ \mathbf{S} & = (\lambda xyz.xz(yz))^\circ & = \lambda xyz.C_z^{z,z}((xz')(yz'')) \\ \mathbf{2} & = (\lambda fx.f(fx))^\circ & = \lambda fx.C_f^{f,f}(f'(f''x)) \\ \mathbf{Y} & = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))^\circ & = \lambda f.C_f^{f,f}(\lambda x.f'(C_x^{x,x}(x'x''))) \\ & & \quad (\lambda x.f''(C_x^{x,x}(x'x''))) \end{array}$$

and a reduction sequence to normal form:

$$\begin{array}{ll} \mathbf{22} = & (\lambda fx.C_f^{f,f}(f'(f''x)))\mathbf{2} \\ \rightsquigarrow_w & (\lambda x.C_f^{f,f}(f'(f''x)))[\mathbf{2}/f] \\ \rightsquigarrow_w^* & \lambda x.\mathbf{2}(\mathbf{2}x) \\ \rightsquigarrow_w & \lambda x.(\lambda x.C_f^{f,f}(f'(f''x)))[\mathbf{2}x/f] \\ \rightsquigarrow_w & \lambda x.(\lambda x.C_f^{f,f}(f'(f''x)))[(\lambda x.C_f^{f,f}(f'(f''x)))[x/f]/f] \end{array}$$

Remark that it is impossible to duplicate the variable x in the last line □ we must wait for a closing substitution before continuing. If we apply this to a closed term, for instance $\mathbf{I}$, then reduction can continue:

$$\begin{aligned}
\mathbf{22I} &\rightsquigarrow_w^* ((\square x \triangleright C_f^{f^{\square} \triangleleft f^{\square\square}}(f'(f''x)))[(\square x \triangleright C_f^{f^{\square} \triangleleft f^{\square\square}}(f'(f''x)))[x \triangleleft f] \triangleleft f])[\mathbf{I} \triangleleft x] \\
&\rightsquigarrow_w^* (\square x \triangleright C_f^{f^{\square} \triangleleft f^{\square\square}}(f'(f''x)))[(\square x \triangleright C_f^{f^{\square} \triangleleft f^{\square\square}}(f'(f''x)))[\mathbf{I} \triangleleft f] \triangleleft f] \\
&\rightsquigarrow_w^* (\square x \triangleright C_f^{f^{\square} \triangleleft f^{\square\square}}(f'(f''x)))[(\square x \triangleright \mathbf{I}(\mathbf{I}x)) \triangleleft f] \\
&\rightsquigarrow_w^* (\square x \triangleright C_f^{f^{\square} \triangleleft f^{\square\square}}(f'(f''x)))[\mathbf{I} \triangleleft f] \\
&\rightsquigarrow_w^* (\square x \triangleright \mathbf{I}(\mathbf{I}x)) \\
&\rightsquigarrow_w^* \square x \triangleright x
\end{aligned}$$

If one studies in detail this example, there are several strategies of evaluation possible. The best (shortest) route to take is always to push closed substitutions through an application, and reduce terms to normal form before copying.

In the □-calculus, $\mathbf{Y}$ is a term which has a weak head normal form, but no head normal form. However, even though we reduce under abstractions there is no in□nite sequence of reductions in $\square_c$ since the only redex (shown underlined) has a free variable f':

$$\square f \triangleright C_f^{f^{\square} \triangleleft f^{\square\square}}(\underline{(\square x \triangleright f'(C_x^{x^{\square} \triangleleft x^{\square\square}}(x'x')))(\square x \triangleright f''(C_x^{x^{\square} \triangleleft x^{\square\square}}(x'x')))})$$

However, the term $\mathbf{YI}$ does generate a non-terminating sequence.

Note that we never need any □-conversions during the reductions, although the same variable can appear several times in the term being reduced.

4 Properties

In this section we show some basic properties of the rewrite system $\rightsquigarrow_w$ which we use to reduce $\square_c$-terms, i.e. ground terms with respect to the rewrite system.

Proposition 2 (Correctness of $\rightsquigarrow_w$).

1. *If $t \rightsquigarrow_w u$ then $\mathsf{fv}(t) = \mathsf{fv}(u)$.*
2. *If t is a $\square_c$-term and $t \rightsquigarrow_w u$, then u is a $\square_c$-term (i.e. the relation $\rightsquigarrow_w$ preserves the variable constraints).*

Proposition 3 (Local Con□uence). *If $t \rightsquigarrow_w u$ and $t \rightsquigarrow_w v$ then there is a term s such that $u \rightsquigarrow_w^* s$ and $v \rightsquigarrow_w^* s$.*

Proof. There are □ve critical pairs to consider, since all the other potential superpositions are eliminated from the system because of the free variable constraints. In all the cases the critical pair converges. □

To prove the termination of the rules for substitution we de□ne the distance ♣ from the root of the term t to the unique occurrence of the free variable x as: ♣ = 1, ♣ = 1 + ♣, ♣ = 1 + ♣ (if $x \in \mathsf{fv}(t)$), ♣ = 1 + ♣ (if $x \in \mathsf{fv}(u)$), ♣ = 1, ♣ = 1 + ♣, ♣ = 1 + ♣ + ♣, ♣ = 1 + ♣, and ♣ = 1 + ♣ (if $x \in \mathsf{fv}(v)$) otherwise ♣ = 1 + ♣ (if $x \in \mathsf{fv}(t)$).

Proposition 4 (Termination of substitutions). *There are no infinite reduction sequences starting from $t[v \lhd x]$ using only the rules for substitution.*

Proof. We define an interpretation that associates to each term a multiset with one element for each sub-term $w[s \lhd x]$ occurring in t, which is $\clubsuit w \clubsuit$. Each application of a substitution rule decreases the interpretation of the term, since we always apply a substitution to a sub-term of t or erase it, and the distance is reduced. □

We now look at the problem of preservation of strong normalization (a calculus of explicit substitutions preserves strong normalization if the compilation of a strongly normalizable λ-term is strongly normalizable).

Our proof is inspired from that of $\lambda\lambda$ [7]. We first define a notion of minimal infinite derivation. Intuitively, a derivation is minimal if we always reduce the lowest possible redex to keep non termination. We denote by $\to_E^*$ a sequence of reductions in $\to_c$ that does not use the *Beta* rule, and by $\to_{Beta\,p}$ a *Beta* reduction at position p.

Definition 6 (Minimal derivation). *An infinite $\to_c$ derivation*

$$a_1 \to_{Beta\,p_1} a'_1 \to_E^* \lll a_i \to_{Beta\,p_i} a'_i \to_E^* \rhd\rhd$$

is minimal *if for any other infinite derivation*

$$a_1 \to_{Beta\,p_1} a'_1 \to_E^* \lll a_i \to_{Beta\,q} b \to_E^* \lll$$

we have $q \not\geq p_i p'$ for every p'.

In other words, in any other infinite derivation, p_i and q are disjoint or q is above p_i, which means that the *Beta*-redex we reduce is a lowest one.

Lemma 1. *1. If $u \to_E^* u'$ then $u^* = u'^*$.*
2. If $u \to_{Beta} u'$ is a step in a minimal derivation starting from the translation of a λ-term then $u^* \to_\lambda^+ u'^*$.

Proof. 1. Straightforward inspection of the rules for substitution.
2. The term u contains a *Beta* redex, and the minimality assumption for the derivation ensures that in the translation to u^* this redex is not erased (but it can of course be copied). Hence $u^* \to_\lambda^+ u'^*$. □

Proposition 5 (Preservation of Strong Normalization). *If t is the translation of a strongly normalizable λ-term then t is strongly normalizable.*

Proof. Assuming that there is an infinite reduction sequence starting from $t = [x_1]\lll[x_n]s^\circ$ in $\to_c$, we will show that there is an infinite derivation for the strongly normalizable λ-term s (contradiction). For this we consider an infinite minimal reduction sequence out of t. Since the rules for substitution are terminating, it contains an infinite number of applications of *Beta*.

$$t \to_E^* t_1 \to_{Beta} t_2 \to_E^* t_3 \to_{Beta} t_4 \lll$$

Since $t^* = s$, we obtain an infinite derivation for s, by Lemma 1 (contradiction):

$$s = t^* = t_1^* \to_\lambda^+ t_2^* = t_3^* \to_\lambda^+ t_4^* \lll \qquad \square$$

5 A Type System for $\Box_c$

There is also a typed version of the calculus, which is given by the following rules. We remark that many of the syntactical constraints are now captured by the type rules.

$$\frac{\Box \lhd x : A \lhd y : B \lhd \Box \ \Box \ t : C}{\Box \lhd y : B \lhd x : A \lhd \Box \ \Box \ t : C}\ (X)$$

$$\frac{}{x : A \ \Box \ x : A}\ (\text{VAR}) \qquad \frac{\Box \lhd x : A \ \Box \ t : B \qquad \Box \ \Box \ v : A}{\Box \lhd \Box \ \Box \ t[v \lhd x] : B}\ (\text{SUB})$$

$$\frac{\Box \lhd x : A \ \Box \ t : B}{\Box \ \Box \ \Box x \rhd t : A \ \Box \ B}\ (\text{ABS}) \qquad \frac{\Box \ \Box \ t : A \ \Box \ B \qquad \Box \ \Box \ u : A}{\Box \lhd \Box \ \Box \ tu : B}\ (\text{APP})$$

$$\frac{\Box \ \Box \ t : B}{\Box \lhd x : A \ \Box \ E_x(t) : B}\ (\text{WEAK}) \qquad \frac{\Box \lhd x : A \lhd y : A \ \Box \ t : B}{\Box \lhd z : A \ \Box \ C_z^{x \lhd y}(t) : B}\ (\text{CONT})$$

Lemma 2 (Subject Reduction). *If* $\Box \ \Box \ t : A$ *and* $t \rightsquigarrow_w u$ *then* $\Box \ \Box \ u : A$.

We extend $\Box_c$ (and the $\Box$-calculus) to include a generic constant $\Box : I$, with the following axiom: $\Box \ \Box : I$, where the type I represents a generic base type. This extension gives us a very minimalistic functional programming language. There are no additional reduction rules for this constant, and the above subject reduction theorem can easily be seen to hold with the addition of this axiom. Obviously, $\mathsf{fv}(\Box) = \Box$.

Lemma 3. *Let* t *be a* $\Box$*-term and* s *its compilation in* $\Box_c$. t *is simply typable if and only if* s *is typable.*

We now look at the problem of termination of typable terms, keeping with the philosophy of only considering terms that are derived from $\Box$-terms. This is an assumption in all the following results.

Since simply typable $\Box$-terms are strongly normalizable, as a direct consequence of Lemma 3 and Proposition 5 we obtain:

Proposition 6 (Termination). *If* $\Box \ \Box \ t : A$ *in* $\Box_c$ *then* t *is strongly normalizable.*

Proposition 7 (Con□uence). *If* $\Box \ \Box \ t : A$, $t \rightsquigarrow_w^* u$ *and* $t \rightsquigarrow_w^* v$ *then there is a term* s *such that* $u \rightsquigarrow_w^* s$ *and* $v \rightsquigarrow_w^* s$.

De□nition 7 (Programs). *A* program *is a closed term of type* I.

We can show that in our calculus programs can be reduced to values which are pure terms. This can be understood as for all closed terms of type I, there are enough closed substitutions so that they can all complete. For that we need a lemma.

Lemma 4 (Completion of closed substitutions). *If v is a closed term, then $t[v \triangleleft x]$ is not a normal form.*

Proof. The compilation function gives a term that satisfies the variable constraints, and reduction preserves the constraints (Proposition 2). Therefore x occurs in t. If t is a variable, application, abstraction, erasing or copying, then we can apply one of the rules for substitution. If $t = u[w \triangleleft y]$ then x occurs free in w (it cannot occur free in u since the *Beta* rule that created the substitution could not be applied with an open function), therefore we can apply the *Comp* rule. □

Theorem 1 (Adequacy). *If t is a program, then $t \rightsquigarrow_w^* \square$.*

Proof. By Subject Reduction, the type of the term is preserved under reduction. Assume for a contradiction that the program t is in normal form, and it is not □. Since t is closed, it cannot be a variable, an erasing construct or a copying construct. Since it is of type I, it cannot be an abstraction either. If $t = u[v \triangleleft x]$ then v is closed (since t is closed), and therefore one of the rules for substitution would apply by Lemma 4. Hence t is an application.

Let $t = u_1 u_2 \rhd\rhd\rhd u_n$, $n \geq 2$, such that u_1 is not an application. Since t is closed, so are $u_1 \lhd\lhd\lhd u_n$. Hence u_1 is not a variable, a copying or an erasing. Since t is a normal form, u_1 cannot be an abstraction either (the *Beta* rule would apply). Therefore u_1 is a term of the form $s[s' \triangleleft x]$ where s' is closed and x is the only free variable of s. But then u_1 is not a normal form (Lemma 4), which contradicts our assumption. □

Note that this result also holds even if we take a weaker calculus with closed substitutions for the application and variable rules. The type system and the notion of a program can be regarded as a simplification of the usual notion from the language PCF [11], where we have just one constant, and no arithmetic functions or recursion. There are no difficulties in extending this calculus to the full language of PCF.

6 Relation with □-reduction

In this section we compare closed reduction with several common evaluation strategies for the □-calculus. In most functional programming languages closed terms are reduced to weak head normal form (WHNF). There are three main strategies of reduction to WHNF: Call-by-name □ leftmost outermost (normal order); Call-by-value □ leftmost innermost (applicative order); and Call-by-need □ call-by-name + sharing.

We will show that we can simulate the three strategies in $\square_c$. Let t be a closed □-term and u its WHNF obtained using one of these strategies. We will show that $t^\circ \rightsquigarrow_w^* u^\circ$, by induction on the length of the derivation $t \square^* u$. If t is already a WHNF the result is trivial. We assume there is a non-empty reduction to WHNF. The following lemma will be useful to establish the connection with □-reduction in the □-calculus.

Lemma 5. *Let P and Q be λ-terms such that* $\mathsf{fv}(P) = \{x_1, \ldots, x_n\}$, $n \geq 1$, *and* $\mathsf{fv}(Q) = \emptyset$, *then* $([x_1] \triangleright \cdots \triangleright [x_n]P^\circ)[Q^\circ \triangleleft x_1] \rightsquigarrow_w^* [x_2] \triangleright \cdots \triangleright [x_n](P^{\{x_1 \mapsto Q\}})^\circ$.

Proof. By Proposition 1 (Part 1), $\mathsf{fv}(Q^\circ) = \emptyset$, and $x_1 \notin \mathsf{fv}([x_1] \triangleright \cdots \triangleright [x_n]P^\circ)$. The result is obtained by induction on P. We show just one case when $P \equiv \lambda y.t$, $y \in \mathsf{fv}(t)$. Assume w.l.o.g. that x is the only free variable of P.

$$\begin{array}{rcl}
([x](\lambda y.t)^\circ)[Q^\circ \triangleleft x] & = & ([x](\lambda y.[y]t^\circ))[Q^\circ \triangleleft x] \\
& = & (\lambda y.[x][y]t^\circ)[Q^\circ \triangleleft x] \\
& \rightsquigarrow_w & \lambda y.([x][y]t^\circ)[Q^\circ \triangleleft x] \\
& \rightsquigarrow_w (IH) & \lambda y.[y](t^{\{x \mapsto Q\}})^\circ \\
& = & (\lambda y.t^{\{x \mapsto Q\}})^\circ \qquad \square
\end{array}$$

Call-by-name Let $t \to u$ be the first step in a call-by-name reduction to WHNF in the λ-calculus. Then let $t = (\lambda x.P)Q$, with $\mathsf{fv}(Q) = \emptyset$ and $u = P^{\{x \mapsto Q\}}$. There are two cases to consider. If $x \notin \mathsf{fv}(P)$ then

$$t^\circ = (\lambda x.P)^\circ Q^\circ = (\lambda x.[x]P^\circ)Q^\circ \rightsquigarrow_w ([x]P^\circ)[Q^\circ \triangleleft x] \rightsquigarrow_w^* (P^{\{x \mapsto Q\}})^\circ$$

using Lemma 5. Otherwise

$$t^\circ = (\lambda x.P)^\circ Q^\circ = (\lambda x.E_x(P^\circ))Q^\circ \rightsquigarrow_w (E_x(P^\circ))[Q^\circ \triangleleft x] \rightsquigarrow_w P^\circ = (P^{\{x \mapsto Q\}})^\circ$$

Call-by-value Let $t = (\lambda x.P)Q$ be a closed λ-term and u its WHNF using call-by-value, then there is a sequence of reductions:

$$(\lambda x.P)Q \to^* (\lambda x.P)V \to P^{\{x \mapsto V\}} \to^* u$$

where V is a value (WHNF for closed terms). Using induction and Lemma 5:

$$t^\circ = (\lambda x.P)^\circ Q^\circ \rightsquigarrow_w^* (\lambda x.P)^\circ V^\circ \rightsquigarrow_w^* (P^{\{x \mapsto V\}})^\circ \rightsquigarrow_w^* u^\circ$$

Call-by-need Let t be a closed term and u its WHNF using call-by-need. To simulate call-by-need it is enough to reduce a substitution to WHNF before applying it when it is needed (i.e. not in $E_x(t)[u \triangleleft x]$). We can simulate the reductions in the call-by-need λ-calculus of Ariola and Felleisen [2], by using the explicit substitutions construction instead of `let`, and using the erasing rule to erase the substitutions that are not needed. Consider the following example, which is taken from [2] and adapted to our notation:

$$\begin{array}{rcl}
(\lambda f \triangleright C_f^{f' \triangleleft f''}(f'\mathbf{I}(f''\mathbf{I})))((\lambda z w \triangleright z w)(\mathbf{I}\mathbf{I})) & \rightsquigarrow_w & C_f^{f' \triangleleft f''}(f'\mathbf{I}(f''\mathbf{I}))[(\lambda z w \triangleright z w)(\mathbf{I}\mathbf{I}) \triangleleft f] \\
& \rightsquigarrow_w & C_f^{f' \triangleleft f''}(f'\mathbf{I}(f''\mathbf{I}))[(\lambda w \triangleright z w)[\mathbf{I}\mathbf{I} \triangleleft z] \triangleleft f] \\
& \rightsquigarrow_w^* & C_f^{f' \triangleleft f''}(f'\mathbf{I}(f''\mathbf{I}))[(\lambda w \triangleright \mathbf{I} w) \triangleleft f] \\
& \rightsquigarrow_w^* & C_f^{f' \triangleleft f''}(f'\mathbf{I}(f''\mathbf{I}))[(\lambda w \triangleright w) \triangleleft f] \\
& \rightsquigarrow_w^* & (\lambda w \triangleright w)\mathbf{I}((\lambda w \triangleright w)\mathbf{I}) \\
& \rightsquigarrow_w^* & \mathbf{I}
\end{array}$$

Remark 3. Note that for reduction of closed terms to WHNF we do not need composition of substitutions (all arguments will be closed), however, having this rule in the system allows additional reductions under abstractions. Although we can reduce open terms (e.g. $(\lambda x.x)y \rightsquigarrow_w^* y$), we cannot in general simulate reduction to WHNF on open terms, for instance: $(\lambda x.xy)(\lambda x.x) \not\rightsquigarrow_w$ is not a WHNF. We cannot in general simulate reduction to head normal form (HNF), even for closed terms, for example: $\lambda x.(\lambda y.E_y(x))t \not\rightsquigarrow_w$. However the calculus is stronger than reduction to WHNF, since we can reduce under abstractions:

$$\lambda x.(\lambda y.y)x \rightsquigarrow_w \lambda x.y[x/y] \rightsquigarrow_w \lambda x.x$$

To obtain HNF in general, we can define a different version of the calculus, which requires that the substitution is closed when pushed into an argument of a function, rather than the abstraction, thus a dual version of this calculus which we leave for future study.

7 Alternative Presentation: Director Strings

In this section it will be shown how to present the calculus by using the concept of directors strings [6], which can be seen as a way to internalize some of the conditions on the rewriting system. The calculus that we are left with is a generalization of the director strings system of [6] (which corresponds to combinator reduction) where we allow reduction under an abstraction.

To make the presentation more concise, we adopt the version of the calculus where the rules *App1*, *App2* and *Comp* also require that the substitution is closed. We shall also adopt the alternative syntax of $\lambda.t$ rather than $\lambda x.E_x(t)$ for this section, which is equivalent, but more compact, as discussed in Section 3.

The elements $\otimes$, $\swarrow$, $\curvearrowright$, $\curvearrowleft$, called *directors*, will be used to annotate binary constructors to indicate that a substitution $[t/x]$ is not required, used in both, or just the left or right argument respectively. Unary constructors will have the annotation $\downarrow$, which just means that the substitution is required in the sub-term. $\Box$ will denote the empty string, which annotates *closed* terms. We will often drop the outermost $\Box$ string, except if it is needed to express a rule.

Working with annotated terms, we no longer need the erasing and copying constructs, since they are captured by the directors. Moreover, variables will always be labeled as $x^{\downarrow}$, and therefore the name of the variable is not important, since only the correct substitution will reach a variable. Thus we will abbreviate $x^{\downarrow}$ as $\downarrow$. Now substitutions and abstraction will no longer need to hold the name of the variable, so we will write $\lambda x.t$ as λt, $\lambda.t$ as $\lambda^{-}t$ and $[t/x]$ simply as $[t]$.

In the following we use the notation $\clubsuit^{\clubsuit}$ for the length of the string s, and if d is a director, then d^n is a string of d's of length n. Let t be a λ-term with $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$; its compilation is now defined as: $[x_1] \triangleright\triangleright \ldots \triangleright\triangleright [x_n]t^{\circ}$ with $(\lesssim^{\circ}$ given by: $x^{\circ} = x$, $(tu)^{\circ} = t^{\circ}u^{\circ}$, and $(\lambda x.t)^{\circ} = \lambda[x]t^{\circ}$ if $x \in \mathsf{fv}(t)$, otherwise

$(\lambda x.t)^\circ = \lambda^- t^\circ$. We define $[x]$ as:

$$\begin{array}{lll} [x]x & = \downarrow & \\ [x](\lambda t)^s & = (\lambda [x]t)^{\downarrow s} & \\ [x](tu)^s & = ([x]t)([x]u)^{\curlywedge s} & x \in \mathsf{fv}(t) \setminus x \in \mathsf{fv}(u) \\ & = (([x]t)u)^{\curvearrowleft s} & x \in \mathsf{fv}(t) \setminus x \notin \mathsf{fv}(u) \\ & = (t([x]u))^{\curvearrowright s} & x \notin \mathsf{fv}(u) \setminus x \in \mathsf{fv}(t) \end{array}$$

For example, $\mathbf{S} = (\lambda xyz.(xz)(yz))^\circ = \lambda(\lambda(\lambda((\downarrow\downarrow)^{\curvearrowright\curvearrowright}(\downarrow\downarrow)^{\curvearrowright\curvearrowright})^{\curvearrowright\curvearrowright\curlywedge})^{\downarrow\downarrow})^{\downarrow}$, $\mathbf{K} = (\lambda xy.x)^\circ = (\lambda(\lambda^- \downarrow)^{\downarrow})$, $\mathbf{I} = (\lambda x.x)^\circ = (\lambda \downarrow)$.

The reduction rules for this calculus are now given as follows:

Name	*Reduction*		
Beta	$((\lambda t)^\lambda u)^s$	$\rightsquigarrow_w$	$(t[u])^s$
BetaErase	$((\lambda^- t)^\lambda u^\lambda)^\lambda$	$\rightsquigarrow_w$	t
Var	$(\downarrow [v^{s_1}])^s$	$\rightsquigarrow_w$	v^{s_1}
App1	$((t^{s_2}u)^{\curvearrowleft s_1}[v^\lambda])^s$	$\rightsquigarrow_w$	$((t^{s_2}[v^\lambda])^{\curvearrowleft_2\clubsuit}u)^{s_1}$
App2	$((tu^{s_2})^{\curvearrowright s_1}[v^\lambda])^s$	$\rightsquigarrow_w$	$(t(u^{s_2}[v^\lambda])^{\curvearrowleft_2\clubsuit})^{s_1}$
App3	$((t^{s_2}u^{s_3})^{\curlywedge s_1}[v^\lambda])^s$	$\rightsquigarrow_w$	$((t^{s_2}[v^\lambda])^{\curvearrowleft_2\clubsuit}(u^{s_3}[v^\lambda])^{\curvearrowleft_3\clubsuit})^{s_1}$
Lam	$((\lambda t)^{\downarrow s_1}[v^\lambda])^s$	$\rightsquigarrow_w$	$(\lambda(t[v^\lambda])^{\curvearrowleft_1\clubsuit})^{s_1}$
Comp	$((t[w^{s_2}])^{\curvearrowright s_1}[v^\lambda])^s$	$\rightsquigarrow_w$	$(t[(w^{s_2}[v^\lambda])^{\curvearrowleft_2\clubsuit}])^{s_1}$

We complete this excursion with a simple example reduction sequence to show that we can reduce under abstractions.

$$(\lambda x.(\lambda y.y)x)^\circ = (\lambda((\lambda \downarrow)^\lambda \downarrow)^{\curvearrowright})^\lambda \rightsquigarrow_w \lambda(\downarrow [\downarrow])^{\curvearrowright} \rightsquigarrow_w \lambda \downarrow = (\lambda x.x)^\circ$$

The idea is that the director strings describe a path from the root of the term to the occurrence of a variable. Our calculus allows reductions in the term such that the path is preserved under these reductions, which makes it a more general reduction system than director strings. It remains to be seen if these ideas can in fact be extended to the whole of the λ-calculus, thus generalizing further than closed reductions.

8 Conclusions

In this paper we have proposed a strategy for reduction in the λ-calculus with explicit substitutions. The essential principle is that the reduction process should be simple, and capture the shortest reduction paths.

An implementation of a slight variant of this calculus exists as a system of interaction nets [8], and surprisingly is more efficient in terms of reduction steps than interaction net implementations of optimal reduction. However, it remains to compare in more detail the relationship between optimal reduction and closed reduction.

Acknowledgements

We would like to thank Jean Goubault-Larrecq, Delia Kesner and the anonymous referees for comments on an early version of this paper.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 233–246. ACM Press, 1995.

[3] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.

[4] J.-Y. Girard. Geometry of interaction 1: Interpretation of System F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, eds, *Logic Colloquium 88*, Studies in Logic and the Foundations of Mathematics. North Holland Publishing Company, 1989.

[5] T. Hardin, L. Maranget, and B. Pagano. Functional back-ends within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.

[6] J. Kennaway and M. Sleep. Director strings as combinators. *ACM Transactions on Programming Languages and Systems*, 10:602–626, 1988.

[7] P. Lescanne and J. Rouyer-Degli. The calculus of explicit substitutions lambda-upsilon. Technical Report RR-2222, INRIA, 1995.

[8] I. Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, 1998.

[9] P.-A. Melliès. Typed lambda-calculi with explicit substitutions may not terminate. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, number 902 in Lecture Notes in Computer Science, pages 328–334. Springer-Verlag, 1995.

[10] C. Muñoz. Confluence and preservation of strong normalisation in an explicit substitutions calculus (extended abstract). In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, New Brunswick, New Jersey. IEEE Computer Society Press, 1996.

[11] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[12] K. H. Rose. Explicit substitution - tutorial and survey. Lecture Series LS-96-3, BRICS, Dept. of Computer Science, University of Aarhus, Denmark, 1996.

[13] N. Yoshida. Optimal reduction in weak lambda-calculus with shared environments. *Journal of Computer Software*, 11(6):3–18, 1994.

Kripke Resource Models of a Dependently-Typed, Bunched λ-Calculus (Extended Abstract)

Samin Ishtiaq and David J. Pym

Department of Computer Science
Queen Mary & Westfield College
University of London
London E1 4NS
{si,pym}@dcs.qmw.ac.uk

Abstract. The $\lambda\Lambda$-calculus is a dependent type theory with both linear and intuitionistic dependent function spaces. It can be seen to arise in two ways. Firstly, in logical frameworks, where it is the language of the RLF logical framework and can uniformly represent linear and other relevant logics. Secondly, it is a presentation of the proof-objects of **BI**, the logic of bunched implications. **BI** is a logic which directly combines linear and intuitionistic implication and, in its predicate version, has both linear and intuitionistic quantifiers. The $\lambda\Lambda$-calculus is the dependent type theory which generalizes both implications and quantifiers. In this paper, we describe the categorical semantics of the $\lambda\Lambda$-calculus. This is given by Kripke resource models, which are monoid-indexed sets of functorial Kripke models, the monoid giving an account of resource consumption. We describe a class of concrete, set-theoretic models. The models are given by the category of families of sets, parametrized over a small monoidal category, in which the intuitionistic dependent function space is described in the established way, but the linear dependent function space is described using Day's tensor product.

1 Introduction

A long-standing problem has been to combine type-dependency and linearity. In [13], we introduced the $\lambda\Lambda$-calculus, a first-order dependent type theory with a full linear dependent function space, as well as the usual intuitionistic dependent function space. The $\lambda\Lambda$-calculus can be seen to arise in two ways. Firstly, in *logical frameworks* [9,18], in which it provides a language that is a suitable basis for a framework capable of properly representing linear and other relevant logics. Secondly, from the *logic of bunched implications*, **BI** [15,19], in which the antecedents of sequents are structured not as lists but as *bunches*, which have two combining operations, ";", which admits Weakening and Contraction, and ",", which does not. The $\lambda\Lambda$-calculus stands in propositions-as-types correspondence with a fragment of **BI** [13,12].

The purpose of this paper is to present the categorical semantics of the $\lambda\Lambda$-calculus. This is given by *Kripke resource models*, which are monoid-indexed sets of functorial Kripke models. The indexing element can be seen as the resource able to realize the structure it indexes. We work with indexed categories rather than, for example, with Cartmell's contextual categories [4], as the indexed approach allows a better separation of the evident conceptual issues.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 235–249, 1999.

Kripke resource models generalize, as we might expect, the functorial Kripke models of the $\lambda\Pi$-calculus [18]. These consist of a functor $\mathcal{J}{:}[\mathcal{W}, [\mathcal{C}^{op}, \mathbf{Cat}]]$, where $\mathcal{W}$ is a Kripke world structure, $\mathcal{C}$ is a category with a $(\times, 1)$ cartesian monoidal structure on it and $[\mathcal{C}^{op}, \mathbf{Cat}]$ is a strict indexed category. The intuitionistic dependent function space Π is modelled as right adjoint to the weakening functor $p^*{:}\mathcal{J}(W)(D) \to \mathcal{J}(W)(D \times A)$.

In the $\lambda\Lambda$-calculus, we have two kinds of context extension operators, so we require $\mathcal{C}$ to have two kinds of monoidal structure on it, $(\otimes, I)$ and $(\times, 1)$. The intuitionistic dependent function space Π can be modelled, as usual, using the right adjoint to projection. However, there is no similar projection functor corresponding to Λ. For this, we must require the existence of the natural isomorphism $Hom_{\mathcal{J}_{r+r'}(W)(D\otimes A)}(1, B) \cong Hom_{\mathcal{J}_r(W)(D)}(1, \Lambda x{:}A.B)$, where $D \otimes A$ is defined in the $r + r'$-indexed model. This is sufficient to define the function space.

While the $\lambda\Lambda$-calculus has familiar soundness and, via a term model, completeness theorems, it is important to ask if there is a natural class of models. For the $\lambda\Pi$-calculus, for instance, the most intuitive concrete model is that of families of sets, **Fam**. This can be viewed as an indexed category **Fam**$:[Ctx^{op}, \mathbf{Cat}]$. The base, Ctx, is a small set-theoretic category whose objects are sets and morphisms are set-theoretic functions. For each $D \in obj(\mathbf{Fam})$, $\mathbf{Fam}(D) = \{y \in B(x) \mid x \in D\}$. The fibre is just a discrete category whose objects are the elements of $B(x)$. If $f \in \mathbf{Fam}(C, D)$, then $\mathbf{Fam}(f)$ just re-indexes the set over D to one over C. As there is little structure required in the fibre, the description of families of sets can also be given sheaf-theoretically, as **Fam**$:[Ctx^{op}, \mathbf{Set}]$, each $\mathbf{Fam}(D)$ being considered as a discrete category. Using Day's construction [7], we obtain a corresponding class of set-theoretic models, parametrized on a small monoidal category, for the $\lambda\Lambda$-calculus. That is, we describe a families of sets model in **BIFam**$:[\mathcal{C}, [Ctx^{op}, \mathbf{Set}]]$, where $\mathcal{C}$ is some small monoidal category.

2 The $\lambda\Lambda$-Calculus

A detailed account of the $\lambda\Lambda$-calculus and the RLF logical framework is given in [13]. The work there develops ideas originally presented in [17].

The $\lambda\Lambda$-calculus is a first-order dependent type theory with both linear and intuitionistic function types. The calculus is used for deriving typing judgements. There are three entities in the $\lambda\Lambda$-calculus: *objects*, *types* and *families of types*, and *kinds*. Objects (denoted by M, N) are classified by types. Families of types (denoted by A, B) may be thought of as functions which map objects to types. Kinds (denoted by K) classify families. In particular, there is a kind Type which classifies the types. We will use U, V to denote any of the entities. The abstract syntax of the $\lambda\Lambda$-calculus is given by the following grammar:

$$
\begin{array}{lll}
K & ::= & \mathsf{Type} \mid \Lambda x{:}A.K \mid \Lambda x!A.K \\
A & ::= & a \mid \Lambda x{:}A.B \mid \Lambda x!A.B \mid \lambda x{:}A.B \mid \lambda x!A.B \mid AM \mid A\&B \\
M & ::= & c \mid x \mid \lambda x{:}A.M \mid \lambda x!A.M \mid MN \mid \langle M, N\rangle \mid \pi_0(M) \mid \pi_1(M)
\end{array}
$$

We write $x{\in}A$ to range over both linear ($x{:}A$) and intuitionistic ($x!A$) variable declarations. The λ and Λ bind the variable x. The object $\lambda x{:}A.M$ is an inhabitant of the linear dependent function type $\Lambda x{:}A.B$. The object $\lambda x!A.M$ is an inhabitant of the

type $\Lambda x!A.B$ (which can also be written as $\Pi\, x{:}A.B$). The notion of linear free- and bound-variables (LFV, LBV) and substitution may be defined accordingly. When x is not free in B we write $A \multimap B$ for $\Lambda x{:}A.B$ and $A \to B$ for $\Lambda x!A.B$).

We can define the notion of *linear occurrence* by extending the general idea of occurrence for the λ-calculus [2], although we note that other definitions may be possible.

Definition 1. *1. x linearly occurs in x;*

2. *If x linearly occurs in U or V (or both), then x linearly occurs in $\lambda y{\in}U.V$, in $\Lambda y{\in}U.V$, and in UV, where $x \neq y$;*
3. *If x linearly occurs in both U and V, then x linearly occurs in $\langle U, V\rangle$, $U\&V$ and $\pi_i(U)$.*

The definition of occurrence is extended to an inhabited type and kind by stating that x occurs in $U{:}V$ if it occurs in U, in V, or in both. These notions are useful in the proof of the subject reduction property of the type theory. We remark, though, that these definitions are not "linear" in Girard's sense [3,1]. However, they seem quite natural in the bunched setting. O'Hearn and Pym give examples of **BI** terms — the $\lambda\Lambda$-calculus is in propositions-as-types correspondence with a non-trivial fragment of **BI** — in which linear variables appear more than once or not at all [15].

Example 1. The linear variable x occurs in the terms $cx{:}Bx$ (assuming $c : \Lambda x{:}A.Bx$), $fx{:}d$ (assuming $f{:}a \multimap d$) and $\lambda y{:}Cx.y : Cx \multimap Cx$ (assuming $C{:}A \multimap \mathsf{Type}$).

We refer informally to the concept of a linearity constraint. Essentially this means that all linear variables declared in the context are used. Given this, the judgement $x{:}A, y{:}cx \vdash_\Sigma y{:}cx$ in which the linear x is consumed by the (type of) y declared after it and the y itself is consumed in the succedent, is a valid one.

In the $\lambda\Lambda$-calculus, signatures are used to keep track of the types and kinds assigned to constants. Contexts are used to keep track of the types, both linear and intuitionistic, assigned to variables. The abstract syntax for signatures and contexts is given by the following grammar:

$$\Sigma ::= \langle\rangle \mid \Sigma, a!K \mid \Sigma, c!A \qquad \Gamma ::= \langle\rangle \mid \Gamma, x{:}A \mid \Gamma, x!A$$

The $\lambda\Lambda$-calculus is a formal system for deriving the following judgements:

$\vdash \Sigma$ sig	Σ is a valid signature	$\Gamma \vdash_\Sigma K$ Kind	K is a valid kind in Σ and Γ
$\vdash_\Sigma \Gamma$ context	Γ is a valid context in Σ	$\Gamma \vdash_\Sigma A{:}K$	A has a kind K in Σ and Γ
		$\Gamma \vdash_\Sigma M{:}A$	M has a type A in Σ and Γ

The definition of the type theory depends crucially on several notions to do with the joining and maintenance of contexts; these are the notions of *context joining*, *variable sharing* and *multiple occurrences*. These notions are crucial in allowing the formation of sufficiently complex linear dependent types and we discuss some of the rules of the type theory which exhibit them. The rules for extending contexts are as follows:

$$\Gamma, \frac{\vdash_\Sigma \Gamma \text{ context} \quad \Gamma \vdash_\Sigma A{:}\mathsf{Type}}{\vdash_\Sigma \Gamma, x{:}A \text{ context}} \qquad \Gamma! \frac{\vdash_\Sigma \Gamma \text{ context} \quad \Gamma \vdash_\Sigma A{:}\mathsf{Type}}{\vdash_\Sigma \Gamma, x!A \text{ context}}$$

The main point about these rules is that a context can be extended with either linear or intuitionistic variables. There is no zone or "stoup" separating the linear from the intuitionistic parts of the context.

Some of the interesting type formation rules are given next. The C rule exports type constants from the signature. This can only be allowed in an entirely intuitionistic context.

$$C\ \frac{\vdash_\Sigma !\Gamma\ \mathsf{sig} \quad a!K \in \Sigma}{!\Gamma \vdash_\Sigma a{:}K} \qquad \Lambda I2\ \frac{\Gamma \vdash_\Sigma A{:}\mathsf{Type} \quad \Delta \vdash_\Sigma B{:}\mathsf{Type}}{\Xi \vdash_\Sigma A \multimap B{:}\mathsf{Type}}\ \Xi = \Xi' \backslash ((lin(\Gamma) \cap lin(\Delta))^{[\Xi';\Gamma;\Delta]}$$

$$\Lambda I1\ \frac{\Gamma, x{:}A \vdash_\Sigma B{:}\mathsf{Type}}{\Gamma \vdash_\Sigma \Lambda x{:}A.B\ :\ \mathsf{Type}} \qquad \Lambda !I\ \frac{\Gamma, x!A \vdash_\Sigma B{:}\mathsf{Type}}{\Gamma \vdash_\Sigma \Lambda x!A.B\ :\ \mathsf{Type}}$$

The $\Lambda I1$ and $\Lambda I2$ rules form linear types. The second of these introduces the notion of context joining for binary multiplicative rules. The join must respect the ordering of the premiss contexts and the type of linear–intuitionistic variables. A method to join Γ and Δ to form Ξ, denoted by $[\Xi;\Gamma;\Delta]$, is defined in § 2.1 below. (The second side-condition is explained in Example 2 below.)

Some of the interesting object-level rules are given next. The two variable declaration rules, Var and $Var!$, declare linear and intuitionistic variables, respectively. These rules should not be seen as weakening in the context Γ as, by induction, the variables declared in Γ are "used" in the construction of the type A. In the rules for abstraction, λI and $\lambda !I$, the type of extension determines the type of function formed, just as in **BI**.

$$Var\ \frac{\Gamma \vdash_\Sigma A{:}\mathsf{Type}}{\Gamma, x{:}A \vdash_\Sigma x{:}A} \qquad Var!\ \frac{\Gamma \vdash_\Sigma A{:}\mathsf{Type}}{\Gamma, x!A \vdash_\Sigma x{:}A} \qquad \lambda I\ \frac{\Gamma, x{:}A \vdash_\Sigma M{:}B}{\Gamma \vdash_\Sigma \lambda x{:}A.M\ :\ \Lambda x{:}A.B}$$

$$\lambda !I\ \frac{\Gamma, x!A \vdash_\Sigma M{:}B}{\Gamma \vdash_\Sigma \lambda x!A.M\ :\ \Lambda x!A.B}$$

Before we give the rules for object-level application, we would like to motivate the notions of variable sharing and multiple occurrences. Consider the following example of a non-derivation:

Example 2. Let $A!\mathsf{Type}$, $c!A \multimap \mathsf{Type} \in \Sigma$ and note that the argument type, cx, is a dependent one; the linear x is free in it:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}A \vdash_\Sigma cx{:}\mathsf{Type}}}{x{:}A, z{:}cx \vdash_\Sigma z{:}cx}}{x{:}A \vdash_\Sigma \lambda z{:}cx.z\ :\ \Lambda z{:}cx.cx} \quad \dfrac{\dfrac{}{x{:}A \vdash_\Sigma cx{:}\mathsf{Type}}}{x{:}A, y{:}cx \vdash_\Sigma y{:}cx}}{x{:}A, x{:}A, y{:}cx \vdash_\Sigma (\lambda z{:}cx.z)y\ :\ cx}$$

The problem is that an excess of linear xs now appears in the combined context after the application step. This is due to the fact that an x each is needed for the well-formedness of each premiss type but only one x is needed for the well-formedness of the conclusion type. Our solution is to recognize the two xs as two *distinct* occurrences of the *same* variable, the one occurring in the argument type cx, and to allow a notion of sharing of this variable. One implication of this solution is that repeated declarations of the same

variable are allowed in contexts; there are side-conditions on the context formation rules which reflect this (but, for reasons of simplicity, were omitted before). It is now necessary to formally define a binding strategy for multiple occurrences; this we do in § 2.2 below. The sharing aspect is implemented via the κ function, defined in § 2.3.

We can now give the rules for function application. The side-condition on these is as follows: first, join the premiss contexts; then, apply κ to maintain the linearity constraint. It can be seen that these side-conditions are type-theoretically and, via the propositions-as-types correspondence, logically natural:

$$\lambda E\,\frac{\Gamma \vdash_\Sigma M : \Lambda x{:}A.B \quad \Delta \vdash_\Sigma N{:}A}{\Xi \vdash_\Sigma MN : B[N/x]}\;\begin{array}{l}[\Xi';\Gamma;\Delta]\\ \Xi = \Xi' \backslash \kappa(\Gamma,\Delta)\end{array}$$

$$\lambda!E\,\frac{\Gamma \vdash_\Sigma M : \Lambda x{:}!A.B \quad !\Delta \vdash_\Sigma N{:}A}{\Xi \vdash_\Sigma MN : B[N/x]}\;[\Xi;\Gamma;!\Delta]$$

An essential difference between linear and intuitionistic function application is that, for the latter, the context for the argument $N{:}A$ is an entirely intuitionistic one ($!\Delta$), which allows the function to use N as many times as it likes.

The definitional equality relation that we consider for the $\lambda\Lambda$-calculus is the β-conversion of terms at all three levels. The definitional equality relation, $\equiv$, between terms at each respective level is defined to be the symmetric and transitive closure of the parallel nested reduction relation. There is little difficulty (other than that for the $\lambda\Pi$-calculus [6,20]) in strengthening the definitional equality relation by the η-rule.

2.1 Context Joining

The method of joining two contexts is a ternary relation $[\Xi;\Gamma;\Delta]$ defined as follows:

$$\frac{}{[\langle\rangle;\langle\rangle;\langle\rangle]}\;\text{(JOIN)} \qquad \frac{[\Xi;\Gamma;\Delta]}{[\Xi,x!A;\Gamma,x!A;\Delta,x!A]}\;\text{(JOIN-!)} \qquad \frac{[\Xi;\Gamma;\Delta]}{[\Xi,x{:}A;\Gamma,x{:}A;\Delta]}\;\text{(JOIN-L)}$$

$$\frac{[\Xi;\Gamma;\Delta]}{[\Xi,x{:}A;\Gamma;\Delta,x{:}A]}\;\text{(JOIN-R)}$$

2.2 Multiple Occurrences

The type theory allows multiple occurrences of variables. For example, if $c{:}A \multimap A \multimap B$ and $x{:}A$, then cxx is a valid term. The two occurrences of x can be seen as different "colourings" of x. For the purposes of binding, we must be able to pick out the first occurrence of x from the second. We define the *left-most free occurrence* of x in U and a corresponding binding strategy for it.

The left-most linear occurrence of x in U is, basically, the first x, syntactically, in the body of U; *e.g.*, if $U = VM$, then the left-most occurrence of x is defined as follows:

$$\begin{array}{l} lm_x(@M) = lm_x(M) \quad x, @ \text{ distinct} \\ lm_x(xM) = \{x\} \end{array} \qquad lm_x(VM) = \begin{cases} lm_x(V) & x \in LFV(V) \\ lm_x(M) & \text{otherwise} \end{cases}$$

where @ ranges over atoms (constants or variables).

The left-most occurrence of $x{\in}A$ in a context Γ is the first declaration of $x{\in}A$ in Γ. Similarly, the right-most occurrence of $x{\in}A$ in Γ is the last such declaration. The following binding strategy now formalizes the concept of linearity constraint:

Definition 2. *Assume $\Gamma, x{:}A, \Delta \vdash_\Sigma U{:}V$ and that $x{:}A$ is the right-most occurrence of x in the context. Then x binds:*

1. *The first left-most occurrence of x in the types in Δ, if there is such a declaration;*
2. *The unbound left-most linear occurrences of x in $U{:}V$.*

There is no linearity constraint for intuitionistic variables: the right-most occurrence of $x!A$ in the context binds all the unbound xs used in the type of a declaration in Δ and all the occurrences of x in $U{:}V$.

The rules for deriving judgements are now read according to the strategy in place. For example, in the λI rule, the $\lambda(\Lambda)$ binds the left-most occurrence of x in $M(B)$. Similarly, in the (admissible) cut rule, the term $N{:}A$ cuts with the left-most occurrence of $x{:}A$ in the context $\Delta, x{:}A, \Delta'$. In the corresponding intuitionistic rules, the $\lambda!(\Lambda!)$ binds all occurrences of x in $M(B)$ and $N{:}A$ cuts all occurrences of $x!A$ in the context $\Delta, x!A, \Delta'$.

Example 3. Let $c!A \multimap A \multimap B \in \Sigma$. Then we can construct a derivation of the judgement $x{:}A, x{:}A \vdash_\Sigma cxx{:}B$ in which the bindings are coloured with the numbers 1 and 2 as follows: $x_2{:}A, x_1{:}A \vdash_\Sigma cx_1x_2{:}B$.

2.3 Variable Sharing

Sharing occurs when linear variables are needed for the well-formedness of the premiss types but not necessarily for the well-formedness of the conclusion type. This requirement is regulated by a function κ. We define κ by considering the situation when either of the two contexts Γ or Δ are of the form $\ldots, x{:}A$ or $\ldots, x{:}A, y{:}Bx$. The only case when the two declarations of $x{:}A$ are not identified with each other is when both Γ and Δ are of the form $\ldots, x{:}A, y{:}Bx$.

The function κ is defined for the binary, multiplicative rules as follows: For each $x{:}A$ occurring in the premiss contexts Γ and Δ, construct from right to left as follows:

$$\kappa(\Gamma, \Delta) = \{\} \qquad \textit{if } lin(\Gamma) \cap lin(\Delta) = \emptyset$$

$$\kappa(\Gamma, \Delta) = \{x{:}A \mid \text{either } (i)\ \textit{there is no } y{:}B(x)\ \textit{to the right of } x{:}A\ \textit{in } \Gamma \text{ or } (ii)\ \textit{there is no } y{:}B(x)\ \textit{to the right of } x{:}A\ \textit{in } \Delta \text{ or both } (i)\ \textit{and}\ (ii)\} \qquad \textit{otherwise}$$

In the absence of sharing of variables, when the first clause only applies, we still obtain a useful linear dependent type theory, with a linear dependent function space but without the dependency of the abstracting A_is on the previously abstracted variables.

Example 4. We can now correct Example 2. Suppose $A!\mathsf{Type}, c!A \multimap \mathsf{Type} \in \Sigma$. Then we construct the following:

$$\dfrac{\dfrac{\dfrac{}{x{:}A \vdash_\Sigma cx{:}\mathsf{Type}}}{x{:}A, z{:}cx \vdash_\Sigma z{:}cx} \quad \dfrac{}{x{:}A \vdash_\Sigma cx{:}\mathsf{Type}}}{x{:}A \vdash_\Sigma \lambda z{:}cx\,.z \,:\, \Lambda z{:}cx\,.cx} \quad \dfrac{}{x{:}A, y{:}cx \vdash_\Sigma y{:}cx}}{x{:}A, y{:}cx \vdash_\Sigma (\lambda z{:}cx\,.z)y \,:\, cx}\ \dagger$$

The † denotes the following action. First, the premiss contexts are joined together to get $x{:}A, x{:}A,\ y{:}cx$. Then, κ removes the extra occurrence of $x{:}A$ and so restores linearity.

The function κ is not required, *i.e.*, its use is vacuous, when certain restrictions of the $\lambda\Lambda$-calculus type theory are considered. For instance, if we restrict type-formation to be entirely intuitionistic so that type judgements are of the form $!\Gamma \vdash_\Sigma A{:}\mathsf{Type}$, then we get the $\{\Pi, \multimap, \&\}$-fragment of Cervesato and Pfenning's $\lambda^{\Pi\multimap\&\top}$ type theory [5].

A summary of the major meta-theorems, proved in [13], pertaining to the type theory and its reduction properties, is given by the following (other properties include admissibilities of structurals, unicities of types and subject reduction):

Theorem 1. *1. All well-typed terms are Church-Rosser.*
2. If $\Gamma \vdash_\Sigma U{:}V$, then U is strongly normalizing.
3. All assertions of the $\lambda\Lambda$-calculus are decidable. □

3 Kripke Resource Semantics

The semantics of **BI**, a fragment of which corresponds to the internal logic of the $\lambda\Lambda$-calculus, can be understood, categorically, by a single category which carries two monoidal structures. It can also be understood, model-theoretically, by a unique combination of two familiar ideas: a Kripke-style possible worlds semantics and an Urquhart-style resource semantics. We will use the internal logic and its semantics to motivate an indexed categorical semantics for the type theory: indeed, we require that our models provide a semantics for both the $\lambda\Lambda$-calculus as a presentation of its internal logic and as a theory of functions.

3.1 Kripke Resource $\lambda\Lambda$-Structure

The key issue in the syntax concerns the co-existing linear and intuitionistic function spaces and quantifiers. This distinction can be explained by reference to a *resource semantics*. The notion of resource, such as time and space, is a primitive in informatics. Essential aspects of a resource include our ability to identify elements (including the null element) of the resource and their combinations. Thus we work with a resource monoid $(R, +, 0)$. We can also imagine a partial order $\sqsubseteq$ between resources, indicating when one resource is better than another, in that it may prove more propositions.

A resource semantics elegantly explains the difference between the linear and intuitionistic connectives in that the action, or computation, of the linear connectives can be seen to consume resources. We consider this for the internal logic judgement $(X)\Delta \vdash \phi$. Let $\mathcal{M} = (M, \cdot, e, \sqsubseteq)$ be a Kripke resource monoid. The forcing relation for the two implications can be defined as follows:

1. $r \models \phi \to \psi$ iff for all $s \in M$, if $r \sqsubseteq s$ then $s \models \psi$
2. $r \models \phi \multimap \psi$ iff for all $s \in M$, if $s \models \phi$ then $r \cdot s \models \psi$

A similar pair of clauses defines the forcing relation for **BI**'s two quantifiers. Here $D{:}\mathcal{M}^{op} \to \mathbf{Set}$ is a domain of individuals and $u \in [\![X]\!]r$ is an environment appropriate to the bunch of variables X at world r, where $[\![X]\!]$ is the interpretation of the bunch of variables X in $\mathbf{Set}^{\mathcal{M}^{op}}$:

1. $(X)u, r \models \forall x.\phi$ iff for all $r \sqsubseteq s$ and all $d \in Ds$, $(X;x)([\![X]\!](r \sqsubseteq s)u, d), s \models \phi$
2. $(X)u, r \models \forall_{\mathrm{new}} x.\phi$ iff for all s and all $d \in Ds$, $(X, x)[u, d], r \cdot s \models \phi$

Here $(-,-)$ is cartesian pairing and $[-,-]$ is the pairing operation defined by Day's tensor product construction in $\mathbf{Set}^{\mathcal{M}^{op}}$. The resource semantics can be seen to combine Kripke's semantics for intuitionistic logic and Urquhart's semantics for relevant logic [14,22]; see [15,19].

Suppose we have a category $\mathcal{E}$ where the propositions will be interpreted. Then we will index $\mathcal{E}$ in two ways for the purposes of interpreting the type theory. First, we index it by a Kripke world structure $\mathcal{W}$. This is to let the functor category $[\mathcal{W}, \mathcal{E}]$ have enough strength to model the $\{\rightarrow, \forall\}$-fragment of the internal logic and so correspond to Kripke-style models for intuitionistic logic. Second, we index $[\mathcal{W}, \mathcal{E}]$ by a resource monoid R. Thus, we obtain R-indexed sets of Kripke functors $\{\mathcal{J}_r{:}[\mathcal{W}, \mathcal{E}] \mid r \in R\}$. We remark that the separation of worlds from resources considered in this structure emphasizes a sort of "phase shift" [8,11]. We reconsider this choice in § 4.

We now consider how to model the propositions and so explicate the structure of $\mathcal{E}$. The basic judgement of the internal logic is $(X)\Delta \vdash \phi$, that ϕ is a proposition in the context Δ over the context X. One reading of this judgement, and perhaps the most natural, is to see X as an index for the propositional judgement $\Delta \vdash \phi$. This reading can be extended to the type theory, where, in the basic judgement $\Gamma \vdash_\Sigma M{:}A$, Γ can be seen as an index for $M{:}A$ or that $M{:}A$ depends on Γ for its meaning. Thus we are led to using the technology of indexed category theory. More specifically, in the case of the type theory, the judgement $\Gamma \vdash_\Sigma M{:}A$ is modelled as the arrow $1 \stackrel{[\![M]\!]}{\rightarrow} [\![A]\!]$ in the fibre over $[\![\Gamma]\!]$ in the strict indexed category $\mathcal{E}{:}\mathcal{C}^{op} \rightarrow \mathbf{Cat}$. Alternative approaches to the semantics of (intuitionistic) dependent types are presented in, for example, Cartmell [4], Pitts [16]. These presentations lack the conceptual distinction provided by indexed categories.

We need the base category $\mathcal{C}$ to account for the structural features of the type theory and its internal logic, hence the following definition:

Definition 3. *A* doubly monoidal category *is a category $\mathcal{C}$ equipped with two monoidal structures, $(\otimes, I)$ and $(\times, 1)$. $\mathcal{C}$ is called* cartesian doubly monoidal *if $\times$ is cartesian. We will use $\bullet$ to range over both products.*

There are a couple of comments we need to make about the monoidal structure on $\mathcal{C}$. Firstly, there is no requirement that the bifunctors $\otimes$ and $\times$ be symmetric, as the contexts that the objects are intended to model are (ordered) lists. Secondly, the use of the symbol $\times$ as one of the context extension operators suggests that $\times$ is a cartesian product. This is indeed the case when $\{\mathcal{J}_r \mid r \in R\}$ is a model of the internal logic, where there are no dependencies within the variable context X, but not when $\{\mathcal{J}_r \mid r \in R\}$ is a model of the type theory, where there are dependencies within Γ. In the latter case, we have the property that for each object D extended by $\times$, there is a first projection map $p_{D,A}{:}D \times A \rightarrow D$. There is no second projection map $q_{D,A}{:}D \times A \rightarrow A$ in $\mathcal{C}$, as A by itself may not correspond to a well-formed type. For modelling the judgement $\Gamma, x{\in}A \vdash_\Sigma x{:}A$, we do, however, require the existence of a map $1 \stackrel{q}{\rightarrow} [\![A]\!]$ in the fibre over $[\![\Gamma]\!] \bullet [\![A]\!]$.

A doubly monoidal category $\mathcal{C}$ with both exponentials or, alternatively, $\mathcal{C}$ equipped with two monoidal closed structures $(\times, \rightarrow, 1)$ and $(\otimes, \multimap, I)$, is called a *cartesian doubly*

closed category (DCC) in O'Hearn and Pym [15,19]. Cartesian DCCs provide a class of models of **BI** in which both function spaces are modelled within $\mathcal{C}$. We will work with the barer doubly monoidal category, requiring some extra structure on the fibres to model the function space. This can be seen as a natural extension to the semantics of bunches to account for dependency. It can be contrasted to the Barber–Plotkin model of DILL [1], which uses a pair of categories, a monoidal one and a cartesian one, together with a monoidal adjunction between them. But this forces too much of a separation between the linear and intuitionistic parts of a context to be of use to us.

We now consider how the function spaces are modelled. In the intuitionistic case, the weakening functor $p^*_{D,A}$ has a right adjoint $\Pi_{D,A}$ which satisfies the Beck-Chevalley condition. In fact, this amounts to the existence of a natural isomorphism

$$cur_W : Hom_{\mathcal{J}_r(W)(D\times A)}(p^*_{D,A}(C), B) \cong Hom_{\mathcal{J}_r(W)(D)}(C, \Pi_{D,A}(B))$$

The absence of weakening for the linear context extension operator means that we can't model Λ in the same way. But the structure displayed above suggests a way to proceed. It is sufficient to require the existence of a natural isomorphism

$$\Lambda_{D,A} : Hom_{\mathcal{J}_{r+r'}(W)(D\bullet A)}(1, B) \cong Hom_{\mathcal{J}_r(W)(D)}(1, \Lambda x{\in}A.B)$$

in the indexed category. There are a couple of remarks that need to be made about the isomorphism. Firstly, it refers only to those hom-sets in the fibre whose source is 1. This restriction, which avoids the need to establish the well-foundedness of an arbitrary object over both D and $D \bullet A$, suffices to model the judgement $\Gamma \vdash_\Sigma M{:}A$ as an arrow $1 \stackrel{[\![M]\!]}{\to} [\![A]\!]$ in the fibre over $[\![\Gamma]\!]$: examples are provided by both the term and set-theoretic models that we will present later. The second remark we wish to make is that the extended context is defined in the $r + r'$-indexed structure. The reason for this can be seen by observing the form of the forcing clause for application in **BI**. Given these two remarks, the above isomorphism allows the formation of function spaces.

Definition 4. *Let $(R, +, 0)$ be a commutative monoid (of "resources"). A* Kripke resource $\lambda\Lambda$-structure *is an R-indexed set of functors $\{\mathcal{J}_r{:}[\mathcal{W}, [\mathcal{C}^{op}, \mathbf{Cat}]] \mid r \in R\}$ where $\langle \mathcal{W}, \leq\rangle$ is a poset, $\mathcal{C}^{op} = \amalg_{W\in\mathcal{W}} \mathcal{C}^{op}_W$, where $W \in \mathcal{W}$ and each $\mathcal{C}_W$ is a small doubly monoidal category with $1 \cong I$, and* **Cat** *is the category of small categories and functors such that:*

1. *Each $\mathcal{J}_r(W)(D)$ has a terminal object, $1_{\mathcal{J}_r(W)(D)}$, preserved on the nose by each $f^* = \mathcal{J}_r(W)(f))$, where $f{:}E \to D \in \mathcal{C}_W$;*
2. *For each $W \in \mathcal{W}$, $D \in \mathcal{C}_W$ and object $A \in \mathcal{J}_r(W)(D)$, there is a $D \bullet A \in \mathcal{C}_W$. For the cartesian extension, there are canonical first projections $D \times A \stackrel{p_{D,A}}{\to} D$ and canonical pullbacks $\begin{array}{ccc} E \times f^*(A) & \xrightarrow{f\times 1_A} & D \times A \\ {\scriptstyle p_{E,f^*A}}\downarrow & & \downarrow{\scriptstyle p_{D,A}} \\ E & \xrightarrow[f]{} & D \end{array}$. The pullback indicates, for the cartesian case, how to interpret realizations as tuples. In particular, for each $1 \stackrel{M}{\to} \mathcal{J}_r(W)(D)$, there exists a unique arrow $D \stackrel{\langle 1\times M\rangle}{\to} D \times A$. It does not cover the case for the monoidal extension. For that, we require there to exist a unique $D(= D \otimes I) \stackrel{\langle 1\otimes M\rangle}{\to} D \otimes A$, the tuples being given by the bifunctoriality of $\otimes$. For both extensions, there is a canonical second projection $1 \stackrel{q_{D,A}}{\to} A$ in the fibre over $D \bullet A$.*

These maps are required to satisfy the strictness conditions that $(1_D)^*(A) = A$ *and* $1_D \bullet 1_A = 1_{D \bullet A}$ *for each* $A \in \mathcal{J}_r(W)(D)$*;* $g^*(f^*(A)) = (g;f)^*(A)$ *and* $(g \bullet f^*(A)); f \bullet A = (g;f) \bullet A$ *for each* $F \xrightarrow{g} E$ *and* $E \xrightarrow{f} D$ *in* $\mathcal{C}_W$*. Moreover, for each* W *and* D*,* $D \bullet 1_{\mathcal{J}_r(W)(D)} = D$ *;*

3. *For each* D*,* A*, there is a natural isomorphism* $\Lambda_{D,A}$

$$\Lambda_{D,A} : Hom_{\mathcal{J}_{r+r'}(W)(D \bullet A)}(1, B) \cong Hom_{\mathcal{J}_r(W)(D)}(1, \Lambda x{\in}A.B)$$

where the extended context is defined in the $r + r'$*-indexed functor. This natural isomorphism is required to satisfy the Beck-Chevalley condition: For each* $E \xrightarrow{f} D$ *in* $\mathcal{C}_W$ *and each* B *in* $\mathcal{J}_r(W)(D \bullet A)$*,* $f^*(\Lambda_{D,A} B) = \Lambda_{E,f^*A}((f \bullet id_A)^* B)$*;*

4. *Each category* $\mathcal{J}_r(W)(D)$ *has cartesian products.*

Our approach is modular enough to also provide a categorical semantics for the intuitionistic fragment of the $\lambda\Lambda$-calculus, the $\lambda\Pi$-calculus. Basically, we work with a single functor $\mathcal{J}{:}[\mathcal{W}, [\mathcal{D}^{op}, \mathbf{Cat}]]$, where $\mathcal{D}$ is a category with only the cartesian structure $(\times, 1)$ on it. The definition of Π as right adjoint to weakening can be recovered from the natural isomorphism. We see this in the following lemma, which is motivated by the propositions-as-types correspondence that we gave earlier:

Lemma 1. *The natural iso* $Hom_{\mathcal{J}(W)(D \times A)}(p^*_{D,A}(1), B) \cong Hom_{\mathcal{J}(W)(D)}(1, \Pi_{D,A}(B))$ *in the Kripke* $\lambda\Pi$*-structure* $\mathcal{J}$ *is just the* $\Lambda_{D,A}$ *natural isomorphism in the* $D \times A$ *case in the Kripke resource* $\lambda\Lambda$*-structure.* □

For the proof, we provide a translation from a Kripke resource $\lambda\Lambda$-structure to a Kripke $\lambda\Pi$-structure which forgets the linear–intuitionistic distinction, translating both $\otimes$ and $\times$ in $\mathcal{C}$ to $\times$ in $\mathcal{D}$. The translation has some similarity with Girard's translation of $- \to -$ into $!- \multimap -$ [8]. Under the translation, the object $\Lambda x{!}A.B$ (in a particular $\mathcal{J}_r$) ends up as $\Pi\, x{:}A.B$ (in $\mathcal{J}$). If we uncurry this, we get the corresponding translation, as $p_{D,A}{:}D \times A \to D$ always exists in $\mathcal{C}$.

3.2 Kripke Resource Σ-$\lambda\Lambda$-Models

We will restrict our discussion of semantics to the $M{:}A$-fragment. The treatment of the $A{:}K$-fragment is undertaken analogously — in a sense, the $A{:}K$-fragment has the same logical structure as the $M{:}A$-fragment but needs some extra structure. To interpret the kind Type, for instance, we must require the existence of a chosen object which obeys some equations regarding substitution and quantification. A treatment of the intuitionistic case is in Streicher [21].

A Kripke resource model is a Kripke resource structure that has enough points to interpret not only the constants of Σ but also the $\lambda\Lambda$-calculus terms defined over Σ and a given context Γ. Formally, a Kripke resource model is made up of five components: a Kripke resource structure that has Σ-operations, an interpretation function, two $\mathcal{C}$-functors, and a satisfaction relation. Except for the structure, the components are defined, due to inter-dependences, simultaneously by induction on raw syntax.

Ignoring these inter-dependencies for a moment, we explain the purpose of each component of the model. First, the Kripke resource structure provides the abstract domain where the type theory is interpreted in. The Σ-operations provide the points to interpret constants in the signature. Second, the interpretation $[\![-]\!]$ is a partial function, mapping raw (that is, not necessary well-formed) contexts Γ to objects of $\mathcal{C}$, types over raw contexts A_Γ to objects in the category indexed by the interpretation of Γ, and terms over raw contexts M_Γ to arrows in the category indexed by the interpretation of Γ. Types and terms are interpreted up to $\beta\eta$-equivalence. Fourth, the $\mathcal{C}$-functors maintain the well-formedness of contexts with regard to joining and sharing. The model also needs to be constrained so that multiple occurrences of variables in the context get the same interpretation. Fifth, satisfaction is a relation on worlds and sequents axiomatizing the desired properties of the model. In stronger logics, such as intuitionistic logic, the abstract definition of the model is sufficient to derive the properties of the satisfaction relation. In our case, the definition has to be given more directly. We give only part of the definition of the model below, the actual definition being long and complex.

Definition 5. *Let Σ be a $\lambda\Lambda$-calculus signature. A* Kripke resource Σ-$\lambda\Lambda$ model *is a 5-tuple $\langle \{\mathcal{J}_r{:}[\mathcal{W}, [\mathcal{C}^{op}, \mathbf{Cat}]] \mid r \in R\}, [\![-]\!], \mathrm{join}, \mathrm{share}, \models_\Sigma \rangle$ defined by simultaneous induction on the raw structure of the syntax as follows (we omit most of the clauses for reasons of brevity):*

1. *$\{\mathcal{J}_r{:}[\mathcal{W}, [\mathcal{C}^{op}, \mathbf{Cat}]] \mid r \in R\}$ is a Kripke resource $\lambda\Lambda$-structure that has Σ-*operations. *That is, for all W in $\mathcal{W}$ there is, corresponding to each constant in the signature, an operation or arrow in each fibre $\mathcal{J}_r(W)(D)$ that denotes the constant;*
2. *An* interpretation $[\![-]\!]^-_{\mathcal{J}_r}$ *from the raw syntax of the $\lambda\Lambda$-calculus to components of the structure satisfies, at each W, at least the following clauses:*

   (a) $[\![\Gamma, x{:}A]\!]^W_{\mathcal{J}_{r+r'}} \simeq [\![\Gamma]\!]^W_{\mathcal{J}_r} \otimes [\![A_\Gamma]\!]^W_{\mathcal{J}_{r'}}$; *(b)* $[\![\Gamma, x{!}A]\!]^W_{\mathcal{J}_r} \simeq [\![\Gamma]\!]^W_{\mathcal{J}_r} \times [\![A_\Gamma]\!]^W_{\mathcal{J}_r}$;

   (c) $[\![c_{!\Gamma}]\!]^W_{\mathcal{J}_r} \simeq \Lambda^m(op_c)$; *(d)* $[\![x_{\Gamma,x:A}]\!]^W_{\mathcal{J}_r} \simeq q_{[\![\Gamma,x:A]\!]^W_{\mathcal{J}_r}}$;

   (e) $[\![\lambda x{:}A.M_\Gamma]\!]^W_{\mathcal{J}_r} \simeq \Lambda_{[\![\Gamma]\!]^W_{\mathcal{J}_r}, [\![A_\Gamma]\!]^W_{\mathcal{J}_r}}([\![M_{\Gamma,x:A}]\!]^W_{\mathcal{J}_r})$;

   (f) $[\![MN_\Xi]\!]^W_{\mathcal{J}_t} \simeq (\langle 1_{[\![\Gamma]\!]^W_{\mathcal{J}_r}}, [\![N_\Delta]\!]^W_{\mathcal{J}_s}\rangle^*(\Lambda([\![M_\Gamma]\!]^W_{\mathcal{J}_r})))$, *where* $[\![\Xi']\!]^W_{\mathcal{J}_{r+s}} = \mathrm{join}\langle [\![\Gamma]\!]^W_{\mathcal{J}_r}, [\![\Delta]\!]^W_{\mathcal{J}_s}\rangle$ *and* $[\![\Xi]\!]^W_{\mathcal{J}_t} = \mathrm{share}([\![\Xi']\!]^W_{\mathcal{J}_{r+s}})$,

   where, in (c), the arity of c is m and Λ^m denotes m applications of Λ.
3. join *and* share *are two $\mathcal{C}$-functors that maintain context well-formedness;*
4. Satisfaction *in the model is a relation over worlds and sequents such that at least the following holds: $\mathcal{J}_r, W \models_\Sigma (M : \Lambda x{:}A.B)\ [\Gamma]$ iff for all $W \leq W'$ and for all $r' \in R$, if $\mathcal{J}_s, W' \models_\Sigma (N{:}A)\ [\Delta]$, then $\mathcal{J}_t, W' \models_\Sigma (MN{:}B[N/x])\ [\Xi]$, where $[\![\Xi]\!]^{W'}_{\mathcal{J}_{r+s}} = \mathrm{join}\langle [\![\Gamma]\!]^{W'}_{\mathcal{J}_r}, [\![\Delta]\!]^{W'}_{\mathcal{J}_s}\rangle$ and $[\![\Xi]\!]^{W'}_{\mathcal{J}_t} = \mathrm{share}([\![\Xi']\!]^{W'}_{\mathcal{J}_{r+s}})$, and analogously for the intuitionistic case.*

We require two conditions on the model: syntactic monotonicity (if X is defined, then so are sub-terms of X) and Kripke accessibility (the interpretation of X, if defined, is the same in all accessible worlds — there is no "relativization").

Given an appropriate notion of validity in the model, we obtain:

Theorem 2. *Soundness and completeness:* $\Gamma \vdash_\Sigma M{:}A$ *iff* $\Gamma \models_\Sigma M{:}A$. □

We sketch the argument for completeness, first giving an algebraic presentation of the type theory.

Definition 6. *Let* Σ *be a signature. The base category* $\mathcal{C}(\Sigma)$ *of contexts and realizations is defined as follows:*

- *Objects: contexts* Γ *such that* **N** *proves* $\vdash_\Sigma \Gamma$ context*;*
- *Arrows: realizations* $\Gamma \stackrel{\langle M_1,\ldots,M_n\rangle}{\longrightarrow} \Delta$ *such that* **N** *proves* $\Gamma \vdash_\Sigma (M_i{:}A_i)[M_j/x_j]_{j=1}^{i-1}$*, where* $\Delta = x_1{\in}A_1,\ldots,x_n{\in}A_n$.
  - *Identities are* $x_1{\in}A_1,\ldots,x_n{\in}A_n \stackrel{\langle x_1,\ldots,x_n\rangle}{\longrightarrow} x_1{\in}A_1,\ldots,x_n{\in}A_n$*. We will write the identity arrow on* Γ *as* 1_Γ*;*
  - *Composition is given by substitution. If* $f = \Gamma \stackrel{\langle M_1,\ldots,M_n\rangle}{\longrightarrow} \Delta$ *and* $g = \Delta \stackrel{\langle N_1,\ldots,N_p\rangle}{\longrightarrow} \Theta$*, then* $f;g = \Gamma \stackrel{\langle N_1[M_j/y_j]_{j=1}^n,\ldots,N_p[M_j/y_j]_{j=1}^n\rangle}{\longrightarrow} \Theta$.

$\mathcal{C}(\Sigma)$ is doubly monoidal because of the two ways of extending the context.

Definition 7. *We inductively define a strict indexed category* $\mathcal{E}(\Sigma){:}\mathcal{C}(\Sigma)^{op} \to$ **Cat** *over the base category* $\mathcal{C}(\Sigma)$ *as follows:*

- *For each* Γ *in* $\mathcal{C}(\Sigma)$*, the category* $\mathcal{E}(\Sigma)(\Gamma)$ *is defined as follows:*
  - *Objects: Types* A *such that* **N** *proves* $\Gamma \vdash_\Sigma A{:}$Type*;*
  - *Morphisms:* $A \stackrel{M}{\to} B$ *where the object* M *is such that* $\Gamma, x{:}A \stackrel{M}{\to} y{:}B$ *in* $\mathcal{C}(\Sigma)$*. Composition is given by substitution;*
- *For each* $f{:}\Gamma \to \Delta$ *in* $\mathcal{C}(\Sigma)$*,* $\mathcal{E}(\Sigma)(f)$ *is a functor* $f^*{:}\mathcal{E}(\Sigma)(\Delta) \to \mathcal{E}(\Sigma)(\Gamma)$ *given by* $f^*(A) \stackrel{\text{def}}{=} A[f]$ *and* $f^*(M) \stackrel{\text{def}}{=} M[f]$.

We remark that each $\mathcal{C}(\Sigma)(\Gamma)$ is a category. Note that the identity arrow $A \stackrel{1}{\to} A$ over Γ is given by the term $\lambda x{:}A.x$, corresponding to the definition of morphisms above. To see that this construction is correct, consider that the axiom sequent is form $\Gamma, x{:}A \vdash_\Sigma x{:}A$, with the side-condition that $\Gamma \vdash_\Sigma A{:}$Type, thereby using the variables in Γ.

Returning to the discussion about the completeness theorem, the syntactic category of contexts, $\mathcal{C}(\Sigma)$, is used to define other components of the term structure too. The indexing monoid $(R, +, 0)$ consists of the objects of $\mathcal{C}(\Sigma)$ combined with the joining relation $[-;-;-]$. The empty context is the monoid unit. The world structure $\mathcal{P}(\Sigma)$ is $\mathcal{C}(\Sigma)$ restricted to only intuitionistic extension.

We can then give an appropriate model existence lemma. The Kripke resource structure $\{\mathcal{T}(\Sigma)_\Delta{:}[\mathcal{P}(\Sigma), [\mathcal{C}(\Sigma)^{op}, \mathbf{Cat}]] \mid \Delta \in obj(\mathcal{C}(\Sigma))\}$ has $\mathcal{C}(\Sigma)^{op} = \amalg_{W\in\mathcal{C}(\Sigma)}\mathcal{C}(\Sigma)^{op}_W$. $\mathcal{T}(\Sigma)_\Delta(\Theta)(\Gamma)$ is the category consisting of those types and terms which can be defined over the sharing-sensitive join of Δ, Θ and Γ. The Σ-operations of the model are given by the constants declared in the signature Σ. The functors $join$ and $share$ are defined by $[-;-;-]$ and κ, respectively. The interpretation function is the obvious one in which a term (type) is interpreted by the class of terms (types) definitionally equivalent to the term (type) in the appropriate component of the structure. The satisfaction relation is given by provability in the type theory. Details of the proof are in [12].

4 A Class of Set-Theoretic Models

We describe a class of set-theoretic Kripke resource models, in which the Kripke resource $\lambda\Lambda$-structure $\{\mathcal{J}_r{:}[\mathcal{W}, [\mathcal{C}^{op}, \mathbf{Cat}]] \mid r \in R\}$ is given by $\mathbf{BIFam}{:}[\mathcal{C}, [Ctx^{op}, \mathbf{Set}]]$, where $\mathcal{C}$ is a small monoidal category and Ctx is a small set-theoretic category of "contexts". The model is a construction on the category of families of sets and exploits Day's construction to define the linear dependent function space.

We begin with the indexed category of families of sets, $\mathbf{Fam}{:}[Ctx^{op}, \mathbf{Cat}]$. The base, Ctx, is a small set-theoretic category defined inductively as follows. The objects of Ctx, called "contexts", are (*i.e.*, their denotations are) sets and the arrows of Ctx, called "realizations", are set-theoretic functions. For each $D \in obj(Ctx)$, $\mathbf{Fam}(D) = \{y \in B(x) \mid x \in D\}$. The fibre can be described as a discrete category whose objects are the ys and whose arrows are the maps $1_y{:}y \to y$ corresponding to the identity functions $id{:}\{y\} \to \{y\}$ on y considered as a singleton set. If $E \xrightarrow{f} D$ is an arrow in Ctx, then $\mathbf{Fam}(f) = f^*{:}\mathbf{Fam}(D) \to \mathbf{Fam}(E)$ re-indexes the set $\{y \in B(x) \mid x \in D\}$ over D to the set $\{f(z) \in B(f(z)) \mid z \in E\}$ over E. We are viewing **Set** within **Cat**; each object of **Set** is seen as an object, a discrete category, in **Cat**. Because of this, the category of families of sets can just be considered as a presheaf $\mathbf{Fam}{:}[Ctx^{op}, \mathbf{Set}]$, rather than as an indexed category; we will adopt this view in the sequel.

We can explicate the structure of Ctx by describing **Fam** as a contextual category [4]. The following definition is from Streicher [21]:

Definition 8. *The contextual category* **Fam**, *along with its denotation* $\mathrm{DEN}{:}\mathbf{Fam} \to \mathbf{Set}$ *and length, is described as follows: (1)* 1 *is the unique context of length* 0 *and* $\mathrm{DEN}(1) = \{\emptyset\}$*; (2) If* D *is a context of length* n *and* $A{:}\mathrm{DEN}(D) \to \mathbf{Set}$ *is a family of sets indexed by elements of* $DEN(D)$*, then* $D \times A$ *is a context of length* $n+1$ *and* $\mathrm{DEN}(D \times A) = \{\langle x, y\rangle \mid x \in \mathrm{DEN}(D),\ y \in A(x)\}$*. If* D *and* E *are objects of the contextual category* **Fam**, *then the morphisms between them are simply the functions between* $\mathrm{DEN}(D)$ *and* $\mathrm{DEN}(E)$.

The codomain of the denotation, **Set**, allows the definition of an extensional context extension $\times$. But **Set** does not have enough structure to define an intensional context extension $\otimes$. In order to be able to define both $\times$ and $\otimes$, we denote **Fam** not in **Set** but in a presheaf $\mathbf{Set}^{\mathcal{C}^{op}}$, where $\mathcal{C}$ is a monoidal category. We emphasize that, in general, $\mathcal{C}$ can be any monoidal category and, therefore, we are actually going to describe a class of set-theoretic models. For simplicity, we take $\mathcal{C}^{op}$ to be a partially ordered commutative monoid $\mathcal{M} = (M, \cdot, e, \sqsubseteq)$. The cartesian structure on the presheaf gives us the $\times$ context extension and a restriction of Day's tensor product [7] gives us the $\otimes$ context extension.

We remark that the restriction of Day's tensor product we consider is merely this: consider the set-theoretic characterization of Day's tensor product as tuples $\langle x, y, f\rangle$ and, of all such tuples, consider only those where the y is an element of the family of sets in x. This is quite concrete, in the spirit of the Cartmell–Streicher models, and is not a general construction for a fibred Day product.

Within the contextual setting, we then have the following definition:

Definition 9. *The contextual category* **BIFam**, *together with its denotation* $\mathrm{DEN}{:}\mathbf{BIFam} \to \mathbf{Set}^{\mathcal{M}}$ *and length, is described as follows:*

1. 1 *is a context of length* 0 *and* $\mathrm{DEN}(1)(Z) = \{\emptyset\}$*;*
2. I *is a context of length* 0 *and* $\mathrm{DEN}(I)(-) = \mathcal{M}[-, I]$*;*
3. *If* D *is a context of length* n *and* $A{:}\mathrm{DEN}(D)(X) \to \mathbf{Set}^{\mathcal{M}}$ *is a family of* $\mathcal{M}$*-sets indexed by elements of* $DEN(D)(X)$*, then*
   a) $D \times A$ *is a context of length* $n+1$ *and*

$$\mathrm{DEN}(D \times A)(X) = \{\langle x, y\rangle \mid x \in \mathrm{DEN}(D)(X),\ y \in (A(x))(X)\}$$

   b) $D \otimes A$ *is a context of length* $n+1$ *and*

$$\mathrm{DEN}(D \otimes A)(Z) = \{\langle x, y, f\rangle \in \textstyle\int^{X,Y} \mathrm{DEN}(D)(X) \times (A(x))(Y) \times \mathcal{M}[Z, X \otimes Y]\}$$

If D *and* E *are objects of* **BIFam***, then the morphisms between them are the functions between* $\mathrm{DEN}(D)(X)$ *and* $\mathrm{DEN}(E)(Y)$*.* **BIFam** *is* **Fam** *parametrized by* $\mathcal{M}$*; objects that were interpreted in* **Set** *are now interpreted in* $\mathbf{Set}^{\mathcal{M}}$*.*

Now, by our earlier argument relating the indexed and contextual presentations of families of sets, **BIFam** can be seen as a functor category $\mathbf{BIFam}{:}[Ctx^{op}, \mathbf{Set}^{\mathcal{M}}]$. This is not quite the presheaf setting we require. However, if we calculate $[Ctx^{op}, [\mathbf{Set}^{\mathcal{M}}]] \cong [Ctx^{op} \times \mathcal{M}, \mathbf{Set}] \cong [\mathcal{M} \times Ctx^{op}, \mathbf{Set}] \cong [\mathcal{M}, [Ctx^{op}, \mathbf{Set}]]$ then this restores the indexed setting and also reiterates the idea that $\mathcal{M}$ parameterizes **Fam**. The right-adjoint to $\otimes$, given by Day's construction, provides the isomorphism needed to define the linear dependent function space.

Lastly, we say what the R and $\mathcal{W}$ components of the concrete model are. Define $(R, +, 0) = (M, \cdot, e)$ and define $(\mathcal{W}, \leq) = (M/\sim, \sqsubseteq)$, where the quotient of M by the relation $w \sim w \cdot w$ is necessary because of the separation of worlds from resources (*cf.* **BI**'s semantics [22,15,19]). This allows us to define $\mathcal{J}_r(w) = \mathbf{BIFam}(r \cdot w)$. The quotiented $\mathcal{M}$ maintains the required properties of monotonicity and bifunctoriality of the internal logic forcing relation. We then check that $\mathbf{BIFam}(r \cdot w)$ does simulate $\mathcal{J}_r(w)$, and that **BIFam** is a Kripke resource $\lambda\Lambda$-structure [12].

Theorem 3. $\mathbf{BIFam}{:}[\mathcal{M}, [Ctx^{op}, \mathbf{Set}]]$ *is a Kripke resource* $\lambda\Lambda$*-structure and can be extended to a Kripke resource model.* □

Definition 9 above comprises the main part of the proof that **BIFam** is a Kripke resource structure. It describes how Ctx can have two kinds of extension. These extensions are then used to describe two kinds of function space in **BIFam**. For the linear case, for instance, $\Lambda x{:}A.B$ is defined as the following set

$$\{f : \mathbf{BIFam}(Y)(A(x)) \to \cup_y\{\mathbf{BIFam}(X \otimes Y)(B(x,y)) \mid y \in \mathbf{BIFam}(Y)(A(x))\} \mid \forall a \in \mathbf{BIFam}(Y)(A(x))\ f(a) \in \mathbf{BIFam}(X \otimes Y)(B(x,a))\}$$

where $x \in \mathbf{BIFam}(X)(D)$. The intuitionistic function space is defined analogously, with the "resource" X over which the sets are defined staying the same. The natural isomorphism is given by abstraction and application in this setting.

In order to extend **BIFam** to a model, the structure must have enough points to interpret the constants of the signature. We can work with an arbitrary signature and interpret

constants and variables as the functors $Const{:}\mathcal{M} \to \mathbf{Set}$ and $D{:}\mathcal{M} \to \mathbf{Set}$ respectively. The interpretation function $[\![-]\!]^X_{\mathbf{BIFam}}$ is parametrized over worlds–resources X. The interpretation of contexts is defined using the same idea as the construction of the category Ctx:

1. $[\![\Gamma, x{:}A]\!]^{X\otimes Y}_{\mathbf{BIFam}} \simeq [\![\Gamma]\!]^X_{\mathbf{BIFam}} \otimes [\![A_\Gamma]\!]^Y_{\mathbf{BIFam}}$; 2. $[\![\Gamma, x!A]\!]^X_{\mathbf{BIFam}} \simeq [\![\Gamma]\!]^X_{\mathbf{BIFam}} \times [\![A_\Gamma]\!]^X_{\mathbf{BIFam}}$

.

The interpretation of functions is defined using abstraction and application. We must also define instances of the functors $join$ and $share$ for this setting; these are defined along the same lines as those for the term model. Finally, satisfaction is a relation over M and $[Ctx^{op}, \mathbf{Set}]$ with the clauses reflecting the properties — in particular, those of application — of the example model:

1. $X \models_\Sigma f{:}\Lambda x{:}A.B\ [D]$ iff $Y \vDash a{:}A\ [E]$ implies $X \otimes Y \models_\Sigma f(a){:}B[a/x]\ [D \otimes E]$;
2. $X \models_\Sigma f{:}\Pi\, x{:}A.B\ [D]$ if and only if $X \models_\Sigma a{:}A\ [E]$ implies $X \models_\Sigma f(a){:}B[a/x]\ [D \times E]$.

References

[1] A Barber. *Linear Type Theories, Semantics and Action Calculi*. PhD thesis, University of Edinburgh, 1997.
[2] HP Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.
[3] PN Benton. A mixed linear and non-linear logic: Proofs, terms and models (preliminary report). Technical Report 352, Computer Laboratory, University of Cambridge, 1994.
[4] J Cartmell. Generalised algebraic theories and contextual categories. *Ann. Pure Appl. Logic*, 32:209–243, 1986.
[5] I Cervesato and F Pfenning. A linear logical framework. In E Clarke, editor, *11th LICS, New Brunswick, NJ*, pages 264–275. IEEE Computer Society Press, 1996.
[6] T Coquand. An algorithm for testing conversion in type theory. In G Huet and G Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.
[7] BJ Day. On closed categories of functors. In S Mac Lane, editor, *Reports of the Midwest Category Seminar*, volume 137 of *Lecture Notes in Mathematics*, pages 1–38. Springer-Verlag, 1970.
[8] J-Y Girard. Linear logic. *Theoret. Comput. Sci.*, 50(1):1–102, 1987.
[9] R Harper, F Honsell, and G Plotkin. A framework for defining logics. *JACM* 40(1):143–184, January 1993.
[10] R Harper, D Sannella, and A Tarlecki. Structured theory representations and logic representations. *Ann. Pure Appl. Logic*, 67:113–160, 1994.
[11] JS Hodas and D Miller. Logic programming in a fragment of intuitionistic linear logic. *Informat. Computat.*, 110(2):327–365, 1994.
[12] S Ishtiaq. *A Relevant Analysis of Natural Deduction*. PhD thesis, Queen Mary & Westfield College, University of London, 1999.
[13] SS Ishtiaq and DJ Pym. A Relevant Analysis of Natural Deduction. *J. Logic Computat.*, 8(6):809–838, 1998.
[14] S Kripke. Semantic analysis of intuitionistic logic I. In JN Crossley and MAE Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. North-Holland, 1965.
[15] PW O'Hearn and DJ Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215-244, 1999.
[16] A Pitts. Categorical logic. In S Abramsky, D Gabbay, and TSE Maibaum, editors, *Handbook of Logic in Computer Science*, volume 6. Oxford, 1992.
[17] DJ Pym. A relevant analysis of natural deduction. Lecture at Workshop, EU Espirit Basic Research Action 3245, Logical Frameworks: Design, Implementation and Experiment, Båstad, Sweden, May 1992.
[18] DJ Pym. A note on representation and semantics in logical frameworks. In *Proc. CADE-13 Workshop on Proof-search in type-theoretic languages*, Rutgers University, New Brunswick, NJ, D. Galmiche, ed., 1996. (Also available as Technical Report 725, Department of Computer Science, Queen Mary and Westfield College, University of London.)
[19] DJ Pym. On Bunched Predicate Logic. In *Proc. LICS '99*, Trento, Italy. IEEE Computer Society Press, 1999.
[20] A Salvesen. A proof of the Church-Rosser property for the Edinburgh LF with η-conversion. Lecture given at the First Workshop on Logical Frameworks, Sophia-Antipolis, France, May 1990.
[21] T Streicher. *Correctness and Completeness of a Categorical Semantics of the Calculus of Constructions*. PhD thesis, Universität Passau, 1988.
[22] A Urquhart. Semantics for relevant logics. *J. Symb. Logic*, 37(1):159–169, March 1972.

A Linear Logical View of Linear Type Isomorphisms

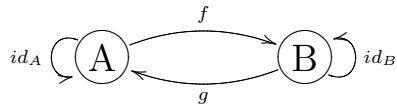
Vincent Balat and Roberto Di Cosmo

LIENS
École normale supérieure
45 rue d'Ulm
75005 Paris, France
balat@dmi.ens.fr dicosmo@dmi.ens.fr

Abstract. The notion of isomorphisms of types has many theoretical as well as practical consequences, and isomorphisms of types have been investigated at length over the past years. Isomorphisms in weak system (like linear lambda calculus) have recently been investigated due to their practical interest in library search. In this paper we give a remarkably simple and elegant characterization of linear isomorphisms in the setting of Multiplicative Linear Logic (MLL), by making an essential use of the correctness criterion for Proof Nets due to Girard.

1 Introduction and Survey

The interest in building models satisfying specific isomorphisms of types (or domain equations) is a long standing one, as it is a crucial problem in the denotational semantics of programming languages. In the 1980s, though, some interest started to develop around the dual problem of finding the domain equations (type isomorphisms) that must hold in *every* model of a given language. Alternatively, one could say that we are looking for those objects that can be encoded into one-another by means of conversion functions f and g *without* loss of information, i.e. such that the following diagram commutes



The seminal paper by Bruce and Longo [5] addressed the case of pure first and second order typed λ-calculus with essentially model-theoretic motivations, but due to the connections between typed λ-calculus, cartesian closed categories, proof theory and functional programming, the notion of *isomorphism of types* showed up as a central idea that translates easily in each of those different but related settings. In the framework of category theory, Soloviev already studied the problem of characterizing types (objects) that are isomorphic in every cartesian closed category, providing a model theoretic proof of completeness for the

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 250–265, 1999.

theory $Th^1_{\times T}$ in Table 1 (actually [21] is based on techniques used originally in [15], while another different proof can be found in [16]). A treatment of this same problem by means of purely syntactic methods for a λ-calculus extended with surjective pairing and unit type was developed in [4], where the relations between these settings, category theory and proof theory, originally suggested by Mints, have been studied, and pursued further on in [11]. Finally, [7] provides a complete characterization of valid isomorphisms of types for second order λ-calculus with surjective pairing and terminal object type, that includes all the previously studied systems (see table 1).

$$(\textbf{swap}) \quad A \to (B \to C) = B \to (A \to C) \Big\} Th^1$$

$$\left.\begin{array}{ll} 1. & A \times B = B \times A \\ 2. & A \times (B \times C) = (A \times B) \times C \\ 3. & (A \times B) \to C = A \to (B \to C) \\ 4. & A \to (B \times C) = (A \to B) \times (A \to C) \\ 5. & A \times \mathbf{T} = A \\ 6. & A \to \mathbf{T} = \mathbf{T} \\ 7. & \mathbf{T} \to A = A \end{array}\right\} Th^1_{\times T}$$

$$\left.\begin{array}{ll} 8. & \forall X.\forall Y.A = \forall Y.\forall X.A \\ 9. & \forall X.A = \forall Y.A[Y/X] \quad {}^{1} \\ 10. & \forall X.(A \to B) = A \to \forall X.B \quad {}^{2} \end{array}\right\} + \textbf{swap} = Th^2$$

$$\left.\begin{array}{ll} 11. & \forall X.A \times B = \forall X.A \times \forall X.B \\ 12. & \forall X.\mathbf{T} = \mathbf{T} \end{array}\right\} Th^2_{\times T}$$

$$\textbf{split} \quad \forall X.A \times B = \forall X.\forall Y.A \times (B[Y/X])$$

$$\Big\} -10, 11 = Th^{ML}$$

A, B, C stand for any type, while $\mathbf{T}$ is a constant for the terminal type `unit`.
Axiom **swap** in Th^1 is derivable in $Th^1_{\times T}$ via 1 and 3.

Table 1. Theories of isomorphisms for some type lambda calculi.

These results have found their applications in the area of Functional Programming, where they provide a means to search functions by type (see [18,19, 17,20,9,8,10]) and to match modules by specifications [2]. Also, they are used in proof assistant to find proofs in libraries up to irrelevant syntactical details [6].

Linear isomorphisms of types Recently, some weaker variants of isomorphism of types showed to be of practical interest, linear isomorphism of types in particular (e.g. , for library search in databases), which correspond to the isomorphism of objects in Symmetric Monoidal Categories, and can be also described as the

[1] Provided X is free for Y in A, and $Y \notin FTV(A)$
[2] Provided $X \notin FTV(A)$

isomorphism of types in the system of lambda calculus which corresponds to intuitionistic multiplicative linear logic. (A description of this system can be found in [22]).

In [22] it was shown that the axiom system consisting of the axioms 1, 2, 3, 5, and 7 defines an equivalence relation on types that coincides with the relation of linear isomorphism of types, and in [1] a very efficient algorithm for deciding equality of linear isomorphisms is provided; also, [12] provides a model theoretic proof of the result in [22].

In this paper, instead of studying linear isomorphisms of intuitionistic formulae, as is done when considering directly linear λ-terms, we focus on linear isomorphisms of types *inside* the multiplicative fragment (MLL) of Linear Logic [13], which is the natural settings for investigating the effect of linearity. We should stress that isomorphisms in MLL are not the same as in linear lambda calculus: MLL is a richer system, allowing formulae, like $A^\perp \otimes A$, and proofs that have no correspondence in linear lambda calculus, so we are investigating a finer world. But a particularly nice result of this change of point of view is that the axioms for linear isomorphisms are reduced to a remarkably simple form, namely, to associativity and commutativity of the logical connectives tensor and par of MLL, plus the obvious axioms for the identities.

For example, if we interpret implication $A \to B$ in linear lambda calculus as linear implication $A \multimap B$ in Linear Logic, which in turn is equivalent to $A^\perp \parr B$, the isomorphisms $\mathbf{T} \to A = A$ becomes the simple identity isomorphism $\bot \parr A = A$. Similarly, currying $((A \times B) \to C = A \to (B \to C))$ becomes just associativity of the par connective $(A^\perp \parr B^\perp) \parr C = A^\perp (\parr B^\perp \parr C)$ and Swap $(A \to (B \to C) = B \to (A \to C))$ becomes just $A^\perp (\parr B^\perp \parr C) = B^\perp (\parr A^\perp \parr C)$, which is a consequence of associativity and commutativity of par.

Formally, isomorphisms of formulae in MLL is defined as follows:

Definition 1 (Linear isomorphism) *Two formulae A and B are* isomorphic *iff*

- *A and B are linearly equivalent, i.e. $\vdash A^\perp, B$ and $\vdash B^\perp, A$*
- *there exists proofs of $\vdash A^\perp, B$ and $\vdash B^\perp, A$ that reduce, when composed by a cut rule to obtain a proof of $\vdash A^\perp, A$ (resp. $\vdash B^\perp, B$)), to the expansion of the axiom $\vdash A^\perp, A$ (resp $\vdash B^\perp, B$) after cut elimination*[3].

We show in this paper that two formulae A and B are linearly isomorphic if and only if $AC(\otimes, \parr) \vdash A = B$ in the case of MLL without units, and $ACI(\otimes, \parr) \vdash A = B$ in the case of MLL with units, where $AC(\otimes, \parr)$ and $ACI(\otimes, \parr)$ are defined in the following way:

[3] That is to say, to the proof of $\vdash A^\perp, A$ obtained by allowing only atomic axioms.

Definition 2 (*$AC(\otimes, \parr)$*) *Let $AC(\otimes, \parr)$ denote the set constituted by the four following equations:*

$$X \otimes Y = Y \otimes X \qquad (X \parr Y) \parr Z = X \parr (Y \parr Z)$$
$$X \parr Y = Y \parr X \qquad (X \otimes Y) \otimes Z = X \otimes (Y \otimes Z)$$

$AC(\otimes, \parr) \vdash A = B$ means that $A = B$ belongs to the equational theory generated by $AC(\otimes, \parr)$ over the set of linear logic formulae; in other words, it means that the formulae are equal modulo associativity and commutativity of $\parr$ and $\otimes$.

Definition 3 (*$ACI(\otimes, \parr)$*) *Let $ACI(\otimes, \parr)$ denote the set constituted by the equations of $AC(\otimes, \parr)$ plus:*

$$X \otimes 1 = X \qquad X \parr \bot = X$$

$ACI(\otimes, \parr) \vdash A = B$ means that $A = B$ belongs to the equational theory generated by $ACI(\otimes, \parr)$.

As always, in the investigation of theories of isomorphic objects, the soundness part is easy to prove:

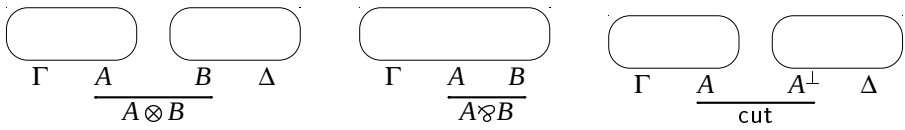
Theorem 4 (Isos soundness). *If $ACI(\otimes, \parr) \vdash A = B$, then A and B are linearly isomorphic*

Proof. By exhibiting the simple nets for the axioms and showing context closure.

All the difficulty really lies in the proof of the other implication, completeness: the rest of the article focuses on its proof. To do that, we use in an essential way the proof-nets of Girard.

Let's now recall some preliminary definitions from linear logic (but we refer to [13] for a complete introduction):

Definition 5 (proof nets) *A proof net is a structure inductively obtained starting from* axiom links $\dfrac{}{A \quad A^\bot}\,\mathsf{ax}$ *via the three construction rules*



Definition 6 (simple nets) *A proof net is simple if it contains only atomic axiom links (i.e. axiom links involving only atomic formulae).*

It is quite simple to show that

Proposition 1 (η-expansion of proof nets). *For any (possibly not simple) proof net S, there is a simple proof net with the same conclusions, which we call $\eta(S)$.*

Proof. This is done by iterating a simple procedure of $\eta - expansion$ of non atomic axiom links, which can be replaced by two axiom links plus one par and one times link.

This means that, as far as provability is concerned, one can restrict our attention to simple nets. We will show that also as far as isomorphism is concerned we can make this assumption.

We first characterize isomorphic formulae in MLL without units, by reducing isomorphism to the existence of two particular proof-nets called *simple bipartite proof-nets*. Then we show that we can restrict to *non-ambiguous formulae* (that is, formulae in which atoms occurs at most once positive and at most once negative). This characterization in term of nets allows to prove completeness of $AC(\otimes, \parr)$ for MLL (without units) by a simple induction on the size of the proof-net.

In the presence of units, we first simplify the formulae by removing all the nodes of the shape $\bot \parr A$ and $1 \otimes A$. Then we remark that isomorphisms for simplified formulae are very similar to the case without units. By showing a remarkable property of proof-nets for simplified formulae, we can indeed reduce the completeness proof to the case without units.

The paper is organized as follows: reduction to simple bipartite proof-nets and non-ambiguous formulae will be detailed in sections 2 and 3. Then we will show the final result in section 4 before extending it to the case with units in section 5.

2 Reduction to Simple Bipartite Proof Nets

First of all, we formalize the reduction to simple nets hinted at in the introduction.

Definition 7 (tree of a formula, identity simple net) *A cut-free simple proof net S proving A is actually composed of the* tree *of A, (named $T(A)$), and a set of axiom links over atomic formulae. We call* identity simple net *of A the simple cut-free proof net obtained by a full η-expansion of the (generally not simple) net $\overline{A \qquad A^\bot}$. This net is made up of $T(A)$, $T(A^\bot)$ and a set of axiom links that connect atoms in $T(A)$ with atoms in $T(A^\bot)$.*
Notice that, in simple nets, the identity axiom for A is interpreted by the identity simple net of A.

A first remark, which is important for a simple treatment of linear isomorphisms, is that we can focus, w.l.o.g., on witnesses of isomorphisms which are simple proof nets.

Lemma 1 (Simple vs. non-simple nets). *If a (non-simple) net S reduces via cut-elimination to S', then the simple net $\eta(S)$ reduces to $\eta(S')$.*

Proof. It is sufficient to show that if $S \triangleright_1 S'$ (one step reduction), then $\eta(S) \triangleright^* \eta(S')$ (arbitrary long reduction). If the redex R reduced in S does not contain any axiom link, then exactly the same redex appears in $\eta(S)$ that can therefore be reduced in one step to $\eta(S')$. Otherwise R contains an axiom link, that may be non atomic (if the axiom link is atomic, then the considered redex is exactly the same in S and $\eta(S)$ and the property is obvious). If F is non atomic and $\underline{F \quad F^\perp}$ is the cut link of R, let n be the number of atoms of F (counted with multiplicity). Then $n-1$ is the number of connectives in F, and $\eta(S)$ can be reduced to $\eta(S')$ in $2n-1$ steps ($n-1$ steps to propagate the cut link to atomic formulae, and n steps to reduce every atomic axiom link produced this way).

Theorem 8 (Reduction to simple proof nets). *Two formulae A and B are* isomorphic *iff there are two simple nets S with conclusions $A^\perp, B$ and S' with conclusions $B^\perp, A$ that when composed using a cut rule over B (resp. A) yield after cut elimination the identity simple net of A (resp. B).*

Proof. The 'if' direction is trivial, since a proof net represents a proof and cut elimination in proof nets correspond to cut elimination over proofs.
For the 'only if' direction, take the two proofs giving the isomorphism and build the associated proof nets S and S'. These nets have as conclusions $A^\perp, B$ (resp. $B^\perp, A$), and we know that after composing them via cut over B (resp. A) and performing cut elimination, one obtains the axiom net of A (resp. B). Now take the full η-expansions of S and S' as the required simple nets: by lemma 1, they reduce by composition over B (resp. A) to the identity simple net of A (resp. B).

We will show now that if two formulae are isomorphic then the isomorphism can be given by means of proof nets whose structure is particularly simple.

Definition 9 (bipartite simple proof-nets) *A cut-free simple proof net is* bipartite *if it has exactly two conclusions A and B, and it consists of $T(A)$, $T(B)$ and a set of axiom links connecting atoms of A to atoms of B, but not atoms of A between them or atoms of B between them (an example is shown in figure 1).*

Lemma 2 (cuts and trees). *Let S be a simple net (not a proof net) without conclusions built out of just $T(A)$ and $T(A^\perp)$, with no axiom link, and the cut $\underline{A \quad A^\perp}$. Then cut-elimination on S yields as a result just a set of atomic cut links $\underline{p_i \quad p_i^\perp}$ between atoms of A and atoms of $A^\perp$.*

Proof. This is a simple induction on the size of the net.

Theorem 10 (Isomorphisms are bipartite). *Let S be a cut-free simple proof net with conclusions $A^\perp$ and B, and S' be a cut-free simple proof net with conclusions $B^\perp$ and A. If their composition by cut gives respectively the identity simple net of A and B, then S and S' are bipartite.*

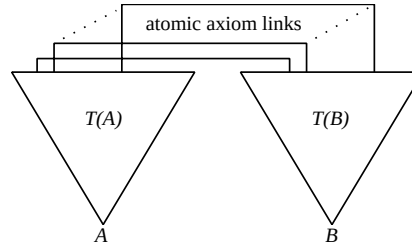


Fig. 1. The shape of a bipartite proof net.

Proof. We actually show the contrapositive: if S or S' is not bipartite, then their composition by cut is not bipartite, hence is not an identity proof net. By symmetry, we can assume w.l.o.g. that S is not bipartite, and contains an axiom link between two atoms of $A^\perp$. We claim that the composition of S and S' by cut over B is not bipartite.
Since S and S' are cut-free, their composition only contains a single cut link (between B and $B^\perp$). Since S and S' are simple, every axiom link of their composition is atomic. Hence every (atomic) axiom link of the net $\underline{S \quad S'}$ that is reduced by cut elimination is connected to an atom of B or $B^\perp$. In particular, the axiom link of S only connected to atoms of A is *not reduced*. This proves that cut elimination in $\underline{S \quad S'}$ does not lead to a bipartite net.
As a consequence, the theorem holds.

3 Reduction to Non-ambiguous Formulae

To prove the correctness theorem, we first show that one can restrict our study to non-ambiguous formulae, i.e. to formulae where each atom appears only once positive and one negated.

Definition 11 (non-ambiguous formulae) *We say that a formula A is* non-ambiguous *if each atom in A occurs at most once positive and at most once negative. For example, $A \otimes B$ and $A \otimes A^\perp$ are non-ambiguous, while $A \otimes A$ is not.*

In the following, we will call *substitution* the usual operation $[G_1/A_1, ...G_n/A_n]$ of replacement of the propositional atoms A_i of a formula by the formulae G_i. A substitution will be denoted by greek letter $\sigma, \tau, ...$, and we will also consider substitutions extended to full proof nets, i.e. if R is a proof net, $\sigma(R)$ will be the proof net obtained from R by replacing all formulae F_j appearing in it by $\sigma(F_j)$.

But we will also need a weaker notion, *renaming*, that may replace *different occurrences* of the *same* atom by different formulae in a net.

Definition 12 (renaming) *An application α from the set of* occurrences *of atoms in a proof net R to a set of atoms is a* renaming *if $\alpha(R)$ (the net obtained by substitution of each occurrence p of an atom of R by $\alpha(p)$) is a correct proof net.*

Note that if R is simple, the definition of α only on the occurrences of atoms in *axiom links* is sufficient to define α on every occurrence in R. If R is simple and bipartite, then the definition of α only on the occurrences of atoms in *one conclusion* of R is sufficient to define α on every occurrence in R.
Note also that, if the conclusions of R are ambiguous formulae, then two different occurrences of the same atom can be renamed differently, unlike what happens in the case of substitutions.

As expected, a non-ambiguous formula can only be isomorphic to another non-ambiguous formula.

Lemma 3 (non-ambiguous isomorphic formulae). *Let A and B be isomorphic formulae, and α a renaming such that $\alpha(A)$ is non-ambiguous, then $\alpha(B^\perp)$ is non-ambiguous.*

Proof. If A and B are isomorphic formulae, then (by theorem 10) there exist a bipartite proof net with conclusions A, $B^\perp$. Since α is a renaming, there also exists a *bipartite proof net with conclusions* $\alpha(A)$, $\alpha(B^\perp)$. Then, $\alpha(A)$ and $\alpha(B^\perp)$ have exactly the same atoms. And since $\alpha(A)$ is non-ambiguous, $\alpha(B^\perp)$ is also non-ambiguous.

We now prove that isomorphism is invariant by renaming.

Theorem 13 (renaming preserve isomorphisms). *If A and B are linearly isomorphic, let R and R' be the associated simple proof nets (with conclusions $A^\perp$, B and $B^\perp$, A respectively). If α is a renaming of (the occurrences of) the atoms of R, then there exists α', a renaming of the atoms of R' such that $\alpha'(A)$ and $\alpha(B)$ are isomorphic, i.e.:*

- *$\alpha'(R')$ is a correct proof net.*
- *$\alpha(A^\perp) \equiv (\alpha'(A))^\perp$ and $\alpha'(B^\perp) \equiv (\alpha(B))^\perp$*
- *The composition of $\alpha(R)$ and $\alpha'(R')$ by cut over $\alpha(B)$ (resp. $\alpha'(A)$) gives the identity simple net of $\alpha'(A)$ (resp. $\alpha(B)$).*

Proof. We first have to define α'. Since (by theorem 10) R' is bipartite, it is sufficient to define α' only on the occurrences of $B^\perp$, i.e. to define $\alpha'(B^\perp)$. So one can define: $\alpha'(B^\perp) \equiv (\alpha(B))^\perp$. Then the composition of $\alpha(R)$ and $\alpha'(R')$ by cut over B is a correct proof net. And since reduction of proof nets does not depend on labels, this composition reduces to an identity net with conclusions $\alpha(A^\perp)$ and $\alpha'(A)$. An easy induction (on the number of connectives of $\alpha(A)$) shows that η-reduction in this net gives an axiom link. One then has $\alpha(A^\perp) \equiv (\alpha'(A))^\perp$.
But then, the composition of $\alpha'(R')$ with $\alpha(R)$ by cut over $\alpha'(A)$ is a correct proof net, that reduces to an identity net (since reduction of proof nets does not depend on labels) that is the identity simple net of $\alpha(B)$.
Hence, $\alpha'(A)$ and $\alpha(B)$ are isomorphic.

We are finally in a position to show that we can restrict attention to non-ambiguous formulae.

Lemma 4 (ambiguous isomorphic formulae). *Let A and B be isomorphic formulae such that A is ambiguous, then there exists a substitution σ and formulae A' and B'* non-ambiguous *such that A' and B' are isomorphic formulae, and $A \equiv \sigma(A')$ and $B \equiv \sigma(B')$.*

Proof. Let R and R' be bipartite proof nets with conclusions $B^\perp$, A and $A^\perp$, B respectively associated to the isomorphism between A and B. Since it is sufficient here to define α only on occurrences of atoms of A, one can define a renaming α such that $\alpha(A)$ has only distinct atoms (i.e. no atom of $\alpha(A)$ occurs twice in $\alpha(A)$, even once positively and once negatively). In particular, $\alpha(A)$ is non-ambiguous. Then, theorem 13 gives an algorithm for defining a renaming α' such that $\alpha(A)$ and $\alpha'(B)$ are isomorphic: in particular $\alpha'(A^\perp) \equiv (\alpha(A))^\perp$ and $\alpha(B^\perp) \equiv (\alpha'(B))^\perp$.

Let $A' \equiv \alpha(A)$ and $B' \equiv \alpha'(B)$. By theorem 13, A' and B' are isomorphic and $\alpha(R)$ has conclusions A' and $B'^\perp$.

On $\alpha(R)$ one can define a renaming α^{-1} such that $\alpha^{-1}(A') \equiv A$, and hence $\alpha^{-1}(B'^\perp) \equiv B^\perp$.

Since $\alpha(R)$ is bipartite, it is equivalent to define α^{-1} on occurrences of R, or only on occurrences of atoms of A'. But since all atoms of A' are distinct, two distinct occurrences of atoms of A' correspond to distinct atoms of A'. One can then define a substitution σ on atoms of A' by: $\sigma(p) \equiv \alpha^{-1}(Occ(p))$ where $Occ(p)$ is the *single* occurrence of the atom p in A'.

Thus, $R \equiv \alpha^{-1}(\alpha(R)) \equiv \sigma(\alpha(R))$: in particular $\sigma(A') \equiv A$ and $\sigma(B'^\perp) \equiv \sigma(\alpha(B^\perp)) \equiv \alpha^{-1}(\alpha(B^\perp)) \equiv B^\perp$, so $\sigma(B') \equiv B$.

Finally, A' and B' are isomorphic non-ambiguous formulae such that $\sigma(A') \equiv A$ and $\sigma(B') \equiv B$.

Corollary 1 (reduction to non-ambiguous formulae). *The set of couples of isomorphic formulae is the set of instances (by a substitution on atoms) of couples of isomorphic* non-ambiguous formulae.

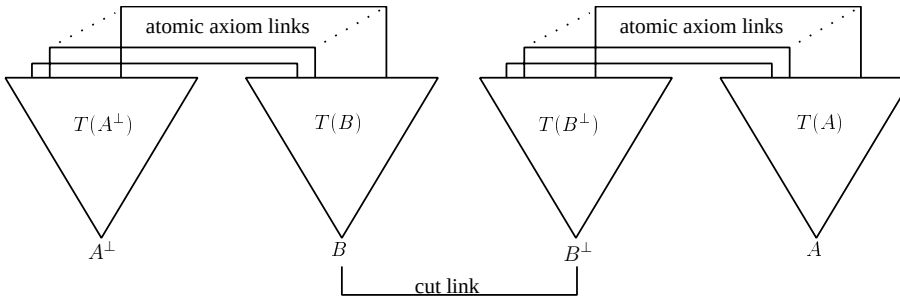
Proof. We show each inclusion separately.
Let A and B be two isomorphic formulae. By lemma 4, A and B are instances of two non-ambiguous isomorphic formulae A' and B' (eventually $A \equiv A'$ and $B \equiv B'$).

Conversely let C and D be isomorphic formulae and σ be a substitution on atoms of C (and then also on atoms of D). Let R and R' be two bipartite proof nets associated to C and D. The substitution σ defines on R a renaming α (any substitution can be seen as a renaming). Let α' be the renaming defined on R', associated to α as in theorem 13. Since $\sigma(C^\perp) \equiv \alpha(C^\perp) \equiv (\alpha'(C))^\perp$, α' is also the renaming induced by σ on R'. By theorem 13, $\alpha(C)$ and $\alpha(D)$ are isomorphic. Hence $\sigma(C)$ and $\sigma(D)$ are isomorphic.

Hence, in what follows, we can focus only on non-ambiguous formulae.

We are now able to show that for non-ambiguous formulae the very existence of bipartite simple nets implies isomorphism.

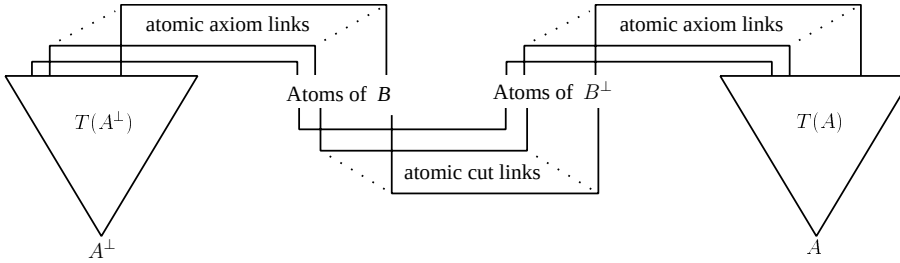
Theorem 14 (bipartite simple nets for non-ambiguous formulae). *Let S be a bipartite simple proof net over $A^\perp$ and B, and S' a bipartite simple proof net over $B^\perp$ and A. Then their composition by cut over B reduces to the identity simple net of A (resp. their composition by cut over A reduces to the identity simple net over B).*



Proof. Consider the composition of S and S' by cut over B (see figure).

By lemma 2, cut elimination in the subnet $\overline{T(B) \quad T(B^\perp)}$ gives a set of atomic cut links between atoms of B and atoms of $\overline{B^\perp}$. Since S and S' are bipartite, each such *atomic cut link*

is connected to an *atomic axiom link* between an atom of $A^\perp$ and an atom of B, and to an *atomic axiom link* between an atom of $B^\perp$ and an atom of A. Now, the net only contains atomic redex composed of cut and axiom links. The reduction of these redex gives the identity tree of A (since there is no axiom link connecting atoms of A —resp. $({}^\perp A)$ between them).



Theorems 14 and 10 have the following fundamental consequence.

Corollary 2. *Two linear non-ambiguous formulae A and B are isomorphic iff and only if there exists simple bipartite proof nets having conclusions $A^\perp, B$, and $B^\perp, A$.*

4 Completeness for Isomorphisms in *MLL*

Using the result of the above section, and the following simple lemma, we can prove our main result, i.e. completeness of $AC(\otimes, \parr)$ for MLL without constants.

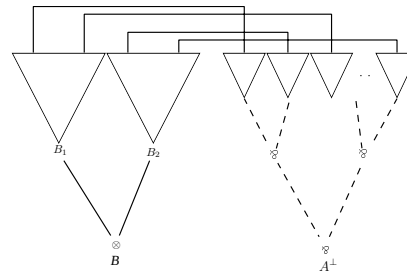
Lemma 5 (isomorphic formulae). *If A and B are linearly isomorphic, then they are both, either $\parr$-formulae, or $\otimes$-formulae.*

Proof. Actually, one $\parr$-formula and one $\otimes$-formula can not be isomorphic. If $A_1 \parr A_2$ and $A_3 \otimes A_4$ were isomorphic, then there would exist a simple bipartite proof-net with conclusion $(A_1^\perp \otimes A_2^\perp), (A_3 \otimes A_4)$, which is impossible because such a net does *not* have a splitting (terminal) tensor, since removing one of both terminal tensor links does not give two disconnected graphs (by bipartiteness). Hence two isomorphic formulae must be both, either $\parr$-formulae, or $\otimes$-formulae.

Theorem 15 (Isos completeness). *If A and B are linearly isomorphic, then: $AC(\otimes, \parr) \vdash A = B$.*

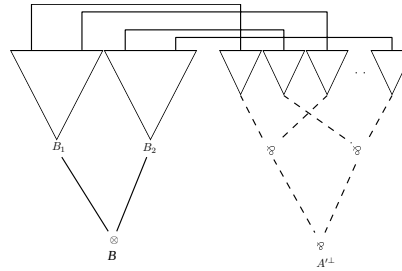
Proof. By induction on the size of the simple bipartite proof net, given by the isomorphism, with conclusions $A^\perp, B$.
If A and B are atomic, then the property is obvious.
Else the formulae $A^\perp$ and B are *both* non atomic (since the net is bipartite, they must contain the same number of atoms). Moreover, (by lemma 5) since A and B are isomorphic, one of the formulae $A^\perp$, B is a $\parr$-formula and the other one a $\otimes$-formula. One can assume, w.l.o.g., that B is a $\otimes$-formula, and $A^\perp$ a $\parr$-formula.

Now, removing all dangling $\parr$ nodes in the proof net with conclusions $A^\perp$, B gives a correct proofs net with conclusions of the form $A_1^\perp, \ldots, A_k^\perp, B_1 \otimes B_2$. If one of the formulae $A_1^\perp, \ldots, A_k^\perp$ contains a tensor node, removing it does not lead to two disconnected graphs, since (by bipartiteness) every atom of $A_1^\perp, \ldots, A_k^\perp$ is connected to the formula $B_1 \otimes B_2$.

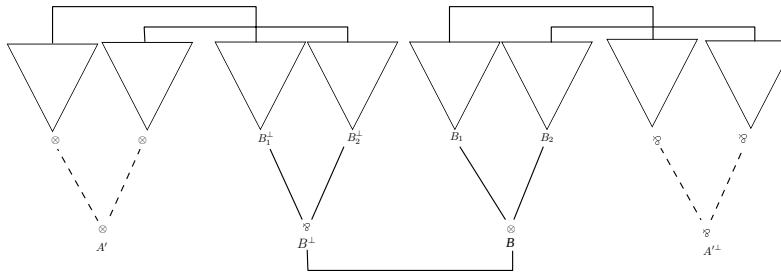


The splitting tensor node (that must exist due to Girard's correctness criterion) is necessarily the $B_1 \otimes B_2$ node. Removing it yields, due to the correctness criterion, two disconnected proof nets, which are still simple since the axiom links were not modified.

We can recover from these nets two bipartite simple proof nets (a bipartite proof net must have exactly two conclusions) by adding $\parr$-links under the A_i (because of $AC(\otimes, \parr)$, the order of the A_i does not matter). This constructs two formulae ${A'_1}^{\perp}$ (connected with B_1) and ${A'_2}^{\perp}$ (connected with B_2). The nets that we obtain contain at least one link less than the original net, and thus, are strictly smaller. It only remains to verify that B_1 and A'_1 are isomorphic, and that B_2 and A'_2 are isomorphic, so that we can apply the induction hypothesis.



To do that, we use the fact that the two initial formulae are isomorphic. If we put the initial $\parr$ and $\otimes$ back, we obtain two formulae A' and B, that are isomorphic (by theorem 4). There exists a simple bipartite proof-nets with conclusions A' and $B^{\perp}$. Since the formulae are non-ambiguous, we can extract from this proof-net to simple bipartite proof-nets; one with conclusions A'_1 and $B_1^{\perp}$, the other with conclusions A'_2 and $B_2^{\perp}$. Hence, by theorem 14, A'_1 and B_1 are isomorphic, and A'_2 and B_2 are isomorphic.



By induction hypothesis, $AC(\otimes, \parr) \vdash A'_1 = B_1$ and $AC(\otimes, \parr) \vdash A'_2 = B_2$, and thus $AC(\otimes, \parr) \vdash A' = B$. We can conclude using the fact that $AC(\otimes, \parr) \vdash A = A'$.

5 Handling the Units

We have shown above soundness and completeness result for the theory of isomorphisms given in the introduction w.r.t. *provable* isomorphisms in *MLL*. This

essentially corresponds to isomorphisms in all *-autonomous categories, which is a superset of all Symmetric Monoidal Closed Categories (SMCC's) *without units*. Nevertheless, if we want to get an interesting result also in terms of models, and handle then also SMCC's in their full form, we need to be able to add units to our treatment.

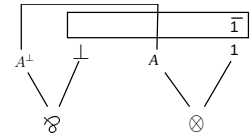
5.1 Expansions of Axioms with Units: Identity Simple Nets Revisited

In the presence of the units **1** and especially $\bot$, proof nets in general get more involved, as $\bot$ forces the introduction of the notion of *box* for which we refer the interested reader to [13], where a detailed explanation is presented.

For our purpose, it will suffice to recall here the proof-net formation rules for the units:

$$\frac{}{\mathbf{1}} \qquad \frac{\boxed{\vdots\ \Gamma}}{\Gamma \quad \bot}$$

Now, the expansion of an axiom can contain boxes, if the axiom formula involves units; for example, the axiom $\vdash (A\otimes\mathbf{1})^{\bot}, (A\otimes\mathbf{1})$ gets fully η-expanded into:



5.2 Reduction of Isomorphisms to Simplified Formulae

First notice that a formula of the shape $1\otimes A$ is always isomorphic to A, and $\bot\parr A$ is isomorphic to A.

Definition 16 (simplified formulae) *A formula is called* simplified *if it has no sub-formula of the shape* $1\otimes A$ *or* $\bot\parr A$ *(where* A *is any formula). To each formula* A*, we associate the formula* $s(A)$ *obtained by normalizing* A *w.r.t the following canonical (confluent and strongly normalizing) rewriting system:*

$$1\otimes A \to A \qquad A\otimes 1 \to A \qquad \bot\parr A \to A \qquad A\parr \bot \to A$$

We can restrict our attention to simplified formulae using the following theorem:

Theorem 17. *Two formulae* A *and* B *are linearly isomorphic if and only if* $s(A)$ *and* $s(B)$ *are linearly isomorphic.*

Proof. We first show that $1\otimes A$ is isomorphic to A, and $\bot\parr A$ is isomorphic to A, and conclude using the fact that linear isomorphism is a congruence.

5.3 Completeness with Units

For simplified formulae, the proof of completeness is very similar than in the case without units. We just have to extent the definition of bipartite simple proof-nets:

Definition 18 (bipartite simple proof-nets) *A cut-free simple proof-net is* bipartite *if it has exactly two conclusions A and B, and it consists of*

- $T(A)$, $T(B)$
- *a set of axiom links connecting atoms of A to atoms of B, but not atoms of A between them or atoms of B between them*
- *and boxes with at least one conclusion in $T(A)$ and at least one conclusion in $T(B)$.*

Theorem 19. *Let A and B be two simplified formulae. Let S be a cut-free simple proof net with conclusions $A^\perp$ and B, and S' be a cut-free proof net with conclusions $B^\perp$ and A. All the boxes in S and S' are boxes containing only the constant* 1*.*

Proof. In this proof, let n_1^X and $n_\perp^X$ denote respectively the number of 1 and the number of $\perp$ in the proof-net X.

- The case where A or B is a constant is obvious.
- Otherwise, all the occurrences of the constant 1 in the two proof-nets are one of the two sub-terms of a $\parr$-link. The only way to have these two sub-terms connected (which is necessary to make a proof-net) is to put the 1 in a box. Each of these boxes correspond to a distinct $\perp$. From there we deduce that $n_\perp^S \geq n_1^S$, and $n_\perp^{S'} \geq n_1^{S'}$.
- But $n_\perp^S = n_1^{S'}$, $n_\perp^{S'} = n_1^S$. Hence $n_\perp^S = n_1^S$. Thus there is no box with somewhat else than an 1 in S. Idem for S'.

It is easy to show that boxes of that kind behave like axiom links for cut-elimination. It suffices to remark that the only possible case of cut-elimination involving boxes in such a proof-net is the following one:

that reduces to

Thus units can be viewed exactly as atoms in this case, and we can proceed precisely as in the case without units to prove that

Theorem 20. *If A and B are linearly isomorphic, then $AC(\otimes, \parr) \vdash s(A) = s(B)$.*

Hence

Theorem 21 (Isos completeness with units). *If A and B are linearly isomorphic, then $ACI(\otimes, \parr) \vdash A = B$.*

6 Conclusions

We have shown that in MLL the only isomorphisms of formulae are given by the most intuitive axioms, namely associativity and commutativity, only. Besides the interest of the result on its own, this gives a very elegant symmetrical interpretation of linear isomorphism in linear lambda calculus, and provides a justification of the fact, observed several times in the past, that currying in functional programming correspond to "a sort of" associativity (this happens in the implementation of abstract machines, as well as in the coding of lambda terms into Berry's CDS [3]). Our result confirms once more that Linear Logic is a looking glass through which fundamental properties of functional computation appear symetrized and simplified. It should also be remarked that the axiom links from Linear Logic play a similar role to the formula links originally introduced by Lambek in his study of SMC objects [14], and that proof nets were really the missing tool to understand linearity. In proving the result, we used essentially the topological properties of the proof nets of linear logic, which simplified enormously our task (for example, the reduction to non ambiguous formulae when working directly with lambda terms is far more complex than here, where the axiom links allows us to give an elegant proof).

References

[1] A. Andreev and S. Soloviev. A deciding algorithm for linear isomorphism of types with complexity $o(nlog^2(n))$. In E. Moggi and G. Rossolini, editors, *Category Theory and Computer Science*, number 1290 in LNCS, pages 197–210, 1997.

[2] M.-V. Aponte and R. Di Cosmo. Type isomorphisms for module signatures. In *Programming Languages Implementation and Logic Programming (PLILP)*, volume 1140 of *Lecture Notes in Computer Science*, pages 334–346. Springer-Verlag, 1996.

[3] G. Berry and P.-L. Curien. Theory and practice of sequential algorithms: the kernel of the applicative language CDS. In M. Nivat and J. Reynolds, editors, *Algebraic methods in semantics*, pages 35–87. Cambridge University Press, 1985.

[4] K. Bruce, R. Di Cosmo, and G. Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.

[5] K. Bruce and G. Longo. Provable isomorphisms and domain equations in models of typed languages. *ACM Symposium on Theory of Computing (STOC 85)*, May 1985.

[6] D. Delahaye, R. Di Cosmo, and B. Werner. Recherche dans une bibliothèque de preuves Coq en utilisant le type et modulo isomorphismes. In *PRC/GDR de programmation, Pôle Preuves et Spécifications Algébriques*, November 1997.

[7] R. Di Cosmo. Invertibility of terms and valid isomorphisms. a proof theoretic study on second order λ-calculus with surjective pairing and terminal object. Technical Report 91-10, LIENS - Ecole Normale Supérieure, 1991.

[8] R. Di Cosmo. Type isomorphisms in a type assignment framework. In *19th Ann. ACM Symp. on Principles of Programming Languages (POPL)*, pages 200–210. ACM, 1992.

[9] R. Di Cosmo. Deciding type isomorphisms in a type assignment framework. *Journal of Functional Programming*, 3(3):485–525, 1993. Special Issue on ML.

[10] R. Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Birkhauser, 1995. ISBN-0-8176-3763-X.

[11] R. Di Cosmo and G. Longo. Constuctively equivalent propositions and isomorphisms of objects (or terms as natural transformations). In Moschovakis, editor, *Logic from Computer Science*, volume 21 of *Mathematical Sciences Research Institute Publications*, pages 73–94. Springer Verlag, Berkeley, 1991.

[12] K. Dosen and Z. Petric. Isomorphic objects in symmetric monoidal closed categories. Technical Report 95-49-R, IRIT, 1995.

[13] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50, 1987.

[14] S. Mac Lane and G. M. Kelly. Coherence in Closed Categories. *Journal of Pure and Applied Algebra*, 1(1):97–140, 1971.

[15] C. F. Martin. Axiomatic bases for equational theories of natural numbers. *Notices of the Am. Math. Soc.*, 19(7):778, 1972.

[16] L. Meertens and A. Siebes. Universal type isomorphisms in cartesian closed categories. Centrum voor Wiskunde en Informatica, Amsterdam, the Netherlands. E-mail: lambert,arno@cwi.nl, 1990.

[17] M. Rittri. Retrieving library identifiers by equational matching of types in 10th Int. Conf. on Automated Deduction. *Lecture Notes in Computer Science*, 449, July 1990.

[18] M. Rittri. *Searching program libraries by type and proving compiler correctness by bisimulation*. PhD thesis, University of Göteborg, Göteborg, Sweden, 1990.

[19] M. Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.

[20] C. Runciman and I. Toyn. Retrieving re-usable software components by polymorphic type. *Journal of Functional Programming*, 1(2):191–211, 1991.

[21] S. V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400, 1983.

[22] S. V. Soloviev. A complete axiom system for isomorphism of types in closed categories. In A. Voronkov, editor, *Logic Programming and Automated Reasoning, 4th International Conference*, volume 698 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 360–371, St. Petersburg, Russia, 1993. Springer-Verlag.

Choice Logic Programs and Nash Equilibria in Strategic Games

Marina De Vos*[1] and Dirk Vermeir[1]

Dept. of Computer Science
Free University of Brussels, VUB
Pleinlaan 2, Brussels 1050, Belgium
Tel: +32 2 6293308
Fax: +32 2 6293525
{marinadv,dvermeir}@tinf1.vub.ac.be
http://tinf1.vub.ac.be

Abstract. We define choice logic programs as negation-free datalog programs that allow rules to have exclusive-only disjunctions in the head. We show that choice programs are equivalent to semi-negative datalog programs, at least as far as stable models are concerned. We also discuss an application where strategic games can be naturally formulated as choice programs; it turns out that the stable models of such programs capture exactly the set of Nash equilibria.
Keywords: nondeterminism, choice, logic programs, stable model semantics, game theory

1 Introduction

Stable model semantics[2] can be regarded as introducing nondeterminism into logic programs, as has been convincingly argued in [11,9]. E.g. a program such as

$$p \leftarrow \neg q$$
$$q \leftarrow \neg p$$

has no (unique) total well-founded model but it has two total stable models, namely $\{p, \neg q\}$ and $\{\neg p, q\}$, representing a *choice* between p and q. This nondeterminism may not show up in the actual models, as in the program

$$p \leftarrow \neg q$$
$$q \leftarrow \neg p$$
$$p \leftarrow \neg p$$

where only the choice $\{p, \neg q\}$ turns out to be acceptable (the alternative leading to a contradiction).

In this paper, we simplify matters by providing for explicit choice sets in the head of a rule. Using $p \oplus q$ to denote a choice between p and q, the first example above can

* Wishes to thank the FWO for their support.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 266–276, 1999.

be rewritten as[1].

$$p \oplus q \leftarrow$$

Intuitively, $\oplus$ is interpreted as "exclusive or", i.e. either p or q, but not both, should be accepted in the above program.

It turns out that such *choice programs*, which do not use negation in the body, can meaningfully simulate arbitrary semi-negative logic programs, at least as far as their (total) stable model semantics are concerned. Since also the converse holds, we can conclude that, in a sense, choice is equivalent to negation.

Providing explicit choice as the conclusion of a rule allows for the natural expression of several interesting problems. In this paper, we show e.g. that strategic games[6] can be conveniently represented using choice programs. Moreover, the stable models of such a program characterize exactly the pure Nash equilibria of the game.

2 Choice Logic Programs

In this paper, we identify a program with its grounded version, i.e. the set of all ground instances of its clauses. This keeps the program finite as we do not allow function symbols (i.e. we stick to datalog).

Definition 1. *A **choice logic program** is a finite set of rules of the form $A \leftarrow B$ where A and B are finite sets of atoms.*

Intuitively, atoms in A are assumed to be xor'ed together while B is read as a conjunction. In examples, we often use $\oplus$ to denote exclusive or, while "," is used to denote conjunction.

Example 1 (Prisoner's Dilemma). The following simple choice logic program models the well-known prisoner's dilemma where d_i means "player i does not confess" and c_i stands for "player i confesses".

$$\begin{aligned} d_1 \oplus c_1 &\leftarrow \\ d_2 \oplus c_2 &\leftarrow \\ c_1 &\leftarrow d_2 \\ c_1 &\leftarrow c_2 \\ c_2 &\leftarrow d_1 \\ c_2 &\leftarrow c_1 \end{aligned}$$

The semantics of choice logic programs can be defined very simply.

Definition 2. *Let P be an choice logic program. The **Herbrand base** of P, denoted $\mathcal{B}_P$, is the set of all atoms occurring in the rules of P. An **interpretation** is any subset of $\mathcal{B}_P$. An interpretation I is a **model** of P if for every rule $A \leftarrow B$, $B \subseteq I$ implies that $I \cap A$ is a singleton. A model of P which is minimal (according to set inclusion) is called **stable**.*

Example 2 (Graph 3-colorability). Given the graph depicted in Fig. 2 assign each node one of three-colors such that no two adjacent nodes have the same color.

[1] Also the second example can be turned into a negation-free "choice" program, see theorem 2 below.

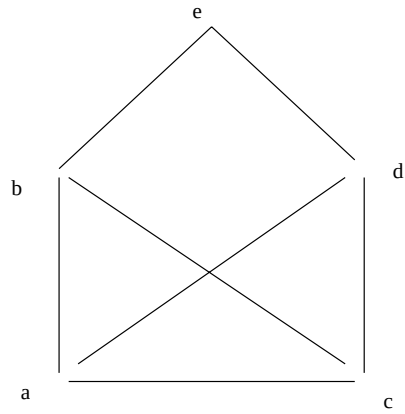


Fig. 1. Example graph which is 3-colorable.

This problem is know as graph 3-colorability and can be easily transformed in the following choice program:

$$
\begin{aligned}
color(N,b) \oplus color(N,y) \oplus color(N,r) &\leftarrow node(N)\\
&\leftarrow edge(N,J), color(N,C), color(J,C)\\
node(a) &\leftarrow\\
node(b) &\leftarrow\\
node(c) &\leftarrow\\
node(d) &\leftarrow\\
node(e) &\leftarrow\\
edge(a,b) &\leftarrow\\
edge(a,c) &\leftarrow\\
edge(a,d) &\leftarrow\\
edge(b,c) &\leftarrow\\
edge(b,e) &\leftarrow\\
edge(c,d) &\leftarrow\\
edge(d,e) &\leftarrow
\end{aligned}
$$

The first rule states that every node should take one and only one of the three available colors: black (b), yellow (y) or red (r). The second demands that two adjacent nodes have different colors. All the other rules are facts describing the depicted graph.

The stable models for this program reflect the possible solutions for this graph's 3-colorability:

$$
\begin{aligned}
N_1 &= F \cup \{color(a,b), color(b,r), color(c,y), color(d,r), color(e,b)\}\\
N_2 &= F \cup \{color(a,b), color(b,r), color(c,y), color(d,r), color(e,y)\}\\
N_3 &= F \cup \{color(a,b), color(b,y), color(c,y), color(d,y), color(e,b)\}\\
N_4 &= F \cup \{color(a,b), color(b,y), color(c,r), color(d,y), color(e,r)\}
\end{aligned}
$$

$$
\begin{aligned}
N_5 &= F \cup \{color(a,r), color(b,y), color(c,b), color(d,y), color(e,b)\} \\
N_6 &= F \cup \{color(a,r), color(b,b), color(c,y), color(d,b), color(e,y)\} \\
N_7 &= F \cup \{color(a,r), color(b,b), color(c,y), color(d,b), color(e,r)\} \\
N_8 &= F \cup \{color(a,r), color(b,y), color(c,b), color(d,y), color(e,r)\} \\
N_9 &= F \cup \{color(a,y), color(b,r), color(c,b), color(d,r), color(e,b)\} \\
N_{10} &= F \cup \{color(a,y), color(b,r), color(c,b), color(d,r), color(e,b)\} \\
N_{11} &= F \cup \{color(a,y), color(b,b), color(c,r), color(d,b), color(e,r)\} \\
N_{12} &= F \cup \{color(a,y), color(b,b), color(c,r), color(d,b), color(e,y)\}
\end{aligned}
$$

where F stands for the sets of facts from the program.

It turns out that choice logic programs can simulate semi-negative datalog programs, using the following transformation, which resembles the one used in [10] for the transformation of general disjunctive programs into negation-free disjunctive programs.

Definition 3. *Let P be a semi-negative logic program. The corresponding choice logic program $P_{\oplus}$ can be obtained from P by replacing each rule $r : a \leftarrow B, \neg C$ from P with $B \cup C \subseteq \mathcal{B}_P$ and $C \neq \emptyset$, by*

$$
\begin{aligned}
a_r \oplus K_C &\leftarrow B \quad (r'_1) \\
a &\leftarrow a_r \quad (r'_2) \\
\forall c \in C \ \cdot K_C &\leftarrow c \quad (r'_3)
\end{aligned}
$$

where a_r and K_C are new atoms that are uniquely associated with the rule r.

Intuitively, K_C is an "epistemic" atom which stands for "the (non-exclusive) disjunction of atoms from C is believed". If the positive part of a rule in the original program P is true, $P_{\oplus}$ will choose (rules r'_1) between accepting the conclusion and K_C where C is the negative part of the body; the latter preventing rule application. Each conclusion is tagged with the corresponding rule (r'_2), so that rules for the same conclusion can be processed independently. Finally, the truth of any member of C implies the truth of K_C (rules r'_3).

Definition 4. *Let P be a semi-negative logic program and let $P_{\oplus}$ be the corresponding choice logic program. An interpretation I for $P_{\oplus}$ is called **rational** iff:*

$$\forall K_C \in I \cdot I \cap C \neq \emptyset$$

Intuitively, a rational interpretation contains a justification for every accepted K_C.

Theorem 1. *Let P be a semi-negative datalog program. M is a rational stable model of $P_{\oplus}$ iff $M \cap \mathcal{B}_P$ is a (total) stable model of P.*

The rationality restriction is necessary to prevent K_C from being accepted without any of the elements of C being true. For positive-acyclic programs, we can get rid of this restriction.

Definition 5. *A semi-negative logic program P is called **positive-acyclic** iff there is an assignment of positive integers to each element of $\mathcal{B}_P$ such that the number of the head of any rule is greater than any of the numbers assigned to any non-negated atom appearing in the body.*

Note that, obviously, all stratified[8] programs are positive-acyclic. Still, many other "nondeterministic" programs such as

$$p \leftarrow \neg q$$
$$q \leftarrow \neg p$$

are also positive-acyclic.

Theorem 2. *Let P be a semi-negative positive-acyclic datalog program. There exists a choice logic program P_c such that M is a stable model of P_c iff $M \cap \mathcal{B}_P$ is a stable model of P.*

We illustrate the construction underlying theorem 2 on the following program.

$$p \leftarrow \neg q$$
$$p \leftarrow \neg p$$
$$q \leftarrow \neg p$$

The equivalent choice logic program is

$$p \oplus p^- \leftarrow$$
$$q \oplus q^- \leftarrow$$
$$p^- \leftarrow p, q$$
$$q^- \leftarrow p$$
$$p \leftarrow q^-$$
$$p \leftarrow p^-$$
$$q \leftarrow p^-$$

Intuitively, p stands for "there is a proof for p" while p^- stands for "there is no proof for p". The first two rules force the program to choose between these alternatives for every atom in the program. The rules concluding p^- (or q^-) are constructed in such a way that the truth of the body effectively blocks all possible proofs for p (resp. q). Note that the example has a single stable model $\{p, q^-\}$ which corresponds to the original's stable model $\{p, \neg q\}$.

Choice programs can be trivially simulated by semi-negative datalog programs.

Theorem 3. *Let $P_\oplus$ be a choice program. There exists a semi-negative datalog program P such that M is a stable model of $P_\oplus$ iff M is a stable model of P.*

3 Computing Stable Models

Stable models for choice logic programs can be computed by a simple "backtracking fixpoint" procedure. Essentially, one extends an interpretation by applying an immediate consequence operation, then makes a choice for every applicable rule[2] which is

[2] A rule $A \leftarrow B$ is applicable w.r.t. an interpretation I iff $B \subseteq I$.

not actually applied[3], backtracking if this leads to an inconsistency (i.e. the current interpretation cannot be extended to a model).

function *fix*(**set**<**atom**> N): **set**<**atom**>
{
set<**atom**> $M = \emptyset$;
repeat
  for each $(A \leftarrow B) \in P$ **do**
    if $B \subseteq M \wedge \exists a \in A \cdot a \notin M \cup N \wedge (A \setminus \{a\}) \subseteq N$ **then**
      $M = M \cup \{a\}$
until no change in M
return M
}

Fig. 2. *fix* is an auxiliary function for *BF*.

Figure 3 on page 272 presents such a fixpoint computation procedure *BF* which is called by the main program using

$$BF(fix(\emptyset),\emptyset).$$

We believe this procedure to be simpler than a similar one presented in [11] for semi-negative logic programs. *BF* uses an auxiliary function *fix* depicted in Fig. 2 on page 271. *Fix* is a variation on the immediate consequence operator: it computes the least fixpoint of this operator given a fixed set N of atoms that are considered to be false. Note that *fix* is deterministic since it only draws tentative conclusions from an applicable rule if there is but one possible choice for an atom in the head that will be true.

The main procedure *BF* in Fig. 3 takes two sets of atoms, M and N, containing the atoms that already have been determined to be *true* and *false*, respectively. Note that, because $M = fix(N)$ upon entry, there are no applicable rules in P that have but one undefined atom in the head. The procedure *BF* works by first verifying that no rules are violated w.r.t. the current M and N (see the definition of V in Fig. 3). It then computes the set C of applicable (but unapplied) rules for which a choice can be made as to the atom from the head that needs to be true in order to apply the rule. If there are no such rules, we have a model. Otherwise, the algorithm successively selects a rule r from C and a possible choice c from the head of r that would make it applied. This choice is "propagated" using *fix*, after which *BF* is called recursively using the new versions of M and N[4].

Theorem 4. *Let P be a choice logic program. Then BF(fix(∅),∅), where BF is described in Fig. 3 terminates and computes exactly the set of stable models of P.*

Note that, because of theorem 1, *BF* can be easily modified to compute the stable models of any semi-negative logic program through its equivalent choice logic program.

[3] An applicable (w.r.t. an interpretation I) rule is applied iff $A \cap I$ is a singleton.

[4] Clearly, the algorithm can be made more efficient, e.g. by memoizing more intermediate results.

```
procedure BF(M, N :set<atom>)
{
set<rule> V = {(A ← B) ∈ P | B ⊆ M ∧ (#(A ∩ M) > 1 ∨ A ⊆ N)}
if (V ≠ ∅)
        return /* because some rules are violated */
set<rule> C = {(A ← B) ∈ P | B ⊆ M ∧ #(A \ (M ∪ N)) > 1}
if (C = ∅)
        output M
else
        for each ((A ← B) ∈ C) {
                set<atom> c = A \ (M ∪ N)
                for each (a ∈ C) {
                        N = N ∪ (c \ {a})
                        BF(fix(N),N)
                        }
                }
}
```

Fig. 3. The *BF* (backtracking fixpoint) procedure for Choice Logic Programs.

	Bach	*Stravinsky*
Bach	$2, 1$	$0, 0$
Stravinsky	$0, 0$	$1, 2$

Fig. 4. Bach or Stravinsky (BoS)

4 An Application to Strategic Games

A strategic game models a situation where several agents (called players) independently make a decision about which action to take, out of a limited set of possibilities. The result of the actions is determined by the combined effect of the choices made by each player. Players have a preference for certain outcomes over others. Often, preferences are modeled indirectly using the concept of *payoff* where players are assumed to prefer outcomes where they receive a higher payoff.

Example 3 (Bach or Stravinsky). Two people wish to go out together to a music concert. They can choose for Bach in one theater or for Stravinsky in another one. Their main concern is to be together, but one person prefers Bach and the other prefers Stravinsky. If they both choose Bach then the person who preferred Bach gets a payoff of 2 and the other one a payoff of 1. If both go for Stravinsky, it is the other way around. If they pick different concerts, they both get a payoff of zero.
The game is represented in Fig. 4. One player's actions are identified with the rows and the other player's with the columns. The two numbers in the box formed by row r and column c are the players' payoffs when the row player chooses r and the column player chooses c, the first component being the payoff of the row player.

Definition 6 ([6]). *A **strategic game** is a tuple $\langle N, (A_i)_{i\in N}, (\succeq_i)_{i\in N}\rangle$ where*

- *N is a finite set of **players**;*

- *for each player $i \in N$, A_i is a nonempty set of **actions** that are available to her (we assume that $A_i \cap A_j = \emptyset$ whenever $i \neq j$) and,*
- *for each player $i \in N$, $\geq_i$ is a **preference relation** on $A = \times_{j \in N} A_j$*

*An element $\mathbf{a} \in A$ is called a **profile**. For a profile $\mathbf{a}$ we use $\mathbf{a}_i$ to denote the component of $\mathbf{a}$ in A_i. For any player $i \in N$, we define $A_{-i} = \times_{j \in N \setminus \{i\}} A_j$. Similarly, an element of A_{-i} will often be denoted as $\mathbf{a}_{-i}$. For $\mathbf{a}_{-i} \in A_{-i}$ and $a_i \in A_i$ we will abbreviate as $(\mathbf{a}_{-i}, a_i)$ the profile $\mathbf{a}' \in A$ which is such that $\mathbf{a}'_i = a_i$ and $\mathbf{a}'_j = \mathbf{a}_j$ for all $j \neq i$.*

Playing a game $\langle N, (A_i)_{i \in N}, (\geq_i)_{i \in N} \rangle$ consists of each player $i \in N$ selecting a single action from the set of actions A_i available to her. Since players are thought to be rational, it is assumed that a player will select an action that leads to a "preferred" profile. The problem, of course, is that a player needs to make a decision not knowing what the other players will choose.

The notion of Nash equilibrium shows that, in many cases, it is still possible to limit the possible outcomes (profiles) of the game.

Definition 7. *A **Nash equilibrium** of a strategic game $\langle N, (A_i)_{i \in N}, (\geq_i)_{i \in N} \rangle$ is a profile $\mathbf{a}^*$ satisfying*

$$\forall a_i \in A_i \cdot (\mathbf{a}^*_{-i}, \mathbf{a}^*_i) \geq_i (\mathbf{a}^*_{-i}, a_i)$$

Intuitively, a profile $\mathbf{a}^*$ is a Nash equilibrium if no player can unilaterally improve upon his choice. Put in another way, given the other players' actions $\mathbf{a}^*_{-i}$, $\mathbf{a}^*_i$ is the best player i can do[5].

Given a strategic game, it is natural to consider those moves that are best for player i, given the other players' choices.

Definition 8. *Let $\langle N, (A_i)_{i \in N}, (\geq_i)_{i \in N} \rangle$ be a strategic game. The **best response** function B_i for player $i \in N$ is defined by*

$$B_i(\mathbf{a}_{-i}) = \{a_i \in A_i \mid \forall a'_i \in A_i \cdot (\mathbf{a}_{-i}, a_i) \geq_i (\mathbf{a}_{-i}, a'_i)\}$$

The following definition shows how games allow an intuitive representation as choice logic programs.

Definition 9. *Let $G = \langle N, (A_i)_{i \in N}, (\geq_i)_{i \in N} \rangle$ be a strategic game. The choice logic program P_G associated with G contains the following rules:*

- *For each player i, P_G contains the rule $A_i \leftarrow$. This rule ensures that each player i chooses exactly one action from A_i.*
- *For each player i and for each profile $\mathbf{a} \in A_{-i}$, P_G contains a rule[6] $B_i(\mathbf{a}) \leftarrow \mathbf{a}$. It models the fact that a player will select a "best response", given the other players' choices.*

Essentially, P_G simply forces players to choose an action. Moreover, the action chosen should be a "best response" to the other players' actual choices.

[5] Note that the actions of the other players are not actually known to i.

[6] We abuse notation by writing $\mathbf{a}$ for the set of components of $\mathbf{a}$.

Theorem 5. *For every strategic game $G = \langle N, (A_i)_{i\in N}, (\geq_i)_{i\in N}\rangle$ there exists a choice logic program P_G such that the set of stable models of P_G coincides with the set of Nash equilibria of G.*

Example 4. Let us reconsider the Bach or Stravinsky game of example 3. This game has two Nash equilibria, namely:

$$(Bach_1, Bach_2) \text{ and } (Stravinsky_1, Stravinsky_2)$$

The corresponding choice logic program is:

$$\begin{aligned} b_1 \oplus s_1 &\leftarrow \\ b_1 \oplus s_2 &\leftarrow \\ b_1 &\leftarrow b_2 \\ s_1 &\leftarrow s_2 \\ b_2 &\leftarrow b_1 \\ s_2 &\leftarrow s_1 \end{aligned}$$

where b_i and s_i are shorthands for player i choosing respectively $Bach$ or $Stravinsky$. This program has two stable models, namely $\{s_1, s_2\}$ and $\{b_1, b_2\}$ that correspond to the Nash equilibria of the game.

Example 5. The program in example 1 is the choice logic program corresponding to the strategic game depicted in Fig. 5. Here two prisoners are interrogated in separate

	Do not confess	*Confess*
Do not confess	$3, 3$	$0, 4$
Confess	$4, 0$	$1, 1$

Fig. 5. Prisoner's Dilemma

rooms. Each one must decide whether or not to confess. Confessing implicates the other prisoner and may result in a lighter sentence, provided the other prisoner did not confess. This game has one Nash equilibrium

$$\{Confess_1, Confess_2\}$$

corresponding the single stable model of the program of example 1.

Note that the construction of P_G can be regarded as an encoding of the fact that the rationality and preferences of the players are common knowledge, as all rules interact and "cooperate" to verify atoms. This observation opens the possibility of extending the present approach to one where players may not be fully aware of each other's beliefs. This could be done, e.g. by considering a "choice" variation of "ordered logic programs"[3, 4,5].

Another interesting aspect of theorem 5 is that, in combination with theorem 4, it provides a systematic method for the computation of Nash equilibria for (finite) strategic games.

Corollary 1. *For every strategic game $G = \langle N, (A_i)_{i \in N}, (\geq_i)_{i \in N} \rangle$ there exists a semi-negative datalog program program P_G such that the set of stable models of P_G coincides with the set of Nash equilibria of G.*

5 Relationship to Other Approaches and Directions for Further Research

The logical foundations of game theory have been studied for a long time in the confines of epistemic logic, see e.g. [1] for a good overview. However, to the best of our knowledge, very little has been done on using logic programming-like formalisms to model game-theoretic concepts.

An important exception is [7] which introduces a formalism called "Independent Choice Logic" (ICL) which uses (acyclic) logic programs to deterministically model the *consequences* of choices made by agents. Since choices are external to the logic program, [7] restricts the programs further to not only be deterministic (i.e. each choice leads to a unique stable model) but also independent in the sense that literals representing alternatives may not influence each other, e.g. they may not appear in the head of rules. ICL is further extended to reconstruct much of classical game theory and other related fields.

The main difference with our approach is that we do not go outside of the realm of logic programming to recover the notion of Nash equilibria. Contrary to ICL, we *rely* on nondeterminism to represent alternatives, and on the properties of stable semantics to obtain Nash equilibria. As for the consequences of choices, these are represented in choice logic programs, much as they would be in ICL.

The present paper succeeded in recovering Nash equilibria without adding any fundamentally new features to logic programs (on the contrary, we got rid of negation in the body). However, the results are restricted to so-called "pure" equilibria where each participant must choose a single response. We would like to extend the formalism further to cover, in a similar way, also other game-theoretic notions. E.g. we are presently working on extending our approach to represent mixed equilibria (which are probability distributions over alternatives) as well. Finally, as mentioned in Sec. 4, using (an extension of) ordered logic could simplify the introduction of epistemic features into the formalism.

References

1. Pierpaolo Battigalli and Giacomo Bonanno. Recent Results on Belief, Knowledge and the Epistemic Foundations of Game Theory. Working Paper.
2. Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1081–1086, Seattle, 1998. ALP, IEEE, The MIT Press.
3. P. Geerts and D. Vermeir. Credulous and Autoepistemic Reasoning using Ordered Logic. *Proceedings of the First International Workshop on Logic Programming and Non-Monotonic Reasoning*, page 21–36, MIT Press, 1991.

4. P. Geerts and D. Vermeir and D. Nute. Ordered logic: Defeasible Reasoning for Multiple Agents. *Decision Support Systems*, **11** (1994) 157–190.
5. E. Laenens and D. Vermeir. A Fixpoint Semantics of Ordered Logic. *Journal of Logic and Computation*, **1(2)**(1990) 159-185.
6. M. J. Osborne and A. Rubinstein. *A Course in Game Theory*, MIT Press,1994.
7. David Poole. The Independent Choice Logic for Modelling Multiple Agents under Uncertainty. *Artificial Intelligence*, **94(1–2)** (1997) 7–56.
8. T. C. Przymusinski. Every Logic Program Has a Natural Stratification and an Iterated Fixed Point Model. In *Proceedings of the Eight ACM Symposium on Principles of Database Systems*, pages 11–21. Association for Computing Machinery, 1989.
9. D. Sacca. Deterministic and Non-Deterministic Stable Models. *Logic and Computation*, **5** (1997) 555–579.
10. Chiaki Sakama and Katsumi Inoue An Alternative Approach to the Semantics of Disjunctive Logic Programs and Decductive Databases. *Journal of Automated Reasoning*, **13** (1994) 145–172.
11. D. Sacca and C. Zaniolo. Stable Models and Non-Determinism for Logic Programs with Negation. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 205-218. Association for Computing Machinery, 1990.

Resolution Method for Modal Logic with Well-Founded Frames

Shigeki Hagihara and Naoki Yonezaki

Department of Computer Science,
Tokyo Institute of Technology,
Meguro-ku, Tokyo, 152-8552, Japan

Abstract. The modal logic KW with finite temporal frames (i.e. well-founded frames) can be used for verifying the properties where system termination is assumed, such as partial correctness of a system. This paper presents an unification-based proof method for the modal logic KW. In order to certify that a formula does not have a model of a well-founded frame, it is necessary to examine the fact if it has a model, then the model contains infinite transitions. It is, however, difficult to examine with unification-based proof methods. This paper introduces the concept of non-iterative frames. The satisfiability of a formula in non-iterative frames agrees with the one in well-founded frames. The size of the evidence of the fact that a formula does not have a model of non-iterative frame is finite. Based on this property, we have constructed a unification-based prover to check unsatisfiability of a formula in KW.

1 Introduction

The modal logic KW, frame of which is restricted to finite temporal structures (i.e. well-founded frame), can be used for the verification of properties where system termination is assumed, such as partial correctness of a system (e.g. " the termination of the system is always normal termination ").

Let us consider the following example of a client server system. Let the server be a beer shop and the client be its customer. The specification of the server is "the shop serves a beer for the customer's request". The specifications of the client are "first, the customer requests a beer" and "after the customer finish drinking the beer, he becomes satisfied or requests one more beer". The property we want to verify is "if the interaction between the shop and its customer terminates, the customer is satisfied".

This example can be specified by the logic KW as follows. The specification of the server *ServerSpec* is written as " $\Box(Req \rightarrow \Diamond Beer)$" and the specification of the client *ClientSpec* is "$\Diamond Req \wedge \Box(Beer \rightarrow \Diamond Satisfied \vee \Diamond Req)$", the property *Correctness* we will verify is "$\Diamond Satisfied$". $\Box$ and $\Diamond$ are modal operators expressing "always" and "sometimes". *Req*, *Beer* and *Satisfied* are propositions expressing "the customer requests the beer", "the shop serves a beer (the customer drinks a beer) " and "the customer is satisfied ". We can verify the property by proving the formula $ServerSpec \wedge ClientSpec \rightarrow Correctness$ in the logic KW.

We can also verify the property by using temporal logics with frames restricted to infinite linear discrete temporal structures (e.g. natural number frame). These logics are not so appropriate as KW for this type of specification, because we can omit the

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 277−291, 1999.

formula representing the assumption of system termination. Since the logic KW treats the finite temporal structures only, it is sufficient to write the conclusion part of the property, i.e. “the system terminates normally” in KW. On the other hand, we have to specify the formula representing assumption of system termination (the assumption part of the property), i.e. “if the system terminates”, if the infinite linear discrete temporal logics are used. For example, to describe the property *Correctness* in the example using the infinite linear discrete temporal logic, we have to write the assumption part “the customer does not request forever” like $\neg\Box\Diamond \mathit{Req} \rightarrow \Diamond \mathit{Satisfied}$, although $\Diamond \mathit{Satisfied}$ is enough for KW.

Among various methods of proving modal formula[1][2] [10][11], unification-based proof methods [12] are efficient and have ability of adaptation to the various modal logics, since the modal unification[9] absorbs a difference of the modal systems. Although such provers are adaptable to basic modal logics K,K4,D,D4,T,S4,etc. they do not deal with the modal logic KW.

Resolution methods using the translation from a modal formula into a formula of clausal normal form of predicate logic were proposed in [6] and [8]. They have advantages of making full use of proof strategies in resolution methods of predicate logic. They can adapt to the modal logics with first-order definable frames. However, they can not deal with the modal logic KW since well-foundedness is not first-order definable.

Proof methods for the modal logics with first-order undefinable frames are suggested in [7] and [3]. The method proposed in [7] uses a combination of Hilbert style reasoning and semantical reasoning. Although it can deal with the logic KW, it is not so efficient because it uses Hilbert style reasoning. On the other hand, the method proposed in [3] uses translation from a modal formula into a formula of set theory. Although it can adapt to the logic KW, it requires the process of translation.

In this paper, we present a unification-based resolution method for KW using clausal normal form of modal formula. Its special feature is that it does not treat well-founded frames directly but uses Herbrand non-iterative frames, introduced in this paper, as a basis. A refutation in the method corresponds to showing unsatisfiability in the class of the Herbrand non-iterative frames, instead of well-founded frames. The Herbrand non-iterative frame is the frame where the same transition never repeats. Intuitively, the Herbrand non-iterative frame can be considered well-founded frame since a formula has only finite number of positive occurrences of $\Diamond$ operators, which correspond to transitions and the same transition never repeats. The satisfiability of a formula in Herbrand non-iterative frames agrees with the one in well-founded frames. In order to show that it does not have a model of a well-founded frame, we have to examine the fact if a formula has a model then the model should contain infinite transitions. It is, however, difficult to examine with unification-based proof methods. The size of the evidence of the fact that a formula does not have a model of Herbrand non-iterative frame is finite. Based on this property, we have constructed a unification-based proof method of checking unsatisfiability of a formula in KW.

This paper is organized as follows. In Sect.2, we introduce the well-founded frames and the modal logic KW. In Sect.3, we define clausal normal form and give the resolution method for KW. Section 4 shows the soundness and completeness of the resolution method by using Herbrand non-iterative frames.

2 Modal Logic KW

The syntax of the modal logic KW is defined normally[4]. The axiomatic system of KW is the system obtained by adding the formula $\Box(\Box A \rightarrow A) \rightarrow \Box A$ as an axiom to the system K. Let f be a formula, $\vdash_{KW} f$ denotes that f is a theorem in KW.

A frame is a tuple $\langle W, R\rangle$ and a model is a triple $\langle W, R, V\rangle$, where W is a set of worlds, R is a binary relation on W (which is sometimes called reachability relation) and V is an assignment which gives a set of worlds to a proposition symbol.

A well-founded frame $\langle W, R\rangle$ and a well-founded model $\langle W, R, V\rangle$ satisfy transitivity and well-foundedness.

Transitivity: $\forall xyz(xRy \wedge yRz \rightarrow xRz)$

Well-foundedness: There is no infinite sequence $x_0x_1x_2\ldots$ where $x_0Rx_1, x_1Rx_2, \ldots$.

Example 1. Let N be a set of natural numbers and Q be a set of rational numbers, then $\langle N, >\rangle$ is a well-founded frame. Neither $\langle Q, >\rangle$ nor $\langle N, <\rangle$ is a well-founded frame.

$M, w \models f$ denotes that a formula f is true at a world $w \in W$ in a model $M = \langle W, R, V\rangle$. The truth condition is defined as usual. A formula f is valid (unsatisfiable) in the class of well-founded frames if and only if for every well-founded model $M = \langle W, R, V\rangle$ and for every world $w \in W$, $M, w \models f$ ($M, w \not\models f$).

The following property holds between the system KW and well-founded frames[4].

$\vdash_{KW} f$ iff f is valid in the class of well-founded frames.

3 Resolution Method

In this section, we define the clausal normal form of modal formulas and a unification-based resolution method for KW.

3.1 Labeled Formula, Clausal Normal Form

We assume that all $\neg$ operators in a formula occur in front of proposition symbols. This restriction keeps generality because the following rules can transform any formula to an equivalent formula of the form.

$f \rightarrow g \Rightarrow \neg f \vee g$, $\neg(f \wedge g) \Rightarrow \neg f \vee \neg g$, $\neg(f \vee g) \Rightarrow \neg f \wedge \neg g$, $\neg\neg f \Rightarrow f$, $\neg\Box f \Rightarrow \Diamond\neg f$, $\neg\Diamond f \Rightarrow \Box\neg f$.

Let f be a formula. The formula obtained by assigning different variable-labels to each occurrence of $\Box$ operator and different constant-labels to each occurrence of $\Diamond$ operator in f is called labeled formula, denoted by f^*. We use $\Box_x$ as the modal operator $\Box$ associated with the variable-label x, and $\Diamond_a$ as $\Diamond$ associated with the constant-label a. The formula obtained by distributing modal operators according to the following rules, and then transforming to conjunctive normal form is called clausal normal form. We write the clausal normal form of f as f^c.

$\Box_x(f \wedge g) \Rightarrow \Box_x f \wedge \Box_x g$, $\quad \Box_x(f \vee g) \Rightarrow \Box_x f \vee \Box_x g$,
$\Diamond_a(f \wedge g) \Rightarrow \Diamond_a f \wedge \Diamond_a g$, $\quad \Diamond_a(f \vee g) \Rightarrow \Diamond_a f \vee \Diamond_a g$

The clausal normal form is a formula of the form

$$\Gamma_1 \wedge \ldots \wedge \Gamma_n$$

where each Γ_i is a formula of the form

$$\alpha_{i1} L_{i1} \vee \ldots \vee \alpha_{im} L_{im}$$

each L_{ij} is a literal and each α_{ij} is a sequence of modal operators associated with labels.

Remark 1. Although the formula $\Box(f \vee g) \rightarrow (\Box f \vee \Box g)$ is not valid in the modal logic KW, the rule $\Box_x(f \vee g) \Rightarrow \Box_x f \vee \Box_x g$ is admissible because common label x is used in the labeled formula $\Box_x f \vee \Box_x g$. The labeled formulas $\Box_x(f \vee g)$ and $\Box_x f \vee \Box_x g$ are equivalent in Herbrand model defined in Sect.4.1.

3.2 Resolution Method

The resolution method is a refutation system. First, we transform a formula f to f^c. Then we apply the following resolution rules to f^c. We say f or f^c is refutable if the empty clause $\perp$ is derived from f^c.

1. $\dfrac{\alpha L \vee \Gamma \quad \beta \bar{L} \vee \Gamma'}{(\alpha\perp \vee \Gamma \vee \Gamma')^{\sigma(\alpha,\beta)}}$,
2. $\dfrac{\alpha\gamma L \vee \Gamma \quad \beta\delta L' \vee \Gamma'}{(\alpha\gamma L \vee \Gamma \vee \Gamma')^{\sigma(\alpha,\beta)}}$,
3. $\dfrac{\alpha\perp \vee \Gamma}{\Gamma}$ (there is no $\Box$ in α.),
4. $\dfrac{\alpha\gamma L \vee \Gamma}{\Gamma}$ (there is no $\Box$ in α and there are $\Diamond$s associated with the same constant-labels in α.),

where L and L', $\bar{L}$ are literals, L and $\bar{L}$ are complementary literals, α,β,γ and δ are sequences of modal operators associated with labels, $\sigma(\alpha,\beta)$ is a substitution which unifies α and β, and $(\alpha\perp \vee \Gamma \vee \Gamma')^{\sigma(\alpha,\beta)}$ and $(\alpha\gamma L \vee \Gamma \vee \Gamma')^{\sigma(\alpha,\beta)}$ are the formulas obtained by replacing modal operators in $\alpha\perp \vee \Gamma \vee \Gamma'$ and $\alpha\gamma L \vee \Gamma \vee \Gamma'$ by the substitution $\sigma(\alpha,\beta)$ respectively. Each substitution is a set of assignments from a modal operator $\Box$ associated with variable-label to a sequence of labeled modal operators $\{\Diamond,\Box\}^+$, length of which is more than 0. If the same variable-labels appear in different clauses, we deal with them as different variable-labels.

The resolution rules 1 and 3 are usual rules. The resolution rule 2 is used for replacing α by using $\sigma(\alpha,\beta)$. The resolution rule 4 is obtained from the characteristic of well-founded frames. The rules 1, 2 and 3 are resolution rules for resolution method for the modal logic K4, frame of which is restricted to transitive frames.

Example 2. A refutation of the following formula f is as follows.

$$f : \Diamond P \wedge \Box(\neg P \vee \Diamond P \vee \Diamond Q) \wedge \Box\neg Q$$

The labeled formula f^* and the clausal normal form f^c are as follows.

$$f^* : \Diamond_a P \wedge \Box_x(\neg P \vee \Diamond_b P \vee \Diamond_c Q) \wedge \Box_y \neg Q$$

$$f^c : \Diamond_a P \wedge (\Box_x \neg P \vee \Box_x \Diamond_b P \vee \Box_x \Diamond_c Q) \wedge \Box_y \neg Q$$

Figure 1 shows a refutation of f^c.

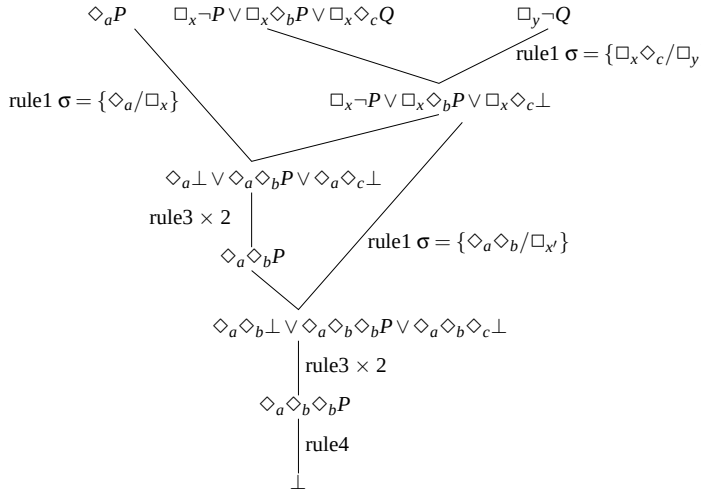


Fig. 1. A refutation of $\Diamond P \wedge \Box(\neg P \vee \Diamond P \vee \Diamond Q) \wedge \Box \neg Q$

Example 3. A refutation of the formula $g : \Diamond P \wedge \Box \Diamond Q$ is as follows. g^* and g^c are $\Diamond_a P \wedge \Box_x \Diamond_b Q$. Figure 2 shows a refutation of g^c.

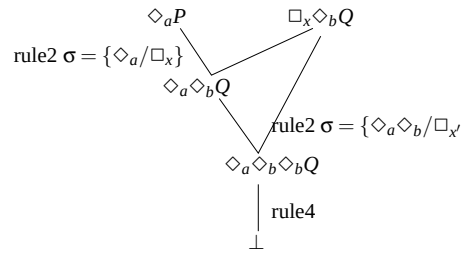


Fig. 2. A refutation of $\Diamond P \wedge \Box \Diamond Q$

4 Soundness and Completeness of Resolution Method

In this section, we show the soundness and the completeness of the resolution method. First, we propose Herbrand non-iterative model which can interpret clausal normal forms in Sect.4.1. In Sect.4.2, we define a tableau method T^*_{KW} for a labeled formula. In Sect.4.3, we show the property that the satisfiability of a formula f in well-founded frames agrees with the satisfiability of f^c in Herbrand non-iterative frames, and the property that the tableau method T^*_{KW} judges whether a formula is unsatisfiable in the class of well-founded frames. In Sect.4.4, we show the soundness of the resolution method by using Herbrand non-iterative frames. In Sect.4.5, we show the completeness by using the tableau method T^*_{KW}.

4.1 Herbrand Non-iterative Model

A frame $\langle W, R\rangle$ is a Herbrand non-iterative frame for f^* if the following conditions are satisfied:

(1) W is a set of terms which consist of the special constant symbol $\circ$ and unary function symbols corresponding to constant labels in f^*. For example, let a, b be constant labels in f^*. W is $\{\circ, a\circ, b\circ, aa\circ, ab\circ, ba\circ, bb\circ, aaa\circ, \ldots\}$. [1]

(2) R is a reachability relation on W which satisfies transitivity and non-iterativity.
Transitivity: $\forall xyz(xRy \wedge yRz \rightarrow xRz)$
Non-iterativity: For every constant label a,
$\forall x(xRax \wedge axRaax \rightarrow \bot)$
$\forall xy(xRax \wedge axRy \wedge yRay \rightarrow \bot)$

Let $\langle W, R\rangle$ be a Herbrand non-iterative frame and V be an assignment which gives a set of worlds to a proposition symbol. We say $\langle W, R, V\rangle$ is a Herbrand non-iterative model.

The truth condition of a labeled formula is defined by a Herbrand non-iterative model $HM = \langle W, R, V\rangle$ as follows:

- $HM, s \models P \Leftrightarrow s \in V(P)$,
- $HM, s \models \neg P \Leftrightarrow \neg(HM, s \models P)$,
- $HM, s \models f \wedge g \Leftrightarrow (HM, s \models f) \wedge (HM, s \models g)$,
- $HM, s \models f \vee g \Leftrightarrow (HM, s \models f) \vee (HM, s \models g)$,
- $HM, s \models \Box_x f \Leftrightarrow \neg sRx \vee (HM, x \models f)$,
- $HM, s \models \Diamond_a f \Leftrightarrow sRas \wedge (HM, as \models f)$.

We write $s \models f$ instead of $HM, s \models f$, when HM is obvious from the context.

Obviously, a labeled formula f^* and its clausal normal form f^c is equivalent in a Herbrand non-iterative model.

Example 4. The labeled formulas $\Box_x(f \vee g)$ and $\Box_x f \vee \Box_x g$ are equivalent in a Herbrand non-iterative model as follows.

$s \models \Box_x(f \vee g)$
$\Leftrightarrow \neg sRx \vee (x \models (f \vee g))$

[1] In this paper, we omit brackets. we write $a\circ, ab\circ, ay$ instead of $a(\circ), a(b(\circ)), a(y)$.

$\Leftrightarrow \neg sRx \vee (x \models f \vee x \models g)$
$\Leftrightarrow (\neg sRx \vee x \models f) \vee (s \models \neg sRx \vee x \models g)$
$\Leftrightarrow s \models \Box_x f \vee \Box_x g$

Example 5. The labeled formula $g : \Diamond_a P \wedge \Box_x(\neg P \vee \Diamond_b P)$ does not have a Herbrand non-iterative model. Because, if g is true in a world s in a Herbrand non-iterative model, then the following contradiction occurs. (Underlined formulas are used for the derivation to the next line.)

$\underline{s \models \Diamond_a P \wedge \Box_x(\neg P \vee \Diamond_b P)}$
$\Rightarrow \underline{(s \models \Diamond_a P)} \wedge \underline{(s \models \Box_x(\neg P \vee \Diamond_b P))}$
$\Rightarrow \underline{sRas} \wedge (as \models P) \wedge \underline{(\neg sRx \vee x \models (\neg P \vee \Diamond_b P))}$
$\Rightarrow sRas \wedge (as \models P) \wedge (\neg sRx \vee x \models (\neg P \vee \Diamond_b P)) \wedge \underline{(as \models (\neg P \vee \Diamond_b P))}$
$\Rightarrow sRas \wedge \underline{(as \models P)} \wedge (\neg sRx \vee x \models (\neg P \vee \Diamond_b P)) \wedge \underline{(\neg(as \models P) \vee (as \models \Diamond_b P))}$
$\Rightarrow sRas \wedge (as \models P) \wedge (\neg sRx \vee x \models (\neg P \vee \Diamond_b P)) \wedge \underline{(as \models \Diamond_b P)}$
$\Rightarrow \underline{sRas} \wedge (as \models P) \wedge (\neg sRx \vee x \models (\neg P \vee \Diamond_b P)) \wedge \underline{asRbas} \wedge (bas \models P)$
$\Rightarrow sRas \wedge (as \models P) \wedge \underline{(\neg sRx \vee x \models (\neg P \vee \Diamond_b P))} \wedge asRbas \wedge (bas \models P) \wedge \underline{sRbas}$ (transitivity)
$\Rightarrow sRas \wedge (as \models P) \wedge \underline{(bas \models (\neg P \vee \Diamond_b P))} \wedge asRbas \wedge (bas \models P) \wedge sRbas$
$\Rightarrow sRas \wedge (as \models P) \wedge \underline{(\neg(bas \models P) \vee (bas \models \Diamond_b P))} \wedge asRbas \wedge \underline{(bas \models P)} \wedge sRbas$
$\Rightarrow sRas \wedge (as \models P) \wedge \underline{(bas \models \Diamond_b P)} \wedge asRbas \wedge (bas \models P) \wedge sRbas$
$\Rightarrow sRas \wedge (as \models P) \wedge \underline{basRbbas} \wedge (bbas \models P) \wedge \underline{asRbas} \wedge (bas \models P) \wedge sRbas$
$\Rightarrow \bot$ (non-iterativity)

Original formula of g is actually unsatisfiable in the class of well-founded frames.

4.2 Tableau Method for Labeled Formula

In this section, we define the tableau method T^*_{KW} for a labeled formula.

In the tableau method T^*_{KW}, we assign the singleton of a labeled formula $\{f^*\}$ to root node, make a tree by applying the following three expansion rules to the nodes and judge the satisfiability of f in the class of well-founded frames according to whether the root node is closed or not. We say the tree with the root node assigned to $\{f^*\}$ is a tableau of f^*. A tableau is said to be closed if its root node is closed. In this paper, $FS(n)$ is an abbreviation of the set of formulas to which a node n is assigned.

– **α-rule:** $\dfrac{\Gamma \cup \{f \wedge g\}}{\Gamma \cup \{f, g\}}$,
– **β-rule:** $\dfrac{\Gamma \cup \{f \vee g\}}{\Gamma \cup \{f\} \quad \Gamma \cup \{g\}}$,
– **π-rule:** $\dfrac{\{\Box_{x_1} f_1, \ldots, \Box_{x_n} f_n, \Diamond_{a_1} g_1, \ldots, \Diamond_{a_m} g_m, h_1, \ldots, h_l\}}{\{f_1, \ldots, f_n, \Box_{x_1} f_1, \ldots, \Box_{x_n} f_n, g_i\} \quad (1 \leq i \leq m)}$

π rule generates m sons.

Let n be a node generated by π-rule, n' be the parent node of n, $g_i \in FS(n)$ and $\Diamond_{a_i} g_i \in FS(n')$. Then we say n is a_i-son of n'. Let n be a node. When π rules are applied and generate b_i-sons in the expansions from the root node to n successively, we say a sequence of labels $\Diamond_{b_1} \ldots \Diamond_{b_m}$ is a path of n.

A node n is closed if the following conditions are satisfied.

1. $FS(n)$ has complementary literals, or

2. Let $\Diamond_{b_1} \ldots \Diamond_{b_m}$ be a path of n. For some constant label a and some i, $b_i = b_m = a$.
3. α-rule or β-rule is applied to n and all n's sons are closed, or π-rule is applied to n and at least one of n's sons is closed.

A node is open if it is not closed.

Expansions in T^*_{KW} terminate in finite steps because a labeled formula has only finite labels and π-rules cannot generate nodes of the same label iteratively. Therefore, we can decide whether the tableau is closed or open.

Example 6. The T^*_{KW} tableau of the formula $g : \Diamond_a P \wedge \Box_x(\neg P \vee \Diamond_b P)$ is closed as shown in Fig.3. $n_1, \ldots, n_9$ are nodes. n_3 is a-son of n_2, n_6 is b-son of n_5, n_9 is b-son of n_8. The path of n_4 is $\Diamond_a$, the path of n_7 is $\Diamond_a\Diamond_b$, and the path of n_9 is $\Diamond_a\Diamond_b\Diamond_b$. n_4 and n_7 are closed by condition 1. n_9 is closed by condition 2. As a result, n_1 is closed by condition 3.

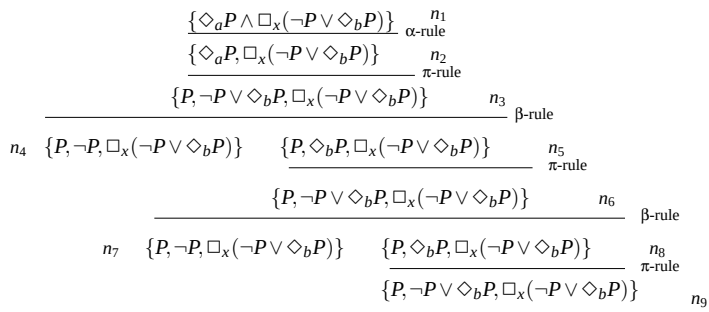


Fig. 3. T^*_{KW} tableau of $\Diamond_a P \wedge \Box_x(\neg P \vee \Diamond_b P)$

4.3 Relation between Herbrand Non-iterative Model and Well-Founded Model

Satisfiability of a formula f in the class of well-founded frames coincides with that of f^c in the class of Herbrand non-iterative frames by the following theorem.

Theorem 1. *Let f be a formula and f^c be its clausal normal form. f is unsatisfiable in the class of well-founded frames iff f^c has no Herbrand non-iterative model.*

Theorem 2 shows that T^*_{KW} judges whether a formula is unsatisfiable in the class of well-founded frames or not.

Theorem 2. *The T^*_{KW} tableau of labeled formula f^* is closed iff f is unsatisfiable in the class of well-founded frames.*

In order to prove theorem 1, we have to prove lemma 1. The lemma shows that T^*_{KW} properly judges whether a formula has a Herbrand non-iterative model. Next, we prove theorem 2. Finally, we prove theorem 1 by using lemma 1 and theorem 2.

Lemma 1. *Let f^* be a labeled formula. The T^*_{KW}tableau of f^* is closed iff f^* has no Herbrand non-iterative model.*

(proof of the only-if-part)

We get a subtree T' from closed T^*_{KW}tableau T of f^* as follows.

- The root node of T' is the root node of T.
- If n is T''s node to which α-rule is applied, T' includes n's son n_1.
- If n is T''s node to which β-rule is applied, T' includes n's two sons n_1,n_2.
- If n is T''s node to which π-rule is applied and n' is n's closed son, T' includes n'.

We prove the following proposition by induction on the construction of T'. $\overline{FS(n)}$ is an abbreviation of the conjunction of all the formulas in $FS(n)$.

From the proposition, we automatically get the result.

Proposition 1. *Let n be a node of T' and $\alpha = \Diamond_{b_1} \ldots \Diamond_{b_m}$ be a path of n. Then, $\alpha\overline{FS(n)}$ has no Herbrand non-iterative model.*

$C(s, b_1 b_2 \ldots b_m)$ is an abbreviation of $sRb_1 s \wedge b_1 sRb_2 b_1 s \wedge \ldots \wedge b_{m-1} \ldots b_1 Rb_m b_{m-1} \ldots b_1$.

- n is closed by condition 1. Suppose some world s satisfies $\alpha\overline{FS(n)}$. This contradicts the truth condition of a labeled formula as follows.
  $s \models \alpha\overline{FS(n)} \Rightarrow (b_m \ldots b_1 s \models L) \wedge (b_m \ldots b_1 s \models \bar{L}) \Rightarrow (b_m \ldots b_1 s \models L) \wedge (b_m \ldots b_1 s \not\models L)$
- n is closed by condition 2. Suppose some world s satisfies $\alpha\overline{FS(n)}$,
  then, $C(s, b_1 b_2 \ldots b_m)$ holds and b_m and b_i $(1 \leq i \leq m-1)$ are the same labels. These contradict transitivity or non-iterativity of R.
- When α-rule or β-rule is applied to n, the proposition holds evidently.
- π-rule is applied to n. $FS(n) = \{\Box_{x_1} f_1, \ldots, \Box_{x_n} f_n, \Diamond_{a_1} g_1, \ldots, \Diamond_{a_m} g_m, h_1, \ldots, h_l\}$. $FS(n') = \{f_1, \ldots, f_n, \Box_{x_1} f_1, \ldots, \Box_{x_n} f_n, g_i\}$. The path of n' is $\alpha\Diamond_{a_i}$. The induction hypothesis is that $\alpha\Diamond_{a_i}\overline{FS(n')}$ has no Herbrand non-iterative model.
  Suppose some world s satisfies $\alpha\overline{FS(n)}$. This contradicts the induction hypothesis as follows.
  $s \models \alpha\overline{FS(n)}$
  $\Rightarrow C(s,\alpha) \wedge b_m \ldots b_1 s \models \overline{FS(n)}$
  $\Rightarrow C(s,\alpha) \wedge (\bigwedge_j (b_m \ldots b_1 s \models \Box_{x_j} f_j) \wedge \bigwedge_j (b_m \ldots b_1 s \models \Diamond_{a_j} g_j) \wedge \bigwedge_j b_m \ldots b_1 s \models h_j)$
  $\Rightarrow C(s,\alpha) \wedge (\bigwedge_j (\neg b_m \ldots b_1 sRx_j \vee x_j \models f_j)$
  $\wedge \bigwedge_j (b_m \ldots b_1 sRb_j b_m \ldots b_1 s \wedge a_j b_m \ldots b_1 s \models g_j) \wedge \bigwedge_j b_m \ldots b_1 s \models h_j)$
  $\Rightarrow C(s,\alpha) \wedge (\bigwedge_j (\neg b_m \ldots b_1 sRx'_j \vee \neg x'_j Rx_j \vee x_j \models f_j) \wedge \bigwedge_j (\neg b_m \ldots b_1 sRx_j \vee x_j \models f_j)$
  $\wedge (b_m \ldots b_1 sRa_i b_m \ldots b_1 s \wedge a_i b_m \ldots b_1 s \models g_i) \wedge \bigwedge_j b_m \ldots b_1 s \models h_j)$
  $\Rightarrow C(s,\alpha) \wedge (\bigwedge_j (\neg a_i b_m \ldots b_1 sRx_j \vee x_j \models f_j) \wedge \bigwedge_j (a_i b_m \ldots b_1 s \models f_j)$
  $\wedge (b_m \ldots b_1 sRa_i b_m \ldots b_1 s \wedge a_i b_m \ldots b_1 s \models g_i)$
  $\Rightarrow C(s,\alpha) \wedge b_m \ldots b_1 sRa_i b_m \ldots b_1 s \wedge (\bigwedge_j (\neg a_i b_m \ldots b_1 sRx_j \vee x_j \models f_j)$
  $\wedge \bigwedge_j (a_i b_m \ldots b_1 s \models f_j) \wedge a_i b_m \ldots b_1 s \models g_i$
  $\Rightarrow s \models \alpha\Diamond_{a_i}\overline{FS(n')}$

(proof of the if-part)

We make a Herbrand non-iterative model and its world which satisfies f^* from the open T^*_{KW}tableau of f^*.

Since expansions of T^*_{KW}tableau terminates in finite steps, there exists a finite subtree T of the tableau of f^* such that $\{f^*\}$ is assigned to the root node of T and each node n of T is open and satisfies one of the following conditions.

I. No rules can be applied to n, or
II. α-rule or β-rule is applied to n and n has an open son n', or
III. π-rule is applied to n, $FS(n) = \{\Box_{x_1} f_1, \ldots, \Box_{x_n} f_n, \Diamond_{a_1} g_1, \ldots, \Diamond_{a_m} g_m, h_1, \ldots, h_l\}$ and n has open a_i-sons $n_{a_1}, \ldots, n_{a_m}$.

By using the subtree T, we construct the Herbrand non-iterative model $\langle W, R, V\rangle$ which satisfies f^*. W is defined by the definition of a Herbrand non-iterative model in Sect.4.1. In order to define R and V, we assign each node of T to a world in W by the following rules.

- The root node of T is assigned to $\circ$.
- If α-rule or β-rule is applied to a node n and n is assigned to a world w, the son of n is also assigned to w.
- If π-rule is applied to n and n is assigned to a world w, each a_i-son of n is assigned to the world $a_i w$ respectively.

R and V are defined as follows.

- wRw' iff $w \neq w'$ and some node assigned to the world w is an ancestor of some node assigned to the world w'.
- $w \in V(p)$ iff some node n is assigned to the world w and $p \in FS(n)$.

$\langle W, R\rangle$ satisfies transitivity and non-iterativity evidently.
From the following proposition, we automatically get the result.

Proposition 2. *For any node n in T, every labeled formula $g \in FS(n)$ is true at the world to which n is assigned.*

We prove this proposition by induction on the construction of T. Let n be a node in T, w be a world to which n is assigned.

1. n satisfies condition I.
   (a) By the definition of V, it is obvious that for any p (w.r.t $\neg p$) in $FS(n)$, $w \in V(p)$ (w.r.t $w \notin V(p)$).
   (b) Every formula of the form $\Box_x f$ in $FS(n)$ is true at w because n is a leaf node and there is no reachable world from w.
   (c) There is none of $f \wedge g, f \vee g$ and $\Diamond_a g$ in $FS(n)$.
2. n satisfies condition II.
   Let $f \wedge g$ (w.r.t. $f \vee g$) be a formula to which α-rule (w.r.t. β-rule) is applied and n' be n's son. Induction hypothesis is that every formula in $FS(n')$ is true at w.
   - f and g (w.r.t. f or g) are true at w, because the induction hypothesis holds and $FS(n')$ contains f and g (w.r.t. f or g). Hence, $f \wedge g$ (w.r.t $f \vee g$) is true at w.
   - Every formula f' besides $f \wedge g$ (w.r.t $f \vee g$) in $FS(n)$ is true at w because the induction hypothesis holds and $f' \in FS(n')$.

3. n satisfies condition III.
   By induction hypothesis, for any n's descendant n', every formula in $FS(n')$ is true at the world to which n' is assigned. —(*)
   (a) The case of $p, \neg p$ in $FS(n)$ is similar to the case 1a.
   (b) Every formula of the form $\Box_x f$ in $FS(n)$ is true at w because of the following two reasons.
      – For any n's descendant n', $f \in FS(n')$ and the property (*) holds. Therefore, f is true at the world to which n' is assigned.
      – Only the worlds to which n's descendants are assigned are reachable from w.
   (c) Every formula of the form $\Diamond_{a_i} g$ in $FS(n)$ is true at w because of the following two reasons.
      – $g \in FS(n_{a_i})$ and the property (*) holds. Therefore, g is true at $a_i w$.
      – n_{a_i} is a son of n and n_{a_i} is assigned to $a_i w$. Hence, $a_i w$ is reachable from w.
   (d) $FS(n)$ contains neither $f \wedge g$ nor $f \vee g$.

□

Next, in order to prove theorem 2, we define a tableau method for KW presented in [5]. We call it T_{KW} in this paper.

In the tableau method T_{KW}, expansion rules are as follows.

– α-rule and β-rule are the same rules as defined in the tableau method T^*_{KW}.
– **π-rule:** $\dfrac{\{\Box f_1, \ldots, \Box f_n, \Diamond g_1, \ldots, \Diamond g_m, h_1, \ldots, h_l\}}{\{f_1, \ldots, f_n, \Box f_1, \ldots, \Box f_n, g_i, \Box \neg g_i\} \quad (1 \leq i \leq m).}$

π-rule is based on the property that in the well-founded frame if there is a world where g_i is true, then there exists also a last world where g_i is true.

A node n is closed if n satisfies condition 1 or 3 in the definition of T^*_{KW}.

The following property holds between T_{KW} and well-founded frames:

f is unsatisfiable in the class of well-founded frames iff T_{KW} tableau of f is closed[5].

Now, we prove theorem 2.

(proof of the only-if-part)

We show that for any closed T^*_{KW} tableau of f^*, there exists closed T_{KW} tableau of f.

Let T be a closed tableau of f^*. We construct a T_{KW} tableau of f using the same rule as used in the expansion of T. Let T' be the result. Then, for every node n of T, there is a corresponding node n' of T' such that $FS(n) \subseteq FS(n')$.

If n is closed by condition 1, $FS(n')$ also has complementary literals because $FS(n) \subseteq FS(n')$. Therefore n' is also closed.

Suppose n is closed by condition 2, i.e. some constant label a occurs in the path of n twice. Then, the following properties hold.

– Let n_1 be n's parent. Then, n is a-son of n_1, $\Diamond_a g \in FS(n_1)$ and $g \in FS(n)$.
– For some ancestor n_2 of n, the following property holds. Let n_3 be n_2's parent. Then, n_2 is a-son of n_3, $\Diamond_a g \in FS(n_3)$ and $g \in FS(n_2)$.

Let n',n'_1,n'_2,n'_3 be nodes in T' corresponding to n,n_1,n_2,n_3 in T respectively. Then, $\Box\neg g \in FS(n'_2)$ holds because n'_2 is n'_3's son expanded by π-rule. Hence, $\Box\neg g \in FS(n'_1)$ and $\neg g \in FS(n')$. Now, $g \in FS(n')$ holds since $g \in FS(n)$ and $FS(n) \subseteq FS(n')$. Hence, n' is also closed.

T_{KW}tableau have the same condition 3 as T^*_{KW}tableau. Hence, T' is closed.

(proof of the if-part)

The Herbrand non-iterative model generated in the proof of the if-part of lemma 1 is a well-founded model because it satisfies well-foundedness. Moreover, it satisfies f. Hence, for any open T^*_{KW}tableau of f^*, there exists a well-founded model which satisfies f. □

By lemma 1 and theorem 2, a formula f is unsatisfiable in the class of well-founded frames if and only if the labeled formula f^* does not have a Herbrand non-iterative model. Moreover, f^* and clausal normal form f^c are equivalent in the Herbrand non-iterative model. Hence, theorem 1 holds. □

4.4 Soundness of Resolution Method

Theorem 3. *If a formula f is refutable, then f is unsatisfiable in the class of well-founded frames.*

Proof. By theorem 1, it is enough to show that for every resolution rule, if the premises is true in a world in a Herbrand non-iterative model, then the conclusion is also true.

Let Γ be $\alpha_1 L_1 \vee \ldots \vee \alpha_m L_m$ and γ be a sequence of modal operators with labels. Then, we use $\gamma\Gamma$ as an abbreviation of $\gamma\alpha_1 L_1 \vee \ldots \vee \gamma\alpha_m L_m$.

resolution rule1

By induction on the construction of the unification, we show that if $(\alpha L \vee \Gamma) \wedge (\beta\bar{L} \vee \Gamma')$ is true in a world in a Herbrand non-iterative model, then $(\alpha\bot \vee \Gamma \vee \Gamma')^\sigma$ is also true.

- In the case of the unification of two empty sequences, immediate from the fact $s \models ((L \vee \Gamma) \wedge (\bar{L} \vee \Gamma')) \Rightarrow s \models (\bot \vee \Gamma \vee \Gamma')$.
- In the case of the unification of $\Box_x\alpha_2$ and $\gamma\beta_2$, by induction on the length of γ, we show that if $(\Box_x\alpha_2 L \vee \Box_x\Gamma_1 \vee \Gamma_2) \wedge (\gamma\beta_2\bar{L} \vee \gamma\Gamma'_1 \vee \Gamma'_2)$ is true in a world s of a Herbrand non-iterative model, then $(\Box_x\alpha_2\bot \vee \Box_x\Gamma_1 \vee \Gamma_2 \vee \gamma\Gamma'_1 \vee \Gamma'_2)^{(\gamma/\Box_x)\cdot\sigma'}$ is also true in s.
  - The length of γ is 1.
    If γ is $\Diamond_a$,
    $s \models ((\Box_x\alpha_2 L \vee \Box_x\Gamma_1 \vee \Gamma_2) \wedge (\Diamond_a\beta_2\bar{L} \vee \Diamond_a\Gamma'_1 \vee \Gamma'_2))$
    $\Leftrightarrow ((\neg sRx \vee (x \models \alpha_2 L)) \vee (\neg sRx \vee (x \models \Gamma_1)) \vee (s \models \Gamma_2)) \wedge ((sRas \wedge (as \models \beta_2\bar{L})) \vee (sRas \wedge (as \models \Gamma'_1)) \vee (s \models \Gamma'_2))$
    $\Rightarrow (sRas \wedge (as \models ((\alpha_2 L \vee \Gamma_1) \wedge (\beta_2\bar{L} \wedge \Gamma'_1)))) \vee (s \models \Gamma_2) \vee (s \models \Gamma'_2) \cdots (*)$.
    By induction hypothesis, $as \models ((\alpha_2 L \vee \Gamma_1) \wedge (\beta_2\bar{L} \vee \Gamma'_1)) \Rightarrow as \models (\alpha_2\bot \vee \Gamma_1 \vee \Gamma'_1)^{\sigma'}$.
    Hence, $(*) \Rightarrow (sRas \wedge as \models (\alpha_2\bot \vee \Gamma_1 \vee \Gamma'_1)^{\sigma'}) \vee (s \models \Gamma_2) \vee (s \models \Gamma'_2) \Rightarrow s \models (\Box_x\alpha_2\bot \vee \Box_x\Gamma_1 \vee \Diamond_a\Gamma'_1 \vee \Gamma_2 \vee \Gamma'_2)^{\Diamond_a/\Box_x\cdot\sigma'}$.
    Similarly, we can show a proof of the case that γ is $\Box_y$.
  - The length of γ is $k > 1$.
    If γ is $\Diamond_a\gamma'$,
    $s \models ((\Box_x\alpha_2 L \vee \Box_x\Gamma_1 \vee \Gamma_2) \wedge (\Diamond_a\gamma'\beta_2\bar{L} \vee \Diamond_a\gamma'\Gamma'_1 \vee \Gamma'_2))$

$\Leftrightarrow ((\neg sRx \vee (x \models \alpha_2 L)) \vee (\neg sRx \vee (x \models \Gamma_1)) \vee (s \models \Gamma_2)) \wedge ((sRas \wedge (as \models \gamma' \beta_2 \bar{L})) \vee (sRas \wedge (as \models \gamma' \Gamma_1')) \vee (s \models \Gamma_2'))$
$\Rightarrow ((\neg sRx' \vee \neg x' Rx \vee (x \models \alpha_2 L)) \vee (\neg sRx' \vee \neg x' Rx \vee (x \models \Gamma_1)) \vee (s \models \Gamma_2)) \wedge ((sRas \wedge (as \models \gamma' \beta_2 \bar{L})) \vee (sRas \wedge (as \models \gamma' \Gamma_1')) \vee (s \models \Gamma_2'))$
$\Rightarrow (sRas \wedge as \models ((\Box_x \alpha_2 L \vee \Box_x \Gamma_1) \wedge (\gamma' \beta_2 \bar{L} \vee \gamma' \Gamma_1'))) \vee (s \models \Gamma_2) \vee (s \models \Gamma_2') \cdots (*)$.
By induction hypothesis, $as \models ((\Box_x \alpha_2 L \vee \Box_x \Gamma_1) \wedge (\gamma' \beta_2 \bar{L} \vee \gamma' \Gamma_1')) \Rightarrow as \models (\Box_x \alpha_2 \perp \vee \Box_x \Gamma_1 \vee \gamma' \Gamma_1')^{\gamma'/\Box_x \cdot \sigma'}$.
Hence, $(*) \Rightarrow (sRas \wedge as \models (\Box_x \alpha_2 \perp \vee \Box_x \Gamma_1 \vee \gamma' \Gamma_1')^{\gamma'/\Box_x \cdot \sigma'}) \vee (s \models \Gamma_2) \vee (s \models \Gamma_2') \Rightarrow s \models (\Box_x \alpha_2 \perp \vee \Box_x \Gamma_1 \vee \Diamond_a \gamma' \Gamma_1' \vee \Gamma_2')^{\Diamond_a \gamma'/\Box_x \cdot \sigma'}$.
Similarly, We can show a proof of the case that γ is $\Box_y \gamma'$.

– we can prove the case of the unification of $\Diamond_a \alpha_2$ and $\Diamond_a \beta_2$ similarly.

resolution rule2, resolution rule3

We can show the proof in the case of the resolution rule2 in the same way as used in the proof in the case of resolution rule1. The proof in the case of the resolution rule3 is trivial.

resolution rule4

Suppose that there exists a Herbrand non-iterative model and its world s in which $\alpha\gamma L$ is true, where α is $\Diamond_{a_1} \ldots \Diamond_{a_m}$ and a_i and a_j are the same labels for some i, j. Then, $a_{i-1} \ldots a_1 s R a_i a_{i-1} \ldots a_1 s$ must hold for any i $(1 \leq i \leq m)$. Therefore, one of the following is satisfied by transitivity of reachability of the worlds in the Herbrand non-iterative model.

– $i = j - 1$ and $a_{i-1} \ldots a_1 s R a_i \ldots a_1 s \wedge a_{j-1} \ldots a_1 s R a_j \ldots a_1 s$
– $i > j - 1$ and $a_{i-1} \ldots a_1 s R a_i \ldots a_1 s \wedge a_i \ldots a_1 s R a_{j-1} \ldots a_1 s \wedge a_{j-1} \ldots a_1 s R a_j \ldots a_1 s$

However, both cannot hold because of non-iterativity of reachability of the worlds in the Herbrand non-iterative model. Hence, there is no world s in Herbrand non-iterative model where $\alpha\gamma L$ is true. Therefore, if $\alpha\gamma L \vee \Gamma$ is true in a world in a Herbrand non-iterative model, then Γ is true. □

4.5 Completeness of Resolution Method

Theorem 4. *If a formula f is unsatisfiable in the class of well-founded frames, then f is refutable.*

Proof. By theorem 2, it is enough to show that f^c is refutable if T^*_{KW}tableau of f^* is closed.

We get a subtree T' from closed T^*_{KW}tableau T of f^* by the same way as used in the proof of the only-if-part of lemma 1. We shall prove the following proposition by induction on the construction of T'. From this proposition, we automatically get the result.

Proposition 3. *Let n be a node of T' and α be a path of n. Then, $\alpha\overline{FS(n)}$ is refutable.*

– n is closed by condition 1. Since $L, \bar{L} \in FS(n)$, there are clauses αL and $\alpha\bar{L}$ in the clausal normal form $(\alpha\overline{FS(n)})^c$. Hence, we can refute it by the resolution rules 1 and 3.

– n is closed by condition 2. Let $\bigwedge_j \bigvee_k \gamma^{jk} L^{jk}$ be $(\overline{FS(n)})^c$. Then, $(\alpha\overline{FS(n)})^c$ is $\bigwedge_j \bigvee_k \alpha\gamma^{jk} L^{jk}$. We choose a clause $\Gamma = \bigvee_{1\le k\le m} \alpha\gamma^{jk}\alpha L^{jk}$ from $(\alpha\overline{FS(n)})^c$. We can refute Γ by applying the resolution rule4 m times.
– When α-rule or β-rule is applied to n, the proposition holds evidently.
– π-rule is applied to n. Let $\{\Box_{x_1} f_1, \ldots, \Box_{x_n} f_n, \Diamond_{a_1} g_1, \ldots, \Diamond_{a_m} g_m,\ h_1, \ldots, h_{m'}\}$ be $FS(n)$ and n_l be n's a_l-son. Then $FS(n_l)$ is $\{f_1, \ldots, f_n, \Box_{y_1} f_1, \ldots, \Box_{y_n} f_n, g_l\}$.
Let $\bigwedge_j \bigvee_k \gamma_i^{jk} L_i^{jk}$ be f_i^c and $\bigwedge_j \bigvee_{1\le k\le k_{ij}} \gamma'_i{}^{jk} L'_i{}^{jk}$ be g_i^c. Then, $(\alpha\overline{FS(n)})^c$ is
$\bigwedge_{1\le i\le n}\bigwedge_j \bigvee_k \alpha\Box_{x_i^j}\gamma_i^{jk} L_i^{jk} \wedge \bigwedge_{1\le i\le m}\bigwedge_j \bigvee_{1\le k\le k_{ij}} \alpha\Diamond_{a_i}\gamma'_i{}^{jk} L'_i{}^{jk} \wedge \bigwedge_{1\le i\le m'} h_i^c$
and $(\alpha\Diamond_{a_l}\overline{FS(n_l)})^c$ is
$\bigwedge_{1\le i\le n}\bigwedge_j \bigvee_k \alpha\Diamond_{a_l}\gamma_i^{jk} L_i^{jk} \wedge \bigwedge_{1\le i\le n}\bigwedge_j \bigvee_k \alpha\Diamond_{a_l}\Box_{y_i^j}\gamma_i^{jk} L_i^{jk} \wedge \bigwedge_j \bigvee_{1\le k\le k_{lj}} \alpha\Diamond_{a_l}\gamma'_l{}^{jk} L'_l{}^{jk}$.
Induction hypothesis is that $(\alpha\Diamond_{a_l}\overline{FS(n_l)})^c$ is refutable.
Then, we can refute $\Delta = \bigwedge_{1\le i\le n}\bigwedge_j \bigvee_k \alpha\Box_{x_i^j}\gamma_i^{jk} L_i^{jk} \wedge \bigwedge_j \bigvee_{1\le k\le k_{lj}} \alpha\Diamond_{a_l}\gamma'_l{}^{jk} L'_l{}^{jk}$ as follows.

1. We deduce the clause $\bigvee_k \alpha\Diamond_{a_l}\gamma_i^{jk} L_i^{jk}$ for any i, j from Δ. We can deduce it by applying the resolution rule2 to the clauses $\bigvee_k \alpha\Box_{x_i^j}\gamma_i^{jk} L_i^{jk}$ and $\bigvee_{1\le k\le k_{l1}} \alpha\Diamond_{a_l}\gamma_l^{1k} L'_l{}^{1k}$ with the substitution $\{\Diamond_{a_l}/\Box_{x_i^j}\}$ k_{l1} times for any i, j.
2. We obtained the clauses $\bigwedge_{1\le i\le n}\bigwedge_j \bigvee_k \alpha\Diamond_{a_l}\gamma_i^{jk} L_i^{jk}$ at 1. Therefore, we refute $\Gamma = \bigwedge_{1\le i\le n}\bigwedge_j \bigvee_k \alpha\Diamond_{a_l}\gamma_i^{jk} L_i^{jk} \wedge \Delta$
$= \bigwedge_{1\le i\le n}\bigwedge_j \bigvee_k \alpha\Diamond_{a_l}\gamma_i^{jk} L_i^{jk} \wedge \bigwedge_{1\le i\le n}\bigwedge_j \bigvee_k \alpha\Box_{x_i^j}\gamma_i^{jk} L_i^{jk} \wedge \bigwedge_j \bigvee_{1\le k\le k_{lj}} \alpha\Diamond_{a_l}\gamma'_l{}^{jk} L'_l{}^{jk}$ as follows.
We define the relation $\gg$ between the clause which appears in the refutation of $(\alpha\Diamond_{a_l}\overline{n_l})^c$ and the clause which appears in the refutation of Γ and define type of a clause as follows.

type	clause in $(\alpha\Diamond_{a_l}\overline{n_l})^c$	$\gg$	clause in Γ
A	$\bigvee_k \alpha\Diamond_{a_l}\gamma_i^{jk} L_i^{jk}$	$\gg$	$\bigvee_k \alpha\Diamond_{a_l}\gamma_i^{jk} L_i^{jk}$
B	$\bigvee_k \alpha\Diamond_{a_l}\Box_{y_i^j}\gamma_i^{jk} L_i^{jk}$	$\gg$	$\bigvee_k \alpha\Box_{x_i^j}\gamma_i^{jk} L_i^{jk}$
A	$\bigvee_k \alpha\Diamond_{a_l}\gamma'_l{}^{jk} L'_l{}^{jk}$	$\gg$	$\bigvee_k \alpha\Diamond_{a_l}\gamma'_l{}^{jk} L'_l{}^{jk}$

$\Gamma_3 \gg \Gamma'_3$ if $\Gamma_1 \gg \Gamma'_1, \Gamma_2 \gg \Gamma'_2$ and $\dfrac{\Gamma_1 \quad \Gamma_2}{\Gamma_3}$, $\dfrac{\Gamma'_1 \quad \Gamma'_2}{\Gamma'_3}$.

For any resolution rules,
- if there is a premise of type A, the conclusion has type A, and
- if all the premises have type B, the conclusion has type B.

We can refute Γ by using the corresponding clauses as used in the refutation of $(\alpha\Diamond_{a_l}\overline{n_l})^c$. Let σ be a substitution used by the resolution rule1 or the resolution rule2 in the refutation of $(\alpha\Diamond_{a_l}\overline{n_l})^c$. Then, we use the following substitution σ' in the refutation of Γ.
- $\dfrac{\text{The resolution rule applied to clauses of type } B}{\sigma' = (\text{the substitution by replacing } \Box_{y_i^j} \text{ with } \Box_{x_i^j} \text{ in } \sigma)}$
- $\dfrac{\text{The resolution rule applied to clause of type } B \text{ and clause of type } A}{\sigma' = (\sigma - \{\beta/\Box_{y_i^j}\}) \cup \{\Diamond_{a_l}\beta/\Box_{x_i^j}\}}$

- $\dfrac{\text{The resolution rule applied to clauses of type } A}{\sigma' = \sigma}$

Now, each clause in Δ appears in $(\alpha\overline{FS(n)})^c$. Therefore, $(\alpha\overline{FS(n)})^c$ is refutable.

□

5 Conclusion

In this paper, we have introduced Herbrand non-iterative frames and constructed an unification-based prover which checks unsatisfiability of the modal logic KW. The satisfiability of a labeled formula in the Herbrand non-iterative frames coincides with the one of its original formula in the well-founded frames.

The main idea introduced in this paper is that the restriction on frames with infinite sequence of reachable worlds can be reformed into the restriction on Herbrand frames with iterations of the transition. Many temporal structures, such as time structure isomorphic to the natural number's structure, satisfy the restrictions on infinite sequences of reachable worlds. Therefore, the idea is applicable to many temporal structures, and unification-based resolution method can be adapted to wider range of (and more practical system of) modal logic.

References

1. Melvin C. Fitting. *Proof Methods for Modal and Intutionistic Logics*. Reidel, Dordrecht, 1983.
2. Annie Foret. Rewrite rule systems for modal propositional logic. *The Journal of Logic Programming*, 12:281–298, 1992.
3. G.D'Agostino, A. Montanari, and A. Policriti. A set-theoretic translation method for polymodal logics. *Journal of Automated Reasoning*, 15(3):317–337, 1995.
4. G.E.Hughes and M.J.Cresswell. *A New Inroduction to Modal Logic*. Routledge, 1996.
5. Rajeev Goré. Technical report tr-arp-15-95. Technical report, Research School of Information Sciences and Engineering and Centre for Information Science Research Australian National University, http://arp.anu.edu.au/, 1995.
6. H.J.Ohlbach. A resolution calculus for modal logics. In *CADE-9*, volume 310 of *Lecture Notes in Computer Science*, pages 500–516. Springer, 1988.
7. H.J.Ohlbach. Combining hilbert style and semantic reasoning in a resolution framework. In *Automated Deduction, CADE-15*, volume 1421 of *LNAI*, pages 205–219, 1998.
8. Andreas Nonnengart. Resolution-based calculi for modal and temporal logics. In *Automated Deduction - CADE-13*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 598–612. Springer, 1996.
9. Jens Otten and Christoph Kreitz. T-string unification: Unifying prefixes in non-classical proof methods. In *Theorem Proving with Analytic Tableaux and Related Methods*, volume 1071 of *Lecture Notes in Artificial Intelligence*, pages 244–260. Springer, 1996.
10. P.Enjalbert and L.Farinas del Cerro. Modal resolution in clausal form. *Theoretical Computer Science*, 65(1):1–33, 1989.
11. Lincoln A. Wallen. *Automated proof search in non-classical logics*. MIT Press, 1990.
12. Naoki Yonezaki and Takashi Hayama. Self-substitution in modal unification. In *Information Modeling and Knowledge Bases IV*, pages 180–195. IOS Press, 1993.

A NExpTime-Complete Description Logic Strictly Contained in C^2

Stephan Tobies

LuFg Theoretical Computer Science, RWTH Aachen
Ahornstr. 55, 52074 Aachen, Germany
Phone: +49-241-8021109
E-mail: tobies@informatik.rwth-aachen.de

Abstract. We examine the complexity and expressivity of the combination of the Description Logic $\mathcal{ALCQI}$ with a terminological formalism based on cardinality restrictions on concepts. This combination can naturally be embedded into c^2, the two variable fragment of predicate logic with counting quantifiers. We prove that $\mathcal{ALCQI}$ has the same complexity as c^2 but does not reach its expressive power.

Keywords. Description Logic, Counting, Complexity, Expressivity

1 Introduction

Description Logic (DL) systems can be used in knowledge based systems to represent and reason about taxonomical knowledge of problem domain in a semantically well-defined manner [WS92]. These systems usually consist at least of the following three components: a DL, a terminological component, and a reasoning service.

Description logics allow the definition of complex concepts (unary predicates) and roles (binary relations) to be built from atomic ones by the application of a given set of constructors; for example the following concept describes those fathers having at least two daughters:

$$\texttt{Parent} \sqcap \texttt{Male} \sqcap (\geq\ 2\ \texttt{hasChild}\ \texttt{Female})$$

The terminological component (TBox) allows for the organisation of defined concepts and roles. The TBox formalisms studied in the DL context range from weak ones allowing only for the introduction of abbreviations for complex concepts, over TBoxes capable of expressing various forms of axioms, to cardinality restrictions that can express restrictions on the number of elements a concept may have. Consider the following three TBox expressions:

$$\texttt{BusyParent} = \texttt{Parent} \sqcap (\geq\ 2\ \texttt{hasChild}\ \texttt{Toddler})$$

$$\texttt{Male} \sqcup \texttt{Female} = \texttt{Person} \sqcap (=\ 2\ \texttt{hasChild}^{-1}\ \texttt{Parent})$$

$$(\leq\ 2\ \texttt{Person} \sqcap (\leq\ 0\ \texttt{hasChild}^{-1}\ \texttt{Parent}))$$

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 292–306, 1999.

The first introduces `BusyParent` as an abbreviation for a more complex concept, the second is an axiom stating that `Male` and `Female` are exactly those persons having two parents, the third is a cardinality restriction expressing that in the domain of discourse there are at most two earliest ancestors.

The reasoning service performs task like subsumption or consistency test for the knowledge stored in the TBox. There exist sound and complete algorithms for reasoning in a large number of DLs and different TBox formalisms that meet the known worst-case complexity of these problems (see [DLNN97] for an overview). Generally, reasoning for DLs can be performed in four different ways:

- by structural comparison of syntactical normal forms of concepts [BPS94].
- by tableaux algorithms that are hand-tailored to suit the necessities of the operators used to form the DL and the TBox formalism. Initially, these algorithms were designed to decide inference problems only for the DL without taking into account TBoxes, but it is possible to generalise these algorithms to deal with different TBox formalisms. Most DLs handled this way are at most PSpace complete but additional complexity may arise from the TBox. The complexity of the tableaux approach usually meets the known worst-case complexity of the problem [SSS91,DLNN97].
- by perceiving the DL as a (fragment of a) modal logic such as PDL [GL96]; for many DLs handled in this manner already concept satisfiability is ExpTime-complete, but axioms can be "internalised" [Baa91] into the concepts and hence do not increase the complexity.
- by translation of the problem into a fragment or first order other logic with a decidable decision problem [Bor96,OSH96].

From the fragments of predicate logic that are studied in the second context, only C^2, the two variable fragment of first order predicate logic augmented with counting quantifiers, is capable of dealing with counting expressions that are commonly used in DLs; similarly it is able to express cardinality restrictions. Another thing that comes "for free" when translating DLs into first order logic is the ability to deal with inverse roles.

Combining all these parts into a single DL, one obtains the DL $\mathcal{ALCQI}$—the well-known DL $\mathcal{ALC}$ [SSS91] augmented by qualifying number restrictions ($\mathcal{Q}$) and inverse roles ($\mathcal{I}$). In this work we study both complexity and expressivity of $\mathcal{ALCQI}$ combined with TBoxes based on cardinality restrictions.

Regarding the complexity we show that $\mathcal{ALCQI}$ with cardinality restrictions already is NExpTime-hard and hence has the same complexity as C^2 [PST97][1]. To our knowledge this is the first DL for which NExpTime-completeness has formally been proved. Since $\mathcal{ALCQI}$ with TBoxes consisting of axioms is still in ExpTime, this indicates that cardinality restrictions are algorithmically hard to handle.

[1] The NExpTime-result is valid only if we assume unary coding of numbers in the counting quantifiers. This is the standard assumption made by most results concerning the complexity of DLs.

Despite the fact that both $\mathcal{ALCQI}$ and C^2 have the same worst-case complexity we show that $\mathcal{ALCQI}$ lacks some of the expressive power of C^2. Properties of binary predicates (e.g. reflexivity) that are easily expressible in C^2 can not be expressed in $\mathcal{ALCQI}$. We establish our result by giving an Ehrenfeucht-Fraïssé game that exactly captures the expressivity of $\mathcal{ALCQI}$ with cardinality restrictions. This is the first time in the area of DL that a game-theoretic characterisation is used to prove an expressivity result involving TBox formalisms. The game as it is presented here is not only applicable to $\mathcal{ALCQI}$ with cardinality restrictions; straightforward modifications make it applicable to both $\mathcal{ALCQ}$ as well as to weaker TBox formalisms such as terminological axioms.

In [Bor96] a DL is presented that has the same expressivity as C^2. This expressivity result is one of the main results of that paper and the DL combines a large number of constructs; the paper does not study the computational complexity of the presented logics. Our motivation is of a different nature: we study the complexity and expressivity of a DL consisting of only a minimal set of constructs that seem sensible when a reduction of that DL to C^2 is to be considered.

2 The Logic $\mathcal{ALCQI}$

Definition 1. *A* signature *is a pair $\tau = (N_C, N_R)$ where N_C is a finite set of* concepts names *and N_R is a finite set of* role names. Concepts *in $\mathcal{ALCQI}$ are built inductively from these using the following rules: All $A \in N_C$ are concepts, and, if C, C_1, and C_2 are concepts, then also $\neg C$, $C_1 \sqcap C_2$, and $(\geq\ n\ S\ C)$ with $n \in \mathbb{N}$, and $S = R$ or $S = R^{-1}$ for some $R \in N_R$ are concepts. We define $C_1 \sqcup C_2$ as an abbreviation for $\neg(\neg C_1 \sqcap \neg C_2)$ and $(\leq\ n\ S\ C)$ as an abbreviation for $\neg(\geq\ (n+1)\ S\ C)$. We also use $(=\ n\ S\ C)$ as an abbreviation for $(\leq\ n\ S\ C) \sqcap (\geq\ n\ S\ C)$.*

A cardinality restriction *of $\mathcal{ALCQI}$ is an expression of the form $(\geq n\ C)$ or $(\leq n\ C)$ where C is a concept and $n \in \mathbb{N}$; a* TBox *T of $\mathcal{ALCQI}$ is a finite set of cardinality restrictions.*

The semantics of a concept is defined relative to an interpretation *$\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, which consists of a domain $\Delta^{\mathcal{I}}$ and a valuation $(\cdot^{\mathcal{I}})$ which maps each concept name A to a subset $A^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$ and each role name R to a subset $R^{\mathcal{I}}$ of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. This valuation is inductively extended to arbitrary concept definitions using the following rules, where $\sharp M$ denotes the cardinality of a set M:*

$$(\neg C)^{\mathcal{I}} := \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}, \quad (C_1 \sqcap C_2)^{\mathcal{I}} := C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}},$$
$$(\geq n\ R\ C)^{\mathcal{I}} := \{a \in \Delta^{\mathcal{I}} \mid \sharp\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \geq n\},$$
$$(\geq n\ R^{-1}\ C)^{\mathcal{I}} := \{a \in \Delta^{\mathcal{I}} \mid \sharp\{b \in \Delta^{\mathcal{I}} \mid (b,a) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \geq n\}.$$

An interpretation $\mathcal{I}$ satisfies *a cardinality restriction $(\geq n\ C)$ iff $\sharp(C^{\mathcal{I}}) \geq n$ and it satisfies $(\leq n\ C)$ iff $\sharp(C^{\mathcal{I}}) \leq n$. It satisfies a TBox T iff it satisfies all cardinality restrictions in T; in this case, $\mathcal{I}$ is called a* model *of T and we will denote this fact by $\mathcal{I} \models T$. A TBox that has a model is called* consistent.

$$
\begin{array}{lll}
\Psi_x(A) & := Ax & \text{for } A \in N_C \\
\Psi_x(\neg C) & := \neg\Psi_x(C) & \\
\Psi_x(C_1 \sqcap C_2) & := \Psi_x(C_1) \wedge \Psi_x(C_2) & \\
\Psi_x(\geq\ n\ R\ C) & := \exists^{\geq n} y.(Rxy \wedge \Psi_y(C)) & \\
\Psi_x(\geq\ n\ R^{-1}\ C) & := \exists^{\geq n} y.(Ryx \wedge \Psi_y(C)) & \\
\Psi(\bowtie\ n\ C) & := \exists^{\bowtie n} x.\Psi_x(C) & \text{for } \bowtie \in \{\geqslant, \leqslant\} \\
\Psi(T) & := \bigwedge\{\Psi(\bowtie\ n\ C) \mid (\bowtie\ n\ C) \in T\} &
\end{array}
$$

Fig. 1. The translation from $\mathcal{ALCQI}$ into C^2 adopted from [Bor96]

With $\mathcal{ALCQ}$ we denote the fragment of $\mathcal{ALCQI}$ that does not contain any inverse roles R^{-1}.

TBoxes consisting of cardinality restrictions have first been studied for the DL $\mathcal{ALCQ}$ in [BBH96]. They can express terminological axioms of the form $C = D$ that are the most expressive TBox formalisms usually studied in the DL context [GL96] as follows: obviously, two concepts C, D have the same extension in an interpretation iff it satisfies the cardinality restriction $(\leq\ 0\ (C \sqcap \neg D) \sqcup (\neg C \sqcap D))$. One standard inference service for DL systems is satisfiability of a concept C with respect to a TBox T (i.e., is there an interpretation $\mathcal{I}$ such that $\mathcal{I} \models T$ and $C^{\mathcal{I}} \neq \emptyset$). For a TBox formalism based on cardinality restrictions this is easily reduced to TBox consistency, because obviously C is satisfiable with respect to T iff $T \cup \{(\geq\ 1\ C)\}$ is a consistent TBox. To this the reason we will restrict our attention to TBox consistency; other standard inferences such as concept subsumption can be reduced to consistency as well.

Until now there does not exist a tableaux based decision procedure for $\mathcal{ALCQI}$ TBox consistency. Nevertheless this problem can be decided with the help of a well-known translation of $\mathcal{ALCQI}$-TBoxes to C^2 [Bor96] given in Fig. 1. The logic C^2 is fragment of predicate logic that allows only two variables but is enriched with counting quantifiers of the form $\exists^{\geq l}$. The translation Ψ yields a satisfiable sentence of C^2 if and only if the translated TBox is consistent. Since the translation from $\mathcal{ALCQI}$ to C^2 can be performed in linear time, the NExpTime upper bound [GOR97,PST97] for satisfiability of C^2 directly carries over to $\mathcal{ALCQI}$-TBox consistency:

Lemma 1. *Consistency of an $\mathcal{ALCQI}$-TBox T can be decided in* NExpTime.

Please note that the NExpTime-completeness result from [PST97] is only valid if we assume unary coding of numbers in the input; this implies that a large number like 1000 may not be stored in logarithmic space in some k-ary representation but consumes 1000 units of storage. This is the standard assumption made by most results concerning the complexity of DLs. We will come back to this issue later in this paper.

3 $\mathcal{ALCQI}$ Is NExpTime-Complete

To show that NExpTime is also the lower bound for the complexity of TBox consistency we use a bounded version of the domino problem. Domino problems [Wan63,Ber66] have successfully been employed to establish undecidability and complexity results for various description and modal logics [Spa93,BS99].

3.1 Domino Systems

Definition 2. *For an $n \in \mathbb{N}$ let $\mathbb{Z}_n$ denote the set $\{0, \ldots, n-1\}$ and $\oplus_n$ denote the addition modulo n. A* domino system *is a triple $\mathcal{D} = (D, H, V)$, where D is a finite set (of tiles) and $H, V \subseteq D \times D$ are relations expressing horizontal and vertical compatibility constraints between the tiles. For $s, t \in \mathbb{N}$ let $U(s,t)$ be the torus $\mathbb{Z}_s \times \mathbb{Z}_t$ and $w = w_0, \ldots, w_{n-1}$ be an n-tuple of tiles (with $n \leq s$). We say that $\mathcal{D}$* tiles *$U(s,t)$* with initial condition *w iff there exists a mapping $\tau : U(s,t) \to D$ such that, for all $(x,y) \in U(s,t)$,*

- *if $\tau(x,y) = d$ and $\tau(x \oplus_s 1, y) = d'$ then $(d,d') \in H$ (horizontal constraint);*
- *if $\tau(x,y) = d$ and $\tau(x, y \oplus_t 1) = d'$ then $(d,d') \in V$ (vertical constraint);*
- *$\tau(i,0) = w_i$ for $0 \leq i < n$ (initial condition).*

Bounded domino systems are capable of expressing the computational behaviour of restricted, so called *simple*, Turing Machines (TM). This restriction is non-essential in the following sense: Every language accepted in time $T(n)$ and space $S(n)$ by some one-tape TM is accepted within the same time and space bounds by a simple TM, as long as $S(n), T(n) \geq 2n$ [BGG97].

Theorem 1 ([BGG97], Theorem 6.1.2). *Let M be a simple TM with input alphabet Σ. Then there exists a domino system $\mathcal{D} = (D, H, V)$ and a linear time reduction which takes any input $x \in \Sigma^*$ to a word $w \in D^*$ with $|x| = |w|$ such that*

- *If M accepts x in time t_0 with space s_0, then $\mathcal{D}$ tiles $U(s,t)$ with initial condition w for all $s \geq s_0 + 2, t \geq t_0 + 2$;*
- *if M does not accept x, then $\mathcal{D}$ does not tile $U(s,t)$ with initial condition w for any $s, t \geq 2$.*

Corollary 1. *Let M be a (w.l.o.g. simple) non-deterministic TM with time- (and hence space-) bound 2^{n^d} (d constant) deciding an arbitrary* NExpTime*-complete language $\mathcal{L}(M)$ over the alphabet Σ. Let $\mathcal{D}$ be the according domino system and and trans the reduction from Theorem 1. The following is a* NExpTime*-hard problem:*

> *Given an initial condition $w = w_0, \ldots, w_{n-1}$ of length n. Does $\mathcal{D}$ tile $U(2^{n^d+1}, 2^{n^d+1})$ with initial condition w?*

Proof. The function *trans* is a linear reduction from $\mathcal{L}(M)$ to the problem above: For $v \in \Sigma^*$ with $|v| = n$ it holds that $v \in \mathcal{L}(M)$ iff M accepts v in time and space $2^{|v|^d}$ iff $\mathcal{D}$ tiles $U(2^{n^d+1}, 2^{n^d+1})$ with initial condition *trans*(v). □

3.2 Defining a Torus of Exponential Size

Just as defining infinite grids is the key problem in proving undecidability by reduction of unbounded domino problems, defining a torus of exponential size is the key to obtaining a NExpTime-completeness proof by reduction of bounded domino problems.

To be able to apply Corollary 1 to TBox consistency for $\mathcal{ALCQI}$ we must characterise the torus $\mathbb{Z}_{2^n} \times \mathbb{Z}_{2^n}$ with a TBox of polynomial size. To characterise this torus we will use $2n$ concepts $X_0, \ldots, X_{n-1}$ and $Y_0, \ldots, Y_{n-1}$, where X_i codes the ith bit of the binary representation of the X-coordinate of an element a:

For an interpretation $\mathcal{I}$ and an element $a \in \Delta^{\mathcal{I}}$, we define $pos(a)$ by

$$pos(a) := (xpos(a), ypos(a)) := \left(\sum_{i=0}^{n-1} x_i \cdot 2^i, \quad \sum_{i=0}^{n-1} y_i \cdot 2^i\right), where$$

$$x_i = \begin{cases} 0, & \text{if } a \notin X_i^{\mathcal{I}} \\ 1, & \text{otherwise} \end{cases} \qquad y_i = \begin{cases} 0, & \text{if } a \notin Y_i^{\mathcal{I}} \\ 1, & \text{otherwise} \end{cases}.$$

We use a well-known characterisation of binary addition (e.g. [BGG97]) to relate the positions of the elements in the torus:

Lemma 2. *Let x, x' be natural numbers with binary representations*

$$x = \sum_{i=0}^{n-1} x_i \cdot 2^i \quad \textit{and} \quad x' = \sum_{i=0}^{n-1} x'_i \cdot 2^i.$$

This implies:

$$x' \equiv x + 1 \pmod{2^n} \quad \textit{iff} \quad \bigwedge_{k=0}^{n-1} \Big(\bigwedge_{j=0}^{k-1} x_j = 1\Big) \rightarrow (x_k = 1 \leftrightarrow x'_k = 0)$$
$$\wedge \bigwedge_{k=0}^{n-1} \Big(\bigvee_{j=0}^{k-1} x_j = 0\Big) \rightarrow (x_k = x'_k)$$

where the empty conjunction and disjunction are interpreted as true and false respectively.

We define the TBox T_n to consist of the following cardinality restrictions:

$$(\forall\ (\geq\ 1\ \mathit{east}\ \top)), \quad (\forall\ (\geq\ 1\ \mathit{north}\ \top)),$$
$$(\forall\ (=\ 1\ \mathit{east}^{-1}\ \top)), \quad (\forall\ (=\ 1\ \mathit{north}^{-1}\ \top)),$$
$$(\geq 1\ C_{(0,0)}), \quad (\geq 1\ C_{(2^n-1,2^n-1)}), \quad (\leq 1\ C_{(2^n-1,2^n-1)}), \quad (\forall\ D_{\mathit{east}} \sqcap D_{\mathit{north}}),$$

where we use the following abbreviations: the expression $(\forall\ C)$ is an abbreviation for the cardinality restriction $(\leq 0\ \neg C)$, the concept $\forall R.C$ stands for

$(\leq\ 0\ R\ \neg C)$, and $\top$ stands for an arbitrary concept that is satisfied in all interpretations (e.g. $A \sqcup \neg A$).

The concept $C_{(0,0)}$ is satisfied by all elements a of the domain for which $pos(a) = (0,0)$ holds. $C_{(2^n-1,2^n-1)}$ is a similar concept, which is satisfied if $pos(a) = (2^n - 1, 2^n - 1)$:

$$C_{(0,0)} = \prod_{k=0}^{n-1} \neg X_k \sqcap \prod_{k=0}^{n-1} \neg Y_k, \quad C_{(2^n-1,2^n-1)} = \prod_{k=0}^{n-1} X_k \sqcap \prod_{k=0}^{n-1} Y_k.$$

The concept D_{east} (resp. D_{north}) enforces that along the role *east* (resp. *north*) the value of *xpos* (resp. *ypos*) increases by one while the value of *ypos* (resp. *xpos*) stays the same. They exactly resemble the formula from Lemma 2:

$$\begin{aligned} D_{east} = & \prod_{k=0}^{n-1} \Big(\prod_{j=0}^{k-1} X_j\Big) \rightarrow ((X_k \rightarrow \forall east.\neg X_k) \sqcap (\neg X_k \rightarrow \forall east.X_k)) \\ & \sqcap \prod_{k=0}^{n-1} \Big(\bigsqcup_{j=0}^{k-1} \neg X_j\Big) \rightarrow ((X_k \rightarrow \forall east.X_k) \sqcap (\neg X_k \rightarrow \forall east.\neg X_k)) \\ & \sqcap \prod_{k=0}^{n-1} ((Y_k \rightarrow \forall east.Y_k) \sqcap (\neg Y_k \rightarrow \forall east.\neg Y_k)). \end{aligned}$$

The concept D_{north} is similar to D_{east} where the role *north* has been substituted for *east* and variables X_i and Y_i have been swapped.

The following lemma is a consequence of the definition of *pos* and Lemma 2.

Lemma 3. *Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be an interpretation and $a, b \in \Delta^{\mathcal{I}}$.*

$$\begin{aligned} (a,b) \in east^{\mathcal{I}} \text{ and } a \in D_{east}^{\mathcal{I}} \text{ implies:} \quad & xpos(b) \equiv xpos(a) + 1 \pmod{2^n} \\ & ypos(b) = ypos(a) \\ (a,b) \in north^{\mathcal{I}} \text{ and } a \in D_{north}^{\mathcal{I}} \text{ implies:} \quad & xpos(b) = xpos(a) \\ & ypos(b) \equiv ypos(a) + 1 \pmod{2^n} \end{aligned}$$

The TBox T_n defines a torus of exponential size in the following sense:

Lemma 4. *Let T_n be the TBox as introduced above. Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be an interpretation such that $\mathcal{I} \models T_n$. This implies*

$$(\Delta^{\mathcal{I}}, east^{\mathcal{I}}, north^{\mathcal{I}}) \cong (U(2^n, 2^n), S_1, S_2)$$

where $U(2^n, 2^n)$ is the torus $\mathbb{Z}_{2^n} \times \mathbb{Z}_{2^n}$ and S_1, S_2 are the horizontal and vertical successor relations on the torus.

Proof. We will only sketch the proof of this lemma. It is established by showing that the function *pos* is an isomorphism from $\Delta^{\mathcal{I}}$ to $U(2^n, 2^n)$. That *pos* is a homomorphism follows immediately from Lemma 3. Injectivity of *pos* is established by showing that each element $(x, y) \in U(2^n, 2^n)$ is the image of at most

one element of $\Delta^{\mathcal{I}}$ by induction over the Manhattan distance of (x, y) to the upper right corner $(2^n - 1, 2^n - 1)$ of the torus. The base case is trivially satisfied because T_n contains the cardinality restrictions $(\leq\ 1\ C_{(2^n-1,2^n-1)})$. The induction step follows from the fact that each element $a \in \Delta^{\mathcal{I}}$ has exactly one *east*- and *north*-predecessor (since $(\forall\ (=\ 1\ \mathit{east}^{-1}\ \top)), (\forall\ (=\ 1\ \mathit{north}^{-1}\ \top)) \in T_n$) and Lemma 3. Surjectivity is established similarly starting from the corner $(0, 0)$. □

It is interesting to note that we need inverse roles only to guarantee that *pos* is injective. The same can be achieved by adding the cardinality restriction $(\leq (2^n \cdot 2^n)\ \top)$ to T_n, from which the injectivity of *pos* follows from its surjectivity and simple cardinality considerations. Of course the size of this cardinality restriction would only be polynomial in n if we allow binary coding of numbers. Also note that we have made explicit use of the special expressive power of cardinality restrictions by stating that, in any model of T_n, the extension of $C_{(2^n-1,2^n-1)}$ must have *at most* one element. This can not be expressed with a TBox consisting of terminological axioms.

3.3 Reducing Domino Problems to TBox Consistency

Once Lemma 4 has been proved, it is easy to reduce the bounded domino problem to TBox consistency. We use the standard reduction that has been applied in the DL context, e.g., in [BS99].

Lemma 5. *Let $\mathcal{D} = (D, V, H)$ be a domino system. Let $w = w_0, \ldots, w_{n-1} \in D^*$. There is a TBox $T(n, \mathcal{D}, w)$ such that:*

- *$T(n, \mathcal{D}, w)$ is consistent iff $\mathcal{D}$ tiles $U(2^n, 2^n)$ with initial condition w.*
- *$T(n, \mathcal{D}, w)$ can be computed in time polynomial in n.*

Proof. We define $T(n, \mathcal{D}, w) := T_n \cup T_{\mathcal{D}} \cup T_w$, where T_n is defined as above, $T_{\mathcal{D}}$ captures the vertical and horizontal compatibility constraints of the domino system $\mathcal{D}$, and T_w enforces the initial condition. We use an atomic concept C_d for each tile $d \in D$. $T_{\mathcal{D}}$ consists of the following cardinality restrictions:

$$(\forall \bigsqcup_{d\in D} C_d), \quad (\forall \bigsqcap_{d\in D} \bigsqcap_{d'\in D\setminus\{d\}} \neg(C_d \sqcap C_{d'})),$$

$$(\forall \bigsqcap_{d\in D} (D_d \rightarrow (\forall \mathit{east}.\ \bigsqcup_{(d,d')\in H} C_{d'}))), \quad (\forall \bigsqcap_{d\in D} (D_d \rightarrow (\forall \mathit{north}.\ \bigsqcup_{(d,d')\in V} C_{d'}))).$$

$\mathcal{T}_w$ consists of the cardinality restrictions

$$(\forall\ (C_{(0,0)} \rightarrow C_{w_0})), \ldots, (\forall\ (C_{(n-1,0)} \rightarrow C_{w_{n-1}})$$

where, for each x, y, $C_{(x,y)}$ is a concept that is satisfied by an element a iff $\mathit{pos}(a) = (x, y)$, similar to $C_{(0,0)}$ and $C_{(2^n-1,2^n-1)}$.

From the definition of $T(n, \mathcal{D}, w)$ and Theorem 4, it follows that each model of $T(n, \mathcal{D}, w)$ immediately induces a tiling of $U(2^n, 2^n)$ and vice versa. Also, for a fixed domino system $\mathcal{D}$, $T(n, \mathcal{D}, w)$ is obviously polynomially computable. □

The next theorem is an immediate consequence of Lemma 5 and Corollary 1:

Theorem 2. *Consistency of $\mathcal{ALCQI}$-TBoxes is* NExpTime*-hard, even if unary coding of numbers is used in the input.*

Recalling the note below Lemma 4, we see that the same argument also applies to $\mathcal{ALCQ}$ if we allow binary coding of numbers.

Corollary 2. *Consistency of $\mathcal{ALCQ}$-TBoxes is* NExpTime*-hard, if binary coding is used to represent numbers in cardinality restrictions.*

Note that for unary coding we needed both inverse roles and cardinality restrictions for the reduction. This is consistent with the fact that satisfiability for $\mathcal{ALCQI}$ concepts with respect to TBoxes consisting of terminological axioms is still in ExpTime, which can be shown by a reduction to Converse-PDL [GM99]. This shows that cardinality restrictions on concepts are an additional source of complexity; one reason for this might be that $\mathcal{ALCQI}$ with cardinality restrictions no longer has a tree-model property in the modal logic sense.

4 Expressiveness of $\mathcal{ALCQI}$

Since reasoning for $\mathcal{ALCQI}$ has the same (worst-case) complexity as for C^2, naturally the question arises how the two logics are related with respect to their expressivity. We show that $\mathcal{ALCQI}$ is strictly less expressive than C^2.

4.1 A Definition of Expressiveness

There are different approaches to define the expressivity of Description Logics [Baa96,Bor96,AdR98], but only the one presented in [Baa96] is capable of handling TBoxes. We will use a definition that is equivalent to the one given in [Baa96] restricted to a special case. It bases the notion of expressivity on the classes of interpretations definable by a sentence (or TBox).

Definition 3. *Let $\tau = (N_C, N_R)$ be a finite signature. A class $\mathcal{C}$ of τ-interpretations is called* characterisable *by a logic $\mathcal{L}$ iff there is a sentence $\varphi_\mathcal{C}$ over τ such that $\mathcal{C} = \{\mathcal{I} \mid \mathcal{I} \models \varphi_\mathcal{C}\}$.*

The class $\mathcal{C}$ is called projectively characterisable *iff there is a sentence $\varphi'_\mathcal{C}$ over a signature $\tau' \supseteq \tau$ such that $\mathcal{C} = \{\mathcal{I}|_\tau \mid \mathcal{I} \models \varphi'_\mathcal{C}\}$, where $\mathcal{I}|_\tau$ denotes the τ-reduct of $\mathcal{I}$.*

A logic $\mathcal{L}_1$ is called as expressive *as another logic $\mathcal{L}_2$ ($\mathcal{L}_1 \geq \mathcal{L}_2$) iff, for any finite signature τ, any $\mathcal{L}_2$-characterisable class $\mathcal{C}$ can be projectively characterised in $\mathcal{L}_1$.*

Since C^2 is usually restricted to a relational signature with relation symbols of arity at most two, this definition is appropriate to relate the expressiveness of $\mathcal{ALCQI}$ and C^2. It is worth noting that $\mathcal{ALCQI}$ is strictly more expressive

than $\mathcal{ALCQ}$, because $\mathcal{ALCQ}$ has the finite model property [BBH96], while the following $\mathcal{ALCQI}$ TBox has no finite models:

$$T_{\text{inf}} = \{(\forall\ (\geq\ 1\ R\ \top)), (\forall\ (\leq\ 1\ R^{-1}\ \top)), (\geq\ 1\ (=\ 0\ R^{-1}\ \top))\}.$$

The first cardinality restriction requires an outgoing R-edge for every element of a model and thus each R-path in the model in infinite. The second and third restriction require the existence of an R-path in the model that contains no cycle, which implies the existence of infinitely many elements in the model. Since $\mathcal{ALCQ}$ has the finite model property, the class $\mathcal{C}_{\text{inf}} := \{\mathcal{I} \mid \mathcal{I} \models \mathcal{T}_{\text{inf}}\}$, which contains only models with infinitely many elements, can not be projectively characterised by an $\mathcal{ALCQ}$-TBox.

The translation Ψ from $\mathcal{ALCQI}$-TBoxes to C^2 sentences given in Fig. 1 not only preserves satisfiability, but the translation also has exactly the same models as the initial TBox. This implies that $\mathcal{ALCQI} \leq C^2$.

4.2 A Game for $\mathcal{ALCQI}$

Usually, the separation of two logics with respect to their expressivity is a hard task and not as easily accomplished as we have just done with $\mathcal{ALCQ}$ and $\mathcal{ALCQI}$. Even for logics of very restricted expressivity, proofs of separation results may become involved and complex [Baa96] and usually require a detailed analysis of the classes of models a logic is able to characterise. Valuable tools for these analyses are Ehrenfeucht-Fraïssé games. In this section we present an Ehrenfeucht-Fraïssé game that exactly captures the expressivity of $\mathcal{ALCQI}$.

Definition 4. *For an $\mathcal{ALCQI}$ concept C, the* role depth $rd(C)$ *counts the maximum number of nested cardinality restrictions. Formally we define rd as follows:*

$$\begin{aligned} rd(A) &:= 0 \quad \text{for } A \in N_C \\ rd(\neg C) &:= rd(C) \\ rd(C_1 \sqcap C_2) &:= max\{rd(C_1), rd(C_2)\} \\ rd(\geq n\ R\ C) &:= 1 + rd(C) \end{aligned}$$

The set $\mathcal{C}_m^n$ is defined to consist of exactly those $\mathcal{ALCQI}$ concepts that have a role depth of at most m, and in which the numbers appearing in number restrictions are bounded by n; the set $\mathcal{L}_m^n$ is defined to consist of all $\mathcal{ALCQI}$-TBoxes T that contain only cardinality restrictions of the form $(\bowtie\ k\ C)$ with $k \leq n$ and $C \in \mathcal{C}_m^n$.

Two interpretations $\mathcal{I}$ and $\mathcal{J}$ are called n-m-equivalent *($\mathcal{I} \equiv_m^n \mathcal{J}$) iff, for all TBoxes T in $\mathcal{L}_m^n$, it holds that $\mathcal{I} \models T$ iff $\mathcal{J} \models T$. Similarly, for $x \in \Delta^{\mathcal{I}}$ and $y \in \Delta^{\mathcal{J}}$ we say that $\mathcal{I}, x$ and $\mathcal{J}, y$ are* n-m-equivalent *($\mathcal{I}, x \equiv_m^n \mathcal{J}, y$) iff, for all $C \in \mathcal{C}_m^n$ it holds that, $x \in C^{\mathcal{I}}$ iff $y \in C^{\mathcal{J}}$.*

Two elements $x \in \Delta^{\mathcal{I}}$ and $y \in \Delta^{\mathcal{J}}$ are called locally equivalent *($\mathcal{I}, x \equiv_l \mathcal{J}, y$), iff for all $A \in N_C$: $x \in A^{\mathcal{I}}$ iff $y \in A^{\mathcal{J}}$.*

Note that, since we assume τ to be finite, there are only finitely many pairwise inequivalent concepts in each class $\mathcal{C}_m^n$.

We will now define an Ehrenfeucht-Fraisse game for $\mathcal{ALCQI}$ to capture the expressivity of concepts in the classes $\mathcal{C}_m^n$: The game is played by two players. Player I is called the spoiler while Player II is called the duplicator. The spoiler's aim is to prove two structures not to be n-m-equivalent, while Player II tries to prove the contrary. The game consists of a number of rounds in which the players move pebbles on the elements of the two structures.

Definition 5. *Let Δ be a nonempty set. Let x be an element of Δ and X a subset of Δ. For any binary relation $\mathcal{R} \subseteq \Delta \times \Delta$ we write $x\mathcal{R}X$ to denote the fact that $(x, x') \in \mathcal{R}$ holds for all $x' \in X$. For the set N_R of role names let $\overline{N_R}$ be the union of N_R and $\{R^{-1} \mid R \in N_R\}$.*

A configuration *captures the state of a game in progress. It is of the form $G_m^n(\mathcal{I}, x, \mathcal{J}, y)$, where $n \in \mathbb{N}$ is a limit on the size of set that may be chosen during the game, m denotes the number of moves which still have to be played, and x and y are the elements of $\Delta^{\mathcal{I}}$ resp. $\Delta^{\mathcal{J}}$ on which the pebbles are placed.*

For the configuration $G_m^n(\mathcal{I}, x, \mathcal{J}, y)$ the rules are as follows:

1. *If $\mathcal{I}, x \not\equiv_l \mathcal{J}, y$, then Player II loses; if $m = 0$ and $\mathcal{I}, x \equiv_l \mathcal{J}, y$, then Player II wins.*
2. *If $m > 0$, then Player I selects one of the interpretations; assume this is $\mathcal{I}$ (the case $\mathcal{J}$ is handled dually). He then picks a role $S \in \overline{N_R}$ and a number $l \leq n$. He picks a set $X \subseteq \Delta^{\mathcal{I}}$ such that $xS^{\mathcal{I}}X$ and $\sharp X = l$. The duplicator has to answer with a set $Y \subseteq \Delta^{\mathcal{J}}$ with $yS^{\mathcal{J}}Y$ and $\sharp Y = l$. If there is no such set, then she loses.*
3. *If Player II was able to pick such a set Y, then Player I picks an element $y' \in Y$. Player II has to answer with an element $x' \in X$.*
4. *The game continues with $G_{m-1}^n(\mathcal{I}, x', \mathcal{J}, y')$.*

We say that Player II has a winning strategy *for $G_m^n(\mathcal{I}, x, \mathcal{J}, y)$ iff she can always reach a winning position no matter which moves Player I plays. We write $\mathcal{I}, x \cong_m^n \mathcal{J}, y$ to denote this fact.*

Theorem 3. *For two structures $\mathcal{I}, \mathcal{J}$ and two elements $x \in \Delta^{\mathcal{I}}, y \in \Delta^{\mathcal{J}}$ it holds that $\mathcal{I}, x \cong_m^{n+1} \mathcal{J}, y$ iff $\mathcal{I}, x \equiv_m^n \mathcal{J}, y$.*

We omit the proof of this and the next theorem. These employ the same techniques that are used to show the appropriateness of the known Ehrenfeucht-Fraïssé games for C^2 and for modal logics, please refer to [Tob99] for details.

The game as it has been presented so far is suitable only if we have already placed pebbles on the interpretations. To obtain a game that characterises $\equiv_m^n$ as a relation between interpretations, we have to introduce an additional rule that governs the placement of the first pebbles. Since a TBox consists of cardinality restrictions which solely talk about concept membership, we introduce an unconstrained set move as the first move of the game $G_m^n(\mathcal{I}, \mathcal{J})$.

Definition 6. *For two interpretations $\mathcal{I}, \mathcal{J}$, $G_m^n(\mathcal{I}, \mathcal{J})$ is played as follows:*

1. *Player I picks one of the structures; assume he picks $\mathcal{I}$ (the case $\mathcal{J}$ is handled dually). He then picks a set $X \subseteq \Delta^{\mathcal{I}}$ with $\sharp X = l$ where $l \leq n$. Player II must pick a set $Y \subseteq \Delta^{\mathcal{J}}$ of equal size. If this is impossible then she loses.*
2. *Player I picks an element $y \in Y$, Player II must answer with an $x \in X$.*
3. *The game continues with $G^n_m(\mathcal{I}, x, \mathcal{I}, y)$.*

Again we say that Player II has a winning strategy for $G^n_m(\mathcal{I}, \mathcal{J})$ iff she can always reach a winning positions no matter which moves Player I chooses. We write $\mathcal{I} \cong^n_m \mathcal{J}$ do denote this fact.

Theorem 4. *For two structures $\mathcal{I}, \mathcal{J}$ it holds that $\mathcal{I} \equiv^n_m \mathcal{J}$ iff $\mathcal{I} \cong^{n+1}_m \mathcal{J}$.*

Similarly, it would be possible to define a game that captures the expressivity of $\mathcal{ALCQI}$ with TBoxes consisting of terminological axioms by replacing the unconstrained set move from Def. 6 by a move where Player I picks a structure and one element from that structure; Player II then has to answer accordingly and the game continues as described in Def. 5.

4.3 The Expressivity Result

We will now use this characterisation of the expressivity of $\mathcal{ALCQI}$ to prove that $\mathcal{ALCQI}$ is less expressive than C^2. Even though we have introduced the powerful tool of Ehrenfeucht-Fraïssé games, the proof is still rather complicated. This is mainly due to the fact that we use a general definition of expressiveness that allows for the introduction of arbitrary additional role- and concept-names into the signature.

Theorem 5. *$\mathcal{ALCQI}$ is not as expressive as C^2.*

Proof. To prove this theorem we have to show that there is a class $\mathcal{C}$ that is characterisable in C^2 but that cannot be projectively characterised in $\mathcal{ALCQI}$:
Claim 1: For an arbitrary $R \in N_R$ the class $\mathcal{C}_R := \{\mathcal{I} \mid R^{\mathcal{I}} \text{ is reflexive}\}$ is not projectively characterisable in $\mathcal{ALCQI}$. Obviously, $\mathcal{C}_R$ is characterisable in C^2.
Proof of Claim 1: Assume Claim 1 does not hold and that $\mathcal{C}_R$ is projectively characterised by the TBox $\mathcal{T}_R \in \mathcal{L}^n_m$ over an arbitrary (but finite) signature $\tau = (N_C, N_R)$ with $R \in N_R$. We will have derived a contradiction once we have shown that there are two τ-interpretations $\mathcal{A}, \mathcal{B}$ such that $\mathcal{A} \in \mathcal{C}_R$, $\mathcal{B} \notin \mathcal{C}_R$, but $\mathcal{A} \equiv^n_m \mathcal{B}$. In fact, $\mathcal{A} \equiv^n_m \mathcal{B}$ implies $\mathcal{B} \models \mathcal{T}_R$ and hence $\mathcal{B} \in \mathcal{C}_R$, a contradiction.

In particular, $\mathcal{C}_R$ contains all interpretations $\mathcal{A}$ with $R^{\mathcal{A}} = \{(x, x) \mid x \in \Delta^{\mathcal{A}}\}$, i.e. interpretations in which R is interpreted as equality. Since $\mathcal{C}^n_m$ contains only finitely many pairwise inequivalent concepts and $\mathcal{C}_R$ contains interpretations of arbitrary size, there is also such an $\mathcal{A}$ such that there are two elements $x_1, x_2 \in \Delta^{\mathcal{A}}$ with $x_1 \neq x_2$ and $\mathcal{A}, x_1 \equiv^n_m \mathcal{A}, x_2$. We define $\mathcal{B}$ from $\mathcal{A}$ as follows:

$$\begin{aligned}
\Delta^{\mathcal{B}} &:= \Delta^{\mathcal{A}}, \\
A^{\mathcal{B}} &:= A^{\mathcal{A}} \quad \text{for each } A \in N_C, \\
S^{\mathcal{B}} &:= S^{\mathcal{A}} \quad \text{for each } S \in N_R \setminus \{R\}, \\
R^{\mathcal{B}} &:= (R^{\mathcal{A}} \setminus \{(x_1, x_1), (x_2, x_2)\}) \cup \{(x_1, x_2), (x_2, x_1)\}.
\end{aligned}$$

Since $R^{\mathcal{B}}$ is no longer reflexive, as desired $\mathcal{B} \notin \mathcal{C}_R$ holds. It remains to be shown that $\mathcal{A} \equiv_m^n \mathcal{B}$ holds. We prove this by showing that $\mathcal{A} \cong_m^{n+1} \mathcal{B}$ holds, which is equivalent to $\mathcal{A} \equiv_m^n \mathcal{B}$ by Theorem 4.

Any opening move of Player I can be answered by Player II in a way that leads to the configuration $G_m^{n+1}(\mathcal{A}, x, \mathcal{B}, x)$, where x depends on the choices of Player I. We have to show that, for any configuration of this type, Player II has a winning strategy. Since certainly $\mathcal{A}, x \cong_m^{n+1} \mathcal{A}, x$ this follows from Claim 2:

Claim 2: For all $k \leq m$: If $\mathcal{A}, x \cong_k^{n+1} \mathcal{A}, y$ then $\mathcal{A}, x \cong_k^{n+1} \mathcal{B}, y$.

Proof of Claim 2: We prove Claim 2 by induction over k. Denote Player II's strategy for the configuration $G_k^{n+1}(\mathcal{A}, x, \mathcal{A}, y)$ by S.

For $k = 0$, Claim 2 follows immediately from the construction of $\mathcal{B}$: $\mathcal{A}, x \cong_0^{n+1} \mathcal{A}, y$ implies $\mathcal{A}, x \equiv_l \mathcal{A}, y$ and $\mathcal{A}, y \equiv_l \mathcal{B}, y$ since $\mathcal{B}$ agrees with $\mathcal{A}$ on the interpretation of all atomic concepts. It follows that $\mathcal{A}, x \equiv_l \mathcal{B}, y$, which means that Player II wins the game $G_0^{n+1}(\mathcal{A}, x, \mathcal{B}, y)$. For $0 < k \leq m$, assume that Player I selects an arbitrary structure and a legal subset of the respective domain. Player II tries to answer that move according to S which provides her with a move for the game $G_k^{n+1}(\mathcal{A}, x, \mathcal{A}, y)$. There are two possibilities:

- The move provided by S is a valid move also for the game $G_k^{n+1}(\mathcal{A}, x, \mathcal{B}, y)$: Player II can answer the choice of Player I according to S without violating the rules, which yields a configuration $G_{k-1}^{n+1}(\mathcal{A}, x', \mathcal{B}, y')$ such that for x', y' it holds that $\mathcal{A}, x' \cong_{k-1}^{n+1} \mathcal{A}, y'$ (because Player II moved according to S). From the induction hypothesis it follows that $\mathcal{A}, x' \cong_{k-1}^{n+1} \mathcal{B}, y'$.
- The move provided by S is not a valid move for the game $G_k^{n+1}(\mathcal{A}, x, \mathcal{B}, y)$ This requires a more detailed analysis: Assume Player I has chosen to move in $\mathcal{A}$ and has chosen an $S \in \overline{N_R}$ and a set X of size $l \leq n+1$ such that $xS^{\mathcal{A}}X$. Let Y be the set that Player II would choose according S. This implies that Y has also l elements and that $yS^{\mathcal{A}}Y$. That this choice is not valid in the game $G_k^{n+1}(\mathcal{A}, x, \mathcal{B}, y)$ implies that there is an element $z \in Y$ such that $(y, z) \notin S^{\mathcal{B}}$. This implies $y \in \{x_1, x_2\}$ and $S \in \{R, R^{-1}\}$, because these are the only elements and relations that are different in $\mathcal{A}$ and $\mathcal{B}$. W.l.o.g. assume $y = x_1$ and $S = R$. Then also $z = x_1$ must hold, because this is the only element such that $(x_1, z) \in R^{\mathcal{A}}$ and $(x_1, z) \notin R^{\mathcal{B}}$. Thus, the choice $Y' := (Y \setminus \{x_1\}) \cup \{x_2\}$ is a valid one for Player II in the game $G_m^{n+1}(\mathcal{A}, x, \mathcal{B}, y)$: $x_1 R^{\mathcal{B}} Y'$ and $|Y'| = l$ because $(x_1, x_2) \notin R^{\mathcal{A}}$.
  There are two possibilities for Player I to choose an element $y' \in Y'$:
  1. $y' \neq x_2$: Player II chooses $x' \in X$ according to S. This yields a configuration $G_{k-1}^{n+1}(\mathcal{A}, x', \mathcal{B}, y')$ such that $\mathcal{A}, x' \cong_{k-1}^{n+1} \mathcal{A}, y'$.
  2. $y' = x_2$: Player II answers with the $x' \in X$ that is the answer to the move x_1 of Player I according to S. For the obtained configuration $G_{k-1}^{n+1}(\mathcal{A}, x', \mathcal{B}, y')$ also $\mathcal{A}, x' \cong_{k-1}^{n+1} \mathcal{A}, y'$ holds: By the choice of x_1, x_2, $\mathcal{A}, x_1 \equiv_m^n \mathcal{A}, x_2$ is satisfied and since $k - 1 < m$ also $\mathcal{A}, x_1 \equiv_{k-1}^n \mathcal{A}, x_2$ holds which implies $\mathcal{A}, x_1 \cong_{k-1}^{n+1} \mathcal{A}, x_2$ by Theorem 4. Since Player II chose x' according to S it holds that $\mathcal{A}, x' \cong_{k-1}^{n+1} \mathcal{A}, x_1$ and hence $\mathcal{A}, x' \cong_{k-1}^{n+1} \mathcal{A}, x_2$ since $\cong_{k-1}^{n+1}$ is transitive.

In both cases we can apply the induction hypothesis which yields $\mathcal{A}, x' \cong_{k-1}^{n+1} \mathcal{B}, y'$ and hence Player II has a winning strategy for $G_k^{n+1}(\mathcal{A}, x, \mathcal{B}, y)$. The case that Player I chooses from $\mathcal{B}$ instead of $\mathcal{A}$ can be handled dually. □

By adding constructs to $\mathcal{ALCQI}$ that allow to form more complex role expressions one can obtain a DL that has the same expressive power as C^2, such a DL is presented in [Bor96]. The logic presented there has the ability to express a universal role that makes it possible to internalise both TBoxes based on terminological axioms and cardinality restrictions on concepts.

5 Conclusion

We have shown that, with a rather limited set of constructors, one can define a DL whose reasoning problems are as hard as those of C^2 without reaching the expressive power of the latter. This shows that cardinality restrictions, although interesting for knowledge representation, are inherently hard to handle algorithmically. At a first glance, this makes $\mathcal{ALCQI}$ with cardinality restrictions on concepts obsolete for knowledge representation, because C^2 delivers more expressive power at the same computational price. Yet, is is likely that a dedicated algorithm for $\mathcal{ALCQI}$ may have better average complexity than the C^2 algorithm; such an algorithm has yet to be developed. An interesting question lies in the coding of numbers: If we allow binary coding of numbers, the translation approach together with the result from [PST97] leads to a 2-NExpTime algorithm. As for C^2, it is an open question whether this additional exponential blow-up is necessary. A positive answer would settle the same question for C^2 while a proof of the negative answer might give hints how the result for C^2 might be improved.

Acknowledgments. I would like to thank Franz Baader, Ulrike Sattler, and Eric Rosen for valuable comments and suggestions.

References

[AdR98] C. Areces and M. de Rijke. Expressiveness revisited. In *Proceedings of DL'98*, 1998.

[Baa91] F. Baader. Augmenting concept languages by transitive closure of roles: An alternative to terminological cycles. In *Proceedings of IJCAI-91*, pages 446–451, 1991.

[Baa96] F. Baader. A formal definition for the expressive power of terminological knowledge representation language. *J. of Logic and Computation*, 6(1):33–54, 1996.

[BBH96] F. Baader, M. Buchheit, and B. Hollunder. Cardinality restrictions on concepts. *Artificial Intelligence*, 88(1–2):195–213, 1996.

[Ber66] R. Berger. The undecidability of the dominoe problem. *Memoirs of the American Mathematical Society*, 66, 1966.

[BGG97] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer-Verlag, Berlin, 1997.

[Bor96] A. Borgida. On the relative expressiveness of description logics and first order logics. *Artificial Intelligence*, 82:353–367, 1996.

[BPS94] A. Borgida and P. Patel-Schneider. A semantics and complete algorithm for subsumption in the CLASSIC description logic. *Journal of Artificial Intelligence Research*, 1:277–308, 1994.

[BS99] F. Baader and U. Sattler. Expressive number restrictions in description logics. *Journal of Logic and Computation*, 9, 1999, to appear.

[DL94] G. De Giacomo and M. Lenzerini. Description logics with inverse roles, functional restrictions, and N-ary relations. In C. MacNish, L. M. Pereira, and D. Pearce, editors, *Logics in Artificial Intelligence*, pages 332–346. Springer-Verlag, Berlin, 1994.

[DLNN97] F. M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. *Information and Computation*, 134(1):1–58, 1997.

[GL96] G. De Giacomo and M. Lenzerini. TBox and ABox reasoning in expressive description logics. In *Proceeding of KR'96*, 1996.

[GM99] G. De Giacomo and F. Massacci. Combining deduction and model checking into tableaux and algorithms for converse-PDL. To appear in *Information and Computation*, 1999.

[GOR97] E. Grädel, M. Otto, and E. Rosen. Two-variable logic with counting is decidable. In *Proceedings of LICS 1997*, pages 306–317, 1997.

[HB91] B. Hollunder and F. Baader. Qualifying number restrictions in concept languages. In *Proceedings of KR'91*, pages 335–346, Boston (USA), 1991.

[OSH96] H. J. Ohlbach, R. A. Schmidt, and U. Hustadt. Translating graded modalities into predicate logic. In H. Wansing, editor, *Proof Theory of Modal Logic*, volume 2 of *Applied Logic Series*, pages 253–291. Kluwer, 1996.

[PST97] L. Pacholski, W. Szwast, and L. Tendera. Complexity of two-variable logic with counting. In *Proceedings of LICS 1997*, pages 318–327, 1997.

[Spa93] E. Spaan. *Complexity of Modal Logics*. PhD thesis, University of Amsterdam, 1993.

[SSS91] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48:1–26, 1991.

[Tob99] S. Tobies. A Nexptime-complete description logic strictly contained in C^2. LTCS-Report 99-05, LuFg Theoretical Computer Science, RWTH Aachen, Germany, 1999. See http://www-lti.informatik.rwth-aachen.de/Forschung/Papers.html.

[Wan63] H. Wang. Dominoes and the AEA case of the Decision Problem. *Bell Syst. Tech. J.*, 40:1–41, 1963.

[WS92] W. A. Woods and J. G. Schmolze. The KL-ONE family. *Computers and Mathematics with Applications – Special Issue on Artificial Intelligence*, 23(2–5):133–177, 1992.

A Road-Map on Complexity for Hybrid Logics

Carlos Areces[1], Patrick Blackburn[2], and Maarten Marx[3]

[1] ILLC. UvA. Plantage Muidergracht 24, 1018TV. The Netherlands.
phone: +31 (20) 525-6925 e-mail: carlos@wins.uva.nl
[2] Computerlinguistik. Universität des Saarlandes. D-66041 Saarbrücken. Germany.
phone: +49 (681) 302-4501 e-mail: patrick@coli.uni-sb.de
[3] Department of Artificial Intelligence, Free University, Amsterdam, and
ILLC. UvA. Plantage Muidergracht 24, 1018TV. The Netherlands.
phone: +31 (20) 525-2459 e-mail: marx@wins.uva.nl

Abstract. Hybrid languages are extended modal languages which can refer to (or even quantify over) states. Such languages are better behaved proof theoretically than ordinary modal languages for they internalize the apparatus of labeled deduction. Moreover, they arise naturally in a variety of applications, including description logic and temporal reasoning. Thus it would be useful to have a map of their complexity-theoretic properties, and this paper provides one.
Our work falls into two parts. We first examine the basic hybrid language and its multi-modal and tense logical cousins. We show that the basic hybrid language (and indeed, multi-modal hybrid languages) are no more complex than ordinary uni-modal logic: all have PSPACE-complete K-satisfiability problems. We then show that adding even one nominal to tense logic raises complexity from PSPACE to EXPTIME. In the second part we turn to stronger hybrid languages in which it is possible to *bind* nominals. We prove a general expressivity result showing that even the weak form of binding offered by the $\downarrow$ operator easily leads to undecidability.

Keywords. Computational Complexity, Modal and Temporal Logic, Description Logic, Labeled Deduction.

1 Introduction

Hybrid languages are modal languages which use atomic formulas called nominals to name states. Nominals are true at exactly one state in any model; they "name" this state by being true there and nowhere else. Although a wide range of hybrid languages have been studied, including hybrid languages in which it is possible to bind nominals in various ways, little is known about their computational complexity. This paper is an attempt to fill the gap.

Before going further, let's be precise about the syntax and semantics of the *basic hybrid language* $\mathcal{H}(@)$, the weakest language we shall consider in the paper.

Definition 1 (Syntax). Let PROP $= \{p, q, r, \ldots\}$ be a countable set of *propositional variables* and NOM $= \{i, j, k, \ldots\}$ a countable set of *nominals*, disjoint

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 307–321, 1999.

from PROP. We call ATOM = PROP ∪ NOM the set of *atoms*. The *well-formed formulas* of the hybrid language (over ATOM) are

$$\varphi := a \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \Box\varphi \mid \ @_i\varphi$$

where $a \in$ ATOM, and $i \in$ NOM. As usual, $\Diamond\varphi$ is defined to be $\neg\Box\neg\varphi$. A formula which contains no symbols from PROP is called *pure*.

Thus, syntactically speaking, the basic hybrid language is a two-sorted uni-modal language which contains a NOM indexed collection of operators $@_i$. Now for the semantics.

Definition 2 (Semantics). A (hybrid) *model* $\mathfrak{M}$ is a triple $\mathfrak{M} = \langle M, R, V\rangle$ such that M is a non-empty set, R is a binary relation on M, and V : ATOM $\rightarrow$ Pow(M) is such that for all $i \in$ NOM, $V(i)$ is a singleton subset of M. We usually call the elements of M states, R is the *transition relation*, and V is the *valuation*. A *frame* is a pair $\mathfrak{F} = \langle M, R\rangle$, that is, a model without a valuation.

Let $\mathfrak{M} = \langle M, R, V\rangle$ be a model and $m \in M$. Then the *satisfaction relation* is defined by:

$$\begin{array}{ll}
\mathfrak{M}, m \Vdash a & \text{iff } m \in V(a),\ a \in \text{ATOM} \\
\mathfrak{M}, m \Vdash \neg\varphi & \text{iff } \mathfrak{M}, m \nVdash \varphi \\
\mathfrak{M}, m \Vdash \varphi \wedge \psi & \text{iff } \mathfrak{M}, m \Vdash \varphi \text{ and } \mathfrak{M}, m \Vdash \psi \\
\mathfrak{M}, m \Vdash \Box\varphi & \text{iff } \forall m'(Rmm' \Rightarrow \mathfrak{M}, m' \Vdash \varphi) \\
\mathfrak{M}, m \Vdash @_i\varphi & \text{iff } \mathfrak{M}, m' \Vdash \varphi, \text{ where } V(i) = \{m'\},\ i \in \text{NOM}.
\end{array}$$

A formula φ is *satisfiable* if there is a model $\mathfrak{M}$, and a state $m \in M$ such that $\mathfrak{M}, m \Vdash \varphi$. We write $\mathfrak{M} \models \varphi$ iff for all $m \in M$, $\mathfrak{M}, m \Vdash \varphi$. If $\mathfrak{F}$ is a frame, and for all valuations V on $\mathfrak{F}$ we have $\langle \mathfrak{F}, V\rangle \models \varphi$, we say φ is *valid* on $\mathfrak{F}$ and write $\mathfrak{F} \models \varphi$.

Because valuations assign *singletons* to nominals, it is clear that each nominal is satisfied at exactly one state in any model. And the clause for formulas of the form $@_i\varphi$ simply says: to evaluate $@_i\varphi$, jump to the unique state named by i and evaluate φ there.

There are at least two reasons for being interested in hybrid languages. First, they can be seen as modal languages which internalize the ideas underlying labeled deduction systems. Second, hybrid languages arise naturally in many applications.

Hybrid languages and labeled deduction Labeled deduction (see [Gab96]) is built around the notation $l{:}\varphi$. Here the meta linguistic symbol : associates the meta linguistic label l with the formula φ. This has a natural modal interpretation: regard labels as names for states and read $l{:}\varphi$ as asserting that φ is satisfied at l. Labeled deduction proceeds by manipulating such labels to guide proof search; the approach has become an important way of handling modal proof theory.

The basic hybrid language places the apparatus of labeled deduction in the *object* language: nominals are essentially object-level labels, and the formula $@_i\varphi$

asserts in the object language what $i:\varphi$ asserts in the metalanguage. And indeed, hybrid languages turn out to be proof-theoretically well behaved. For a start, the basic hybrid language enables us to directly "internalize" labeled deduction (see [Bla98]), and to define sequent calculi and natural deduction systems (see [Sel97]). In fact, even if @ is dropped from the language, elegant Fitting-style systems which exploit the presence of nominals can be defined (see [Tza98]).

Furthermore, such calculi *automatically* handle the logics of a wide range of frame classes, including many that are awkward for ordinary modal logic. To give a simple example, no ordinary modal formula defines irreflexivity (that is, no ordinary modal formula is valid on precisely the irreflexive frames). But the (pure) formula $@_i\neg\Diamond i$ does so, as the reader can easily check. Moreover, when used as an additional axiom, this formula (and indeed, *any* pure formula) is complete with respect to the class of frames it defines. For a full discussion of these issues, see [Bla98,BT99].

Hybrid languages and applied logic Modal logicians like to claim that notational variants of modal logics are often reinvented by workers in artificial intelligence, computational linguistics, and other fields — in this case, it would be more accurate to say that it is *hybrid languages* which are reinvented in this way. For example, it is well known that the description language $\mathcal{ALC}$ (see [SSS91]) is a notational variant of multi-modal logic (see [Sch91]). But this relation is established at the level of what is called the TBox reasoning. TBox reasoning is complemented with ABox assertions, which corresponds to the addition of nominals (see [AdR99,BS98]). Moreover, many authors have pointed out how natural hybrid languages are for temporal reasoning (see [Bul70,Gor96,BT99]). Among other things, hybrid languages make it possible to introduce specific times (days, dates, etc.), and to define many temporally relevant frame properties (such as irreflexivity, asymmetry, trichotomy, and directedness) that ordinary modal languages cannot handle. Furthermore, if one starts with a modal interval language and adds nominals and @, one obtains variants of the $\mathsf{Holds}(t,\varphi)$-driven interval logics discussed in [All84] (with @ playing the role of Holds).

The emergence of hybrid languages in applied logic is not particularly surprising. Modal languages offer a simple notation for modeling many problems — but the ability to reason about what happens at a *particular* state is often important and this is precisely what orthodox modal languages lack. This seems to have encouraged a drift (often implicit) towards hybrid languages.

Our work falls into two parts. We first examine the basic hybrid language and its multi-modal and tense logical variants. We show that the basic and even the multi-modal hybrid languages are no more complex than ordinary uni-modal logic: all have PSPACE-complete K-satisfiability problems. We also show that adding even one nominal to tense logic raises complexity from PSPACE to EXPTIME. In the second part of the paper we turn to stronger hybrid languages in which it is possible to *bind* nominals. We shall show, via a general expressivity result called the *Spypoint Theorem*, that even the restricted form of binding offered by the $\downarrow$ operator easily leads to undecidability.

2 Complexity of the basic hybrid language

We begin with a positive result. We know from [Lad77] that ordinary propositional uni-modal logic has a PSPACE-complete K-satisfaction problem (the K meaning that no restrictions are placed on the transition relation R). What happens when we add nominals and @ to form the basic hybrid language? The answer (up to a polynomial) is: *nothing*.

Theorem 1. *The* K*-satisfaction problem for the basic hybrid language is* PSPACE*-complete.*

Proof. The lower bound follows from [Lad77]. We show the upper bound by defining the notion of a ξ-game between two players. We will show that the existential player has a winning strategy for the ξ-game if and only if ξ is satisfiable. Moreover every ξ-game stops after at most as many rounds as the modal depth of ξ and the information on the playing board is polynomial in the length of ξ. Using the close correspondence between Alternating Turing Machines (ATM's) and two player games [Chl86], it is straightforward to implement the problem of whether the existential player has a winning strategy in the ξ-game on a PTIME ATM. Because any PTIME ATM algorithm can be turned into a PSPACE Turing Machine program, we obtain our desired result. We present the proof only for uni-modal $\mathcal{H}(@)$; it can be straightforwardly extended to the multi-modal case.

Fix a formula ξ. A ξ-Hintikka set is a maximal consistent set of subformulas of ξ. We denote the set of subformulas of ξ by $SF(\xi)$. The ξ-game is played as follows. There are two players, ∀belard (male) and ∃loise (female). She starts the game by playing a collection $\{X_0, \ldots, X_k\}$ of Hintikka sets and specifying a relation R on them.

∃loise loses immediately if one of the following conditions is false:

1. X_0 contains ξ, and all others X_l contain at least one nominal occurring in ξ.
2. no nominal occurs in two different Hintikka sets.
3. for all X_l, for all $@_i\varphi \in SF(\xi)$, $@_i\varphi \in X_l$ iff $\{i, \varphi\} \subseteq X_k$, for some k.
4. for all $\Diamond\varphi \in SF(\xi)$, if RX_lX_k and $\Diamond\varphi \notin X_l$, then $\varphi \notin X_k$.

Now ∀belard may choose an X_l and a "defect-formula" $\Diamond\varphi \in X_l$. ∃loise must respond with a Hintikka set Y such that

1. $\varphi \in Y$ and for all $\Diamond\psi \in SF(\xi)$, $\Diamond\psi \notin X_l$ implies that $\psi \notin Y$.
2. for all $@_i\varphi \in SF(\xi)$, $@_i\varphi \in Y$ iff $\{i, \varphi\} \subseteq X_k$, for some k.
3. if $i \in Y$ for some nominal i, then Y is one of the Hintikka sets she played at the start. In this case the game stops and ∃loise wins.

If ∃loise cannot find a suitable Y, the game stops and ∀belard wins. If ∃loise does find a suitable Y (one that is not covered by the halting clause in item 3 above) then Y is added to the list of played sets, and play continues.

∀belard must now choose a defect $\Diamond\varphi$ from the last played Hintikka set with the following restriction: in round k he can only choose defects $\Diamond\varphi$ such that

the modal depth of $\Diamond\varphi$ is less than or equal to the modal depth of ξ minus k. $\exists$loise must respond as before. She wins if she can survive all his challenges (in other words, he loses if he reaches a situation where he can't choose any more defects).

It is clear that the ξ-game stops after at most modal depth of ξ many rounds. The size of the information on the board is at any stage of the game polynomial in the length of ξ, as Hintikka sets are polynomial in the length of ξ and ξ can only contain polynomially many nominals. We claim that $\exists$loise has a winning strategy iff ξ is satisfiable.

Now the right-to-left direction is clear: $\exists$loise has a winning strategy if ξ is satisfiable, for she need simply play by reading the required Hintikka sets off the model. The other direction requires more work. Suppose $\exists$loise has a winning strategy for the ξ-game. We shall create a model $\mathfrak{M}$ for ξ as follows. The domain M is build in steps by following her winning strategy. M_0 consists of her initial move $\{X_0, \ldots, X_n\}$. Suppose M_j is defined. Then M_{j+1} consists of a copy of those Hintikka sets she plays when using her winning strategy for each of $\forall$belard's possible moves played in the Hintikka sets from M_j (except when she plays a Hintikka set from her initial move, then of course we do not make a copy). Let M be the disjoint union of all M_j for j smaller than the modal depth of ξ. Set Rmm' iff for all $\Diamond\varphi \in SF(\xi)$, $\Diamond\varphi \notin m \Rightarrow \varphi \notin m'$ holds, and set $V(p) = \{m \in M \mid p \in m\}$. Note that the rules of the game guarantee that nominals are interpreted as singletons.

We claim that the following truth-lemma holds. For all $m \in M$ which she plays in round j (i.e., $m \in M_j$), for all φ of modal depth less than or equal to the modal depth of ξ minus j, $\mathfrak{M}, m \Vdash \varphi$ if and only if $\varphi \in m$.

Proof of Claim. By induction on the structure of formulas. For atoms, the booleans and @ the proof is easy. For $\Diamond$, if $\Diamond\varphi \in m$, then $\forall$belard challenged this defect, so $\exists$loise could respond with an m' containing φ. Since for all $\Diamond\varphi \in SF(\xi)$, $\Diamond\varphi \notin m \Rightarrow \varphi \notin m'$ holds, we have Rmm' and by induction hypothesis $\mathfrak{M}, m \Vdash \Diamond\varphi$. If $\Diamond\varphi \notin m$ but Rmm' holds, then by our definition of R, $\varphi \notin m'$, so again $\mathfrak{M}, m \nVdash \Diamond\varphi$.

Since she plays a Hintikka set containing ξ in the first round, $\mathfrak{M}$ satisfies ξ.

This result generalizes to the multi-modal case. Recall that in a multi-modal language we have an indexed collection of modalities $[\alpha]$, each interpreted by some relation R_α. From [HM92] we know that the K-satisfaction problem for multi-modal languages is PSPACE-complete (here the K means that no restrictions are placed on the individual R_α, or on the way they are inter-related). If we add nominals and @ to such a language, the previous proof straightforwardly extends to show that we are still PSPACE-complete.

We have already mentioned that the description language $\mathcal{ALC}$ with assertional axioms is a restriction of multi-modal logic enriched with nominals an @: nominals cannot be freely used in formulas and can only act as subindices of the @ operator. The logic $\mathcal{ALCO}$ [Sch94] moves closer to $\mathcal{H}(@)$ by allowing the

formation of concepts by means of sets of nominals. Eliminating the restrictions on @ from such a language in effect would give us an equational calculus for reasoning about individuals, and make it possible to specify additional frame properties.

3 Hybrid Tense Logic

The language of tense logic is a bimodal language; its $\Box$-modalities are written G and H and the respective $\Diamond$-modalities F and P. But these modalities are interrelated: while G and F look forward along the transition relation R, the H and P modalities look *backwards* along this relation (that is, H and P are interpreted using the converse of R). Now, we know from [Spa93b] that the K-satisfaction problem for tense logic is PSPACE-complete. However because G and H are interrelated the results of the previous section are not applicable. And in fact, adding even one nominal to tense logic causes a jump in complexity from PSPACE to EXPTIME, and we *don't* need to add @ to obtain this result. Our proof uses the *spy-point* technique from [BS95]; we will be exploring this technique in great detail in the following section when we discuss undecidable systems.

Theorem 2. *The* K*-satisfaction problem for a language of tense logic containing at least one nominal is* EXPTIME*-hard.*

Proof. We shall reduce the EXPTIME-complete *global* K-satisfaction problem for uni-modal languages to the (local) K-satisfaction problem for a basic tense language that contains at least one nominal. The global K-satisfaction problem for uni-modal languages is this: given a formula φ in the uni-modal language, does there exist a Kripke model $\mathfrak{M}$ such that $\mathfrak{M} \models \varphi$ (in other words, where φ is true in *all* states)? The EXPTIME-completeness of this problem is an easy consequence of (the proof of) the EXPTIME-completeness of modal logic K expanded with the universal modality in [Spa93a].

Define the following translation function $(\cdot)^t$ from ordinary uni-modal formulas to formulas in a tense language that contains at least one nominal i: $p^t = p$, $(\neg\varphi)^t = \neg\varphi^t$, $(\varphi \wedge \psi)^t = \varphi^t \wedge \psi^t$, $(\Diamond\varphi)^t = \mathsf{F}(\mathsf{P}i \wedge \varphi^t)$. Note that i is a *fixed* nominal in this translation. Clearly $(\cdot)^t$ is a linear reduction. We claim that for any formula φ, φ is globally K-satisfiable if and only if $i \wedge \mathsf{G}(\mathsf{P}i \rightarrow \varphi^t)$ is K-satisfiable.

For the left to right direction, let $\mathfrak{M} \models \varphi$, where $\mathfrak{M} = \langle M, R, V\rangle$ is a ordinary Kripke model. Define $\mathfrak{M}^*$ as follows: $M^* = M \cup \{i\}$, $R^* = R \cup \{(i, m) \mid m \in M\}$, $V^* = V \cup \{(n, \{i\}) \mid \text{ for all nominals } n\}$. $\mathfrak{M}^*$ is a hybrid model where all nominals (including i) are interpreted by the singleton set $\{i\}$, our spy-point. We claim that for all $m \in M$, for all ψ, we have $\mathfrak{M}, m \Vdash \psi$ if and only if $\mathfrak{M}^*, m \Vdash \psi^t$. This follows by a simple induction. The only interesting step is for $\Diamond$:

$$
\begin{aligned}
& \mathfrak{M}, m \Vdash \Diamond\psi \\
\iff\ & (\exists m' \in M) : Rmm' \ \&\ \mathfrak{M}, m' \Vdash \psi \\
\iff\ & (\exists m' \in M^*) : R^*mm' \ \&\ \mathfrak{M}^*, m' \Vdash \psi^t \ \&\ R^*im' \text{ (by IH and def. of } R^*) \\
\iff\ & \mathfrak{M}^*, m \Vdash \mathsf{F}(\mathsf{P}i \wedge \psi^t) \\
\iff\ & \mathfrak{M}^*, m \Vdash (\Diamond\psi)^t.
\end{aligned}
$$

It follows that $\mathfrak{M}^*, i \Vdash i \wedge \mathsf{G}(\mathsf{P}i \rightarrow \varphi^t)$, as desired.

For the other direction, let $\mathfrak{M}, w \Vdash i \wedge \mathsf{G}(\mathsf{P}i \rightarrow \varphi^t)$, where $\mathfrak{M} = \langle M, R, V \rangle$ is a hybrid model. Define $\mathfrak{M}^*$ as follows: $M^* = \{m \in M \mid Rwm\}$, $R^* = R_{\restriction M^*}$, $V^* = V_{\restriction M^*}$. We claim that for all $m \in M^*$, for all ψ, $\mathfrak{M}, m \Vdash \psi^t$ if and only if $\mathfrak{M}^*, m \Vdash \psi$. Again we only present the inductive step for $\Diamond$:

$$\begin{aligned}
& \mathfrak{M}, m \Vdash \mathsf{F}(\mathsf{P}i \wedge \psi^t) \\
\iff\ & (\exists m' \in M) : Rmm' \ \&\ Rwm' \ \&\ \mathfrak{M}, m' \Vdash \psi^t \\
\iff\ & (\exists m' \in M^*) : Rmm' \ \&\ Rwm' \ \&\ \mathfrak{M}, m' \Vdash \psi^t \\
\iff\ & (\exists m' \in M^*) : R^* mm' \ \&\ \mathfrak{M}^*, m' \Vdash \psi \text{ (by IH and definition of } M^*) \\
\iff\ & \mathfrak{M}^*, m \Vdash \Diamond\psi.
\end{aligned}$$

For all $m \in M^*$, Rwm holds, whence for all $m \in M^*$, $\mathfrak{M}, m \Vdash \mathsf{P}i$. So, since $\mathfrak{M}, w \Vdash \mathsf{G}(\mathsf{P}i \rightarrow \varphi^t)$, for all $m \in M^*$, $\mathfrak{M}, m \Vdash \varphi^t$. Hence by our last claim $\mathfrak{M}^* \models \varphi$, which is what we needed to show.

A matching upper bound can be obtained by interpreting the fragment in the guarded fragment with two variables [Grä97].

4 Binding nominals

Once we are used to treating labels as formulas, it is easy to obtain further expressivity. For example, instead of viewing nominals as *names*, we could think of them as *variables over states* and bind them with quantifiers. That is, we could form expressions like

$$\exists x.\Diamond(x \wedge \forall y.\Diamond(y \wedge \Diamond y \wedge p)).$$

This sentence is satisfied at a state w if and only if there is some state x accessible from w such that all states y accessible from x are reflexive and satisfy p. Historically, hybrid languages offering quantification over states were the first to be explored ([Bul70,PT85]). In their multi-modal version, they are essentially description languages which offer full first-order expressivity (see [BS98]). If the underlying modal language is taken to be the modal interval logic described in [Ben83a], the resulting system is essentially the full version of Allen's $\mathsf{Holds}(t, \varphi)$-based interval logic in which quantification over t is permitted (see [All84]). But because they offer full first-order expressivity over states, such hybrid languages are obviously undecidable.

More recently, there has been interest in hybrid languages which use a weaker binder called $\downarrow$ (see [Gor96,BS95]). Unlike $\exists$ and $\forall$, this is not a quantifier: it is simply a device which binds a nominal to the state where evaluation is being performed (that is, the *current state*). For example, the interplay between $\downarrow$ and @ allows us to define the *Until* operator:

$$\mathit{Until}(\varphi, \psi) \ := \downarrow x.\Diamond \downarrow y.@_x(\Diamond(y \wedge \varphi) \wedge \Box(\Diamond y \rightarrow \psi)).$$

This works as follows: we name the current state x, use $\Diamond$ to move to an accessible state, which we name y, and then use @ to jump us back to x. We then use $\Diamond$

to insist that φ holds at the state named y, while ψ holds at all successors of the current state that precede this y-labeled state.

$\mathcal{H}(\downarrow, @)$, the extension of $\mathcal{H}(@)$ with the $\downarrow$ binder, is proof theoretically well behaved, and completeness results for a wide class of frames can be obtained automatically (see [BT99,Bla98,Tza98]). But $\downarrow$ turns out to be extremely powerful: not only is $\mathcal{H}(\downarrow, @)$ undecidable, the sublanguage $\mathcal{H}(\downarrow)$ containing *only* the $\downarrow$ binder is too. However the only published undecidability result for $\mathcal{H}(\downarrow)$ is the one in [BS95], and this makes use of $\downarrow$ over a modal language with four modalities. In unpublished work, Valentin Goranko, and Blackburn and Seligman have proved undecidability in the uni-modal case, but these proofs make use of propositional variables to carry out the encoding. We are now going to prove the sharpest undecidability result yet for $\mathcal{H}(\downarrow)$ through a general expressivity result called the *Spypoint Theorem*. Roughly speaking, the Spypoint Theorem shows that $\downarrow$ is powerful enough to encode modal satisfaction over a wide range of Kripke models, and that it doesn't need the help of propositional variables or multiple modalities to do this.

4.1 The language $\mathcal{H}(\downarrow, @)$

Let's first make the syntax and semantics of $\mathcal{H}(\downarrow, @)$ precise.

Definition 3 (Syntax). As in Definition 1, PROP $= \{p, q, r, \ldots\}$ is a countable set of propositional variables, and NOM $= \{i, j, k, \ldots\}$ is a countable set of nominals. To this we add SVAR $= \{x_1, x_2, \ldots\}$ a countable set of *state variables*. We assume that PROP, NOM and SVAR are pairwise disjoint. We call SSYM $=$ NOM $\cup$ SVAR the set of *state symbols*, and ATOM $=$ PROP $\cup$ NOM $\cup$ SVAR the set of *atoms*. The *well-formed formulas* of $\mathcal{H}(\downarrow, @)$ (over ATOM) are

$$\varphi := a \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \Box\varphi \mid @_s\varphi \mid \downarrow v.\varphi$$

where $a \in$ ATOM, $v \in$ SVAR and $s \in$ SSYM.

The difference between nominals and state variables is simply this: nominals cannot be bound by $\downarrow$ whereas state variables can. The notions of *free* and *bound* state variable are defined as in first-order logic, with $\downarrow$ the only binding operator. A *sentence* is a formula containing no free state variables. A formula is *pure* if it contains no propositional variables, and *nominal-free* if it contains no nominals. In what follows we assume that some choice of PROP, NOM, and SVAR has been fixed.

Definition 4 (Semantics). Hybrid models $\mathfrak{M}$ are defined as in Definition 2. An *assignment* g for $\mathfrak{M}$ is a mapping $g :$ SVAR $\to M$. Given an assignment g, we define the assignment g^v_m by $g^v_m(v') = g(v')$ for $v' \neq v$ and $g^v_m(v) = m$. We say that g^v_m is *a v-variant of g*.

Let $\mathfrak{M} = \langle M, R, V\rangle$ be a model, $m \in M$, and g an assignment. For any atom a, let $[V, g](a) = \{g(a)\}$ if a is a state variable, and $V(a)$ otherwise. Then:

$\mathfrak{M}, g, m \Vdash a$ iff $m \in [V, g](a)$, $a \in$ ATOM
$\mathfrak{M}, g, m \Vdash \neg\varphi$ iff $\mathfrak{M}, g, m \nVdash \varphi$
$\mathfrak{M}, g, m \Vdash \varphi \wedge \psi$ iff $\mathfrak{M}, g, m \Vdash \varphi$ and $\mathfrak{M}, g, m \Vdash \psi$
$\mathfrak{M}, g, m \Vdash \Box\varphi$ iff $\forall m'(Rmm' \Rightarrow \mathfrak{M}, g, m' \Vdash \varphi)$
$\mathfrak{M}, g, m \Vdash \downarrow v.\varphi$ iff $\mathfrak{M}, g^v_m, m \Vdash \varphi$
$\mathfrak{M}, g, m \Vdash @_s\varphi$ iff $\mathfrak{M}, g, m' \Vdash \varphi$, where $[V, g](s) = \{m'\}$, $s \in$ SSYM.

We write $\mathfrak{M}, g \Vdash \varphi$ iff for all $m \in M$, $\mathfrak{M}, g, m \Vdash \varphi$, and $\mathfrak{M} \models \varphi$ iff for all g, $\mathfrak{M}, g \Vdash \varphi$.

Thus, as promised, $\downarrow$ enables us to bind a state variables to the current state. Note that, just as in first-order logic, if φ is a *sentence* it is irrelevant which assignment g is used to perform evaluation. Hence for sentences the relativization to assignments of the satisfaction relation can be dropped. A formula φ is *satisfiable* if there is a model $\mathfrak{M}$, an assignment g on $\mathfrak{M}$, and a state $m \in M$ such that $\mathfrak{M}, g, m \Vdash \varphi$.

We can now get down to business. First, we shall present a fragment of first-order logic (the *bounded fragment*) which is precisely as expressive as $\mathcal{H}(\downarrow, @)$ and provide explicit translations between these two languages. Secondly, we shall give an easy proof that (uni-modal) $\mathcal{H}(\downarrow, @)$ is undecidable. Third, we shall show how the dependency in this proof on @ and propositional variables can be systematically eliminated (in particular, we will show how to encode the valuation V so that the use of propositional variables can be simulated) and how we can encode any frame-condition expressible inside the pure fragment of $\mathcal{H}(\downarrow)$. This leads directly to the Spypoint Theorem and our undecidability result.

4.2 $\mathcal{H}(\downarrow, @)$ and the bounded fragment

We first relate $\mathcal{H}(\downarrow, @)$ to a certain bounded fragment of first-order logic. We shall work with a first-order language which contains a binary relation symbol R, a unary relation symbol P_j for each $p_j \in$ PROP, and whose constants are the elements of NOM. Obviously any hybrid model $\mathfrak{M} = \langle M, R, V\rangle$ can be regarded as a first-order model for this language: the domain of the model is M, the accessibility relation R is used to interpret the binary predicate R, unary predicates are interpreted by the subsets that V assigns to propositional variables, and constants are interpreted by the states that nominals name. Conversely, any model for our first-order language can be regarded as a hybrid model. So we shall let context determine whether we are referring to first-order or hybrid models, and continue to use the notation $\mathfrak{M} = \langle M, R, V\rangle$ for models.

First the easy part: we extend the well-known *standard translation ST* of modal correspondence theory (see [Ben83b]) to $\mathcal{H}(\downarrow, @)$. We assume that the first-order variables are SVAR$\cup\{x, y\}$ (where x and y are distinct new variables) and define the required translation by mutual recursion between two functions ST_x and ST_y. Here $\varphi[x/y]$ means "replace all free instances of x by y in φ."

$ST_x(p_j) = P_j(x)$, $p_j \in$ PROP.	$ST_y(p_j) = P_j(y)$, $p_j \in$ PROP.
$ST_x(i_j) = x = i_j$, $i_j \in$ NOM.	$ST_y(i_j) = y = i_j$, $i_j \in$ NOM.
$ST_x(x_j) = x = x_j$, $x_j \in$ SVAR.	$ST_y(x_j) = y = x_j$, $x_j \in$ SVAR.
$ST_x(\neg\varphi) = \neg ST_x(\varphi)$.	$ST_y(\neg\varphi) = \neg ST_y(\varphi)$.
$ST_x(\varphi \wedge \psi) = ST_x(\varphi) \wedge ST_x(\psi)$.	$ST_y(\varphi \wedge \psi) = ST_y(\varphi) \wedge ST_y(\psi)$.
$ST_x(\Diamond\varphi) = \exists y.(Rxy \wedge ST_y(\varphi))$.	$ST_y(\Diamond\varphi) = \exists x.(Ryx \wedge ST_x(\varphi))$.
$ST_x(\downarrow x_j.\varphi) = (ST_x(\varphi))[x_j/x]$.	$ST_y(\downarrow x_j.\varphi) = (ST_y(\varphi))[x_j/y]$.
$ST_x(@_s\varphi) = (ST_x(\varphi))[x/s]$.	$ST_y(@_s\varphi) = (ST_y(\varphi))[y/s]$.

Proposition 1. *Let φ be a hybrid formula, then for all hybrid models $\mathfrak{M}$, $m \in M$ and assignments g, $\mathfrak{M}, g, m \Vdash \varphi$ iff $\mathfrak{M} \Vdash ST_x(\varphi)[g^x_m]$.*

Proof. Induction on the structure of φ.

Now for the interesting question: what is the *range* of ST? In fact it belongs to a *bounded* fragment of our first-order language. This fragment consists of the formulas generated as follows:

$$\varphi := Rtt' \mid P_j t \mid t = t' \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \exists x_i.(Rtx_i \wedge \varphi) \text{ (for } x_i \neq t).$$

where x_i is a variable and t, t' are either variables or constants.

Clearly ST generates formulas in the bounded fragment. Crucially, however, we can also translate any formula in the bounded fragment into $\mathcal{H}(\downarrow, @)$ as follows:

$$\begin{array}{ll} HT(Rtt') & = @_t \Diamond t'. \\ HT(P_j t) & = @_t p_j. \\ HT(t = t') & = @_t t'. \\ HT(\neg\varphi) & = \neg HT(\varphi). \\ HT(\varphi \wedge \psi) & = HT(\varphi) \wedge HT(\psi). \\ HT(\exists v.(Rtv \wedge \varphi)) & = @_t \Diamond \downarrow v.HT(\varphi). \end{array}$$

By construction, $HT(\varphi)$ is a hybrid formula, but furthermore it is a boolean combination of @-formulas (formulas whose main operator is @). We can now prove the following strong truth preservation result.

Proposition 2. *Let φ be a bounded formula. Then for every first-order model $\mathfrak{M}$ and for every assignment g, $\mathfrak{M} \Vdash \varphi[g]$ iff $\mathfrak{M}, g \Vdash HT(\varphi)$.*

Proof. Induction on the structure of φ.

To summarize, there are effective translations between $\mathcal{H}(\downarrow, @)$ and the bounded fragment.

4.3 Undecidability of $\mathcal{H}(\downarrow, @)$

We are now ready to discuss undecidability. The result we want to prove is this:

> *The fragment of $\mathcal{H}(\downarrow)$ consisting of* pure nominal-free sentences *has an undecidable satisfaction problem.*

However we begin by quickly sketching an easy undecidability proof for the full language $\mathcal{H}(\downarrow, @)$. The proof uses the spypoint technique from the previous section together with results from [Spa93a]. By generalizing the methods used in this simple proof, we will be lead to the Spypoint Theorem and the undecidability result just stated.

Hemaspaandra shows in [Spa93a] that the global satisfaction problem of the uni-modal logic of the class K_{23} of frames is undecidable. K_{23} consists of all modal frames $\langle W, R\rangle$ in which every state has at most 2 R-successors and at most 3 two-step R-successors. We will show that we can reduce the satisfiability problem of this logic to $\mathcal{H}(\downarrow, @)$.

Let *Grid* be the conjunction of the following formulas:

G_1 $@_s\neg\Diamond s$
G_2 $@_s\Diamond\top$
G_3 $@_s(\Box\Box \downarrow x.@_s\Diamond x)$
G_4 $@_s(\Box \downarrow y.\Box \downarrow x_1.@_y\Box \downarrow x_2.@_y\Box \downarrow x_3.(@_{x_1}x_2 \vee @_{x_1}x_3 \vee @_{x_2}x_3))$
G_5 $@_s(\Box \downarrow y.\Box\Box \downarrow x_1.@_y\Box\Box \downarrow x_2.@_y\Box\Box \downarrow x_3.@_y\Box\Box \downarrow x_4.(\bigvee_{1\leq i\neq j\leq 4} @_{x_i}x_j))$.

What does *Grid* express? Suppose it is satisfied in a model $\mathfrak{M}$ on a frame (W, R). Then there exists a state which is named by s (the spypoint). By G_1, s is not related to itself. By G_2, s is related to some state, and by G_3, every state which can be reached from s in two steps can also be reached from s in one step. This means that in $\mathfrak{M}_s$ —the submodel of $\mathfrak{M}$ generated by s— every state is reachable from s in one step. Now G_4 and G_5 express precisely the two conditions characterizing the class K_{23} on successors of s. Instead of spelling out this proof we show that the similar formula $@_s\Box \downarrow y.\Box \downarrow x_1.@_y\Box \downarrow x_2.@_{x_1}x_2$ expresses that every successor of s in $\mathfrak{M}_s$ has at most one R-successor. As G_4 and G_5 follow the same pattern, it is easy to extend the argument below to verify their meaning.

$$
\begin{aligned}
& \mathfrak{M}, g, s \Vdash \Box \downarrow y.\Box \downarrow x_1.@_y\Box \downarrow x_2.@_{x_1}x_2 \\
\iff\ & (\forall w : sRw) : \mathfrak{M}, g^y_w, w \Vdash \Box \downarrow x_1.@_y\Box \downarrow x_2.@_{x_1}x_2 \\
\iff\ & (\forall u : wRu) : \mathfrak{M}, (g^y_w)^{x_1}_u, u \Vdash @_y\Box \downarrow x_2.@_{x_1}x_2 \\
\iff\ & \mathfrak{M}, (g^y_w)^{x_1}_u, w \Vdash \Box \downarrow x_2.@_{x_1}x_2 \\
\iff\ & (\forall v : wRv) : \mathfrak{M}, ((g^y_w)^{x_1}_u)^{x_2}_v, v \Vdash .@_{x_1}x_2 \\
\iff\ & (\forall w : sRw)(\forall u : wRu)(\forall v : wRv) : u = v.
\end{aligned}
$$

Now we are ready to complete the proof. We claim that for every formula φ,

φ is globally satisfiable on a K_{23}-frame iff *Grid* $\wedge\ @_s\Box\varphi$ is satisfiable.

The proof of the claim is a simple copy of the two constructions given in the proof of Theorem 2.

4.4 Undecidability of pure nominal-free sentences of $\mathcal{H}(\downarrow)$

We are ready to prove our main result. We do so by analysing the previous proofs and generalizing the underlying ideas. The models used in the proof of Theorem 2 and the undecidability proof just given both had a certain characteristic form. Let's pin this down:

Definition 5. A model $\mathfrak{M} = (W, R, V)$ is called a *spypoint model* if there is an element $s \in W$ (the spypoint) such that

i. $\neg sRs$;

ii. For all $w \in W$, if $w \neq s$, then sRw and wRs.

Notice that by ii above, any spypoint model is generated by its spy point. We will now show that with $\downarrow$ we can easily create spypoint models. On these models we can create for every variable x introduced by $\downarrow x$, a formula which has precisely the meaning of $@_x$.

Proposition 3. *Let* $\mathfrak{M} = \langle M, R, V\rangle$ *and* $s \in M$ *be such that* $\mathfrak{M}, s \Vdash \downarrow s.(\neg\Diamond s \wedge \Box\Box \downarrow x.\Diamond(s \wedge \Diamond x) \wedge \Box\Diamond s)$. *Then,*

i. $\mathfrak{M}_s$*, the submodel of* $\mathfrak{M}$ *generated by* s*, is a spypoint model with* s *the spypoint.*

ii. $@_s\varphi$ *is definable on* $\mathfrak{M}_s$ *by* $(s \wedge \varphi) \vee \Diamond(s \wedge \varphi)$.

iii. Let g *be any assignment. Then for all* $u \in M$, $\mathfrak{M}_s, g, u \Vdash @_x\varphi$ *iff* $\mathfrak{M}_s, g, u \Vdash @_s(\varphi \vee \Diamond(x \wedge \varphi))$.

Proof. *i* is immediate. *ii* and *iii* follow from the properties of a spypoint model.

Now, spypoint models are very powerful: we can encode lots of information about Kripke models (for finitely many propositional variables) inside a spypoint model. More precisely, for each Kripke model $\mathfrak{M}$, we define the notion of *a spypoint model of* $\mathfrak{M}$.

Definition 6. Let $\mathfrak{M} = \langle M, R, V\rangle$ be a Kripke model in which the domain of V is a finite set $\{p_1, \ldots, p_n\}$ of propositional variables. The spypoint model of $\mathfrak{M}$ (notation $\mathsf{Spy}[\mathfrak{M}]$) is the structure $\langle M', R', V'\rangle$ in which

i. $M' = M \cup \{s\} \cup \{w_{p_1}, \ldots, w_{p_n}\}$, for $s, w_{p_1}, \ldots, w_{p_n} \notin M$

ii. $R' = R \cup \{(s, x), (x, s) \mid x \in M'\backslash\{s\}\} \cup \{(x, w_{p_i}) \mid x \in M \text{ and } x \in V(p_i)\}$

iii. $V' = \emptyset$.

Let $\{s, x_{p_1}, \ldots, x_{p_n}\}$ be a set of state variables. A *spypoint assignment* for this set is an assignment g which sends s to the spypoint s and x_{p_i} to w_{p_i}. We use $\mathbf{m}$ as an abbreviation for $\neg s \wedge \neg x_{p_1} \wedge \ldots \wedge \neg x_{p_n}$. Note that when evaluated under the spypoint assignment, the denotation of $\mathbf{m}$ in $\mathsf{Spy}[\mathfrak{M}]$ is precisely M.

$\mathsf{Spy}[\mathfrak{M}]$ encodes the valuation on $\mathfrak{M}$ and we can take advantage of this fact. Define the following translation from uni-modal formulas to hybrid formulas:

$$\begin{array}{ll} IT(p_i) & = \Diamond(x_{p_i}) \\ IT(\neg\varphi) & = \neg IT(\varphi) \\ IT(\varphi \wedge \psi) & = IT(\varphi) \wedge IT(\psi) \\ IT(\Diamond\varphi) & = \Diamond(\mathbf{m} \wedge IT(\varphi)). \end{array}$$

Proposition 4. *Let* $\mathfrak{M}$ *be a Kripke model and* φ *a uni-modal formula. Then for any spypoint assignment* g*,*

$$\mathfrak{M} \models \varphi \text{ if and only if } \mathsf{Spy}[\mathfrak{M}], g, s \Vdash \Box(\mathbf{m} \rightarrow IT(\varphi)).$$

Proof. Immediate by the fact that the spypoint is R-related to all states in the domain of $\mathfrak{M}$, and the interpretation of $\mathbf{m}$ under any spypoint assignment g.

We modify the hybrid translation HT to its relativized version $HT^{\mathbf{m}}$ which also defines away occurrences of @. Define $HT^{\mathbf{m}}(\exists v.(Rtv \wedge \varphi))$ as $@_t\Diamond \downarrow v.(\mathbf{m} \wedge HT^{\mathbf{m}}\varphi)$ and replace all @ symbols by their definition as indicated in Propostion 3.*ii* and 3.*iii*.

The crucial step is now the fact that $\downarrow$ is strong enough to encode many frame-conditions.

Proposition 5. *Let $\mathfrak{M} = \langle M, R, V\rangle$ be a Kripke model. Let $C(y)$ be a formula in the bounded fragment in the signature $\{R, =\}$. Then for any assignment g,*

$$\langle M, R\rangle \models \forall y.C(y) \text{ if and only if } \mathsf{Spy}[\mathfrak{M}], g, s \Vdash \Box \downarrow y.(\mathbf{m} \rightarrow HT^{\mathbf{m}}(C(y))).$$

Proof. Immediate by the properties of HT, Proposition 3, and the fact that the spypoint is R-related to all states in the domain of $\mathfrak{M}$.

Theorem 3 (Spypoint theorem). *Let φ be a uni-modal formula in $\{p_1, \ldots, p_n\}$ and $\forall y.C(y)$ a first-order frame condition in $\{R, =\}$ with $C(y)$ in the bounded fragment. The following are equivalent.*

i. There exists a Kripke model $\mathfrak{M} = \langle M, R, V\rangle$ such that $\langle M, R\rangle \models \forall y.C(y)$ and $\mathfrak{M} \models \varphi$.

ii. The pure hybrid sentence F in the language $\mathcal{H}(\downarrow)$ is satisfiable. F is

$$\downarrow s.(SPY \wedge \Diamond \downarrow x_{p_1}.@_s\Diamond \downarrow x_{p_2}@_s \ldots \Diamond \downarrow x_{p_n}.@_s(DIS \wedge VAL \wedge FR)),$$

where

$$\begin{aligned} SPY &= \neg\Diamond s \wedge \Box\Box \downarrow x.\Diamond(s \wedge \Diamond x) \wedge \Box\Diamond s \\ DIS &= \Box(\textstyle\bigwedge_{1\leq i\leq n}(x_{p_i} \rightarrow \bigwedge\{\neg x_{p_j} \mid 1 \leq j \neq i \leq n\})) \\ VAL &= \Box(\mathbf{m} \rightarrow IT(\varphi)) \\ FR &= \Box \downarrow y.(\mathbf{m} \rightarrow HT^{\mathbf{m}}(C(y))). \end{aligned}$$

Proof. The way we have written it, F contains occurrences of $@_s$; but this does not matter, by Proposition 3 all these occurrences can be term-defined. So let's check that F works as claimed.

For the implication from *i* to *ii*, let $\mathfrak{M}$ be a Kripke model as in *i*. We claim that $\mathsf{Spy}[\mathfrak{M}], s \Vdash F$. The first conjunct of F is true in $\mathsf{Spy}[\mathfrak{M}]$ at s by Proposition 3. The diamond part of the second disjunct can be satisfied using any spypoint assignment g. In the spypoint model all w_{p_i} are pairwise disjoint, whence $\mathsf{Spy}[\mathfrak{M}], g, s \Vdash DIS$. By Propositions 4 and 5, also $\mathsf{Spy}[\mathfrak{M}], g, s \Vdash VAL \wedge FR$.

For the other direction, let $\mathfrak{M}, s \Vdash F$. By Proposition 3, the submodel $\mathfrak{M}_s = \langle M_s, R_s, V_s\rangle$ generated by s is a spypoint model. Let g be the assignment such that $\mathfrak{M}, g, s \Vdash DIS \wedge VAL \wedge FR$. By DIS, $g(x_{p_i}) \neq g(x_{p_j})$ for all $i \neq j$, and (since $\neg sRs$) also $g(x_{p_i}) \neq s$, for all i. Define the following Kripke model $\mathfrak{M}' = \langle M', R', V'\rangle$, where

$$\begin{aligned} M' &= M \setminus \{g(s), g(x_{p_1}), \ldots, g(x_{p_n})\} \\ R' &= R \restriction_{M'} \\ V'(p_i) &= \{w \mid wRg(x_{p_i})\}. \end{aligned}$$

Note that $\mathsf{Spy}[\mathfrak{M}']$ is precisely $\mathfrak{M}_s$, and g is a spypoint assignment. But then by Propositions 4 and 5 and the fact that $\mathfrak{M}_s, g, s \Vdash \mathit{VAL} \wedge \mathit{FR}$, we obtain $\mathfrak{M}' \models \varphi$ and $\langle M', R' \rangle \models \forall y.C(y)$.

The proof of the claimed undecidability result is now straightforward.

Corollary 1. *The fragment of $\mathcal{H}(\downarrow)$ consisting of all pure nominal-free sentences has an undecidable satisfaction problem.*

Proof. We will reduce the undecidable global satisfaction problem in the uni-modal language over the class K_{23}, just as we did in our easy undecidability result for $\mathcal{H}(\downarrow, @)$. The first-order frame conditions defining K_{23} are of the form $\forall y.C(y)$ with $C(y)$ in the bounded fragment. (This is easy to check. For instance, y has at most two successors can be written as $\forall x_1.(yRx_1 \rightarrow \forall x_2.(\rightarrow \forall x_3. \rightarrow (x_1 = x_2 \vee x_1 = x_3 \vee x_2 = x_3)))$.) Now apply the Spypoint Theorem. The formula F (after all occurrences of $@_s$ have been term-defined) is a pure nominal-free sentence of $\mathcal{H}(\downarrow)$, and the result follows.

Because of the generality of the Spypoint Theorem, it seems unlikely that even restricted forms of label binding will lead to decidable systems. For this reason, much of our ongoing research is focusing on binder free systems, such as *Until*-based languages enriched with nominals and @, and modal languages with counting modalities (these are widely used in description logic) enriched in the same way.

5 Concluding remarks

In this paper we have examined the complexity of a number of hybrid languages. Our results have been both positive and negative and we sum them up here:

1. Adding nominals and @ to the uni-modal language, or even the multi-modal language, does not lead to an increase in complexity: K-satisfiability remains PSPACE-complete.
2. On the other hand, adding even one nominal to the language of tense logic takes the complexity from PSPACE-complete to EXPTIME-complete.
3. We provide a simple proof of the known fact that $\mathcal{H}(\downarrow, @)$ is undecidable. Furthermore, we prove that very restricted use of $\downarrow$ leads already to undecidability. In fact, undecidability strikes even in the sentential fragment of the uni-modal language without @ or propositional variables.

Furthermore, a simple extension of the undecidability proof provided in this paper shows that this last fragment is even a conservative reduction class in the sense of [BGG97].

Needless to say, the results we presented conform just a preliminary sketch of the complexity-theoretic territory occupied by hybrid languages. The spectrum of plausible directions for further work is huge. As an example, we have only considered logics with full Boolean expressive power. In the description logic community fragments which restrict negation or dissallow disjuctions (aiming to obtain good computational behavior) are standard. Again, the generality of the Spypoint Theorem will be of much help in mapping this new variations.

References

[AdR99] C. Areces and M. de Rijke. Accounting for assertional information. Manuscript, 1999.

[All84] J. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, July 1984.

[Ben83a] J. van Benthem. *The logic of time*. Reidel, 1983.

[Ben83b] J. van Benthem. *Modal Logic and Classical Logic*. Bibliopolis, Naples, 1983.

[BGG97] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer Verlag, 1997.

[Bla98] P. Blackburn. Internalizing labeled deduction. Technical Report CLAUS-Report 102, Computerlinguistik, Universität des Saarlandes, 1998. `http://www.coli.uni-sb.de/cl/claus`.

[BS95] P. Blackburn and J. Seligman. Hybrid languages. *J. of Logic, Language and Information*, 4(3):251–272, 1995. Special issue on decompositions of first-order logic.

[BS98] P. Blackburn and J. Seligman. What are hybrid languages? In M. Kracht, M. de Rijke, H. Wansing, and M. Zakharyaschev, editors, *Advances in Modal Logic*, volume 1, pages 41–62. CSLI Publications, Stanford University, 1998.

[BT99] P. Blackburn and M. Tzakova. Hybrid languages and temporal logics. *Logic J. of the IGPL*, 7(1):27–54, 1999.

[Bul70] R. Bull. An approach to tense logic. *Theoria*, 36:282–300, 1970.

[Chl86] B. Chlebus. Domino-tiling games. *J. Comput. System Sci.*, 32(3):374–392, 1986.

[Gab96] D. Gabbay. *Labeled deductive systems*. Oxford University Press, 1996.

[Gor96] V. Goranko. Hierarchies of modal and temporal logics with reference pointers. *J. Logic Lang. Inform.*, 5(1):1–24, 1996.

[Grä97] E. Grädel. On the restraining power of guards. To appear in J. of Symbolic Logic, 1997.

[HM92] J. Halpern and Y. Moses. A guide to completeness and complexity for modal logics of knowledge and belief. *Artificial Intelligence*, 54:319–379, 1992.

[Lad77] R. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM J. of Computing*, 6(3):467–480, 1977.

[PT85] S. Passy and T. Tinchev. Quantifiers in combinatory PDL: completeness, definability, incompleteness. In *Fundamentals of computation theory (Cottbus, 1985)*, pages 512–519. Springer, Berlin, 1985.

[Sch91] K. Schild. A correspondence theory for terminological logics. In *Proc. of the 12th IJCAI*, pages 466–471, 1991.

[Sch94] A. Schaerf. *Query Answering in Concept-Based Knowledge Representation Systems: Algorithms, Complexity, and Semantic Issues*. PhD thesis, Dipartimento di Informatica e Sistemistica, Univ. di Roma "La Sapienza", 1994.

[Sel97] J. Seligman. The logic of correct description. In M. de Rijke, editor, *Advances in Intensional Logic*, pages 107–135. Kluwer, 1997.

[Spa93a] E. Spaan. *Complexity of modal logics*. PhD thesis, ILLC. University of Amsterdam, 1993.

[Spa93b] E. Spaan. The complexity of propositional tense logics. In *Diamonds and defaults (Amsterdam, 1990/1991)*, pages 287–307. Kluwer Acad. Publ., Dordrecht, 1993.

[SSS91] M. Schmidt-Schauss and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48:1–26, 1991.

[Tza98] M. Tzakova. Tableaux calculi for hybrid logics. Manuscript, 1998.

MonadicNLIN and Quantifier-Free Reductions

Clemens Lautemann, Bernhard Weinzinger
Institut für Informatik, Johannes Gutenberg–Universität, D–55099 Mainz
email: {cl,bw}@informatik.uni-mainz.de

Abstract. We define a monadic logic Monadic**NLIN** which is a fragment of Grandjean's logic for the class **NLIN** of problems solvable in linear time on nondeterministic random–access machines. This logic operates on *functional* rather than the usual relational structures of finite model theory, and we adapt the notions of quantifier–free interpretation and reduction to this functional setting. We also introduce the notion of *compatible* successor function, which, in our setting, replaces the built–in linear order relations used in logical characterisations of complexity classes. We show that Monadic**NLIN** is closed under quantifier–free functional reductions, that CNF–SAT is complete for Monadic**NLIN** under these reductions, and that Monadic**NLIN** contains a large number of **NP**–complete problems, but not the set of connected graphs.

Keywords: descriptional complexity, computational complexity, functional structures, linear time

1 Introduction

Following Fagin's seminal paper [Fag74], logical characterisations of complexity classes have become a very active research area, and today, for most major classes of computational complexity such characterisations have been found (cf. [Imm99]). Apart from providing a static, syntactic description of a dynamic, semantic notion, one main aspect of most of these characterisations is that they are in terms of *relational structures*, without any explicit reference to string encodings. For instance, by Fagin's theorem, the class of graph problems in **NP** is the class of those sets of graphs that can be defined in existential second–order logic, Σ^1_1, *on graphs*, i.e., over a signature containing one single binary predicate symbol (for the edge relation). This is possible where complexity classes are insensitive to the precise nature of input encodings; for most classes, this can vary in length within a polynomial without affecting the class. The situation is quite different if we consider more fine–grained complexity classes, e.g., linear time. Although frequently used in algorithm design, this notion is far from having a generally accepted, precise definition. It seems to depend too strongly on details of both, the computational model, and the input representation. For instance, in the case of graph algorithms, linear usually means linear in $|V|+|E|$. This calls for a different high–level representation of graphs, with a universe which consists of both, vertices and edges. Although such representations had been used

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 322−337, 1999.

before, with the graph structure given by the incidence relation (cf. [Cou94]), one further idea was needed in order to model linear time on graphs: in [GO94,GO96], Grandjean and Olive gave a logical characterisation of the class **NLIN** (linear time on nondeterministic random–access machines) on *functional* structures. In this setting, a graph is represented by two functions, *head* and *tail*, which map edges to their endpoints. This representation corresponds rather closely to the adjacency lists used in algorithm design, and Grandjean and Olive showed, building on an earlier logical characterisation over strings, given by Grandjean ([Gra94b,Gra94a,Gra96]), that **NLIN** contains precisely those sets of functional structures which can be defined by formulae of the form $\exists \bar{f} \forall x \varphi$, where $\bar{f}$ is a list of unary function symbols, and φ is a quantifier–free first–order formula.

One central motivation for studying logical characterisations of complexity classes is the hope that it might be possible to bring tools from logic to bear on complexity theoretic problems. In an attempt to shed light on the **NP** vs. **coNP** problem, Fagin investigated the class Monadic**NP** of those problems which are defined by sentences in the logic MonadicΣ^1_1, the restriction of Σ^1_1 in which second–order quantifiers range only over sets, instead of arbitrary relations ([Fag75]). This class still contains **NP**–complete problems, but Fagin showed that it does not contain the set of connected graphs (hence is not closed under complement).

In this paper we pursue a similar programme for **NLIN**. By restricting the scope of the second–order quantifiers in the above formula to sets rather than functions, we define the class Monadic**NLIN**, a subclass of **NLIN** which possesses a number of interesting properties: it contains a large number of **NP**–complete problems and consists of precisely those problems which are reducible to CNF–SAT by quantifier–free reductions. On the other hand there are computationally simple problems not contained in it, notably, as in the case of monadic**NP**, the set of connected graphs.

In the course of our investigations we have to adapt notions and methods from the realm of relational to that of functional structures. This concerns the notions of (quantifier–free) *interpretation* and *reduction*, which, mainly due to the possibility of nested terms, become more complicated in our setting.
More importantly, however, we also have to deal with the problem of input representation, which cannot be ignored in the context of linear time. In order to model the sequentiality of computation, logics which are intended to characterise complexity classes, have to make use of some kind of order or successor on the input data. If the logic is expressive enough, as in the case of Σ^1_1, or Grandjean's logic, such an order can be quantified. In the case of weaker logics, however, one has to assume the structures to be given with a built–in order. Since we want to express properties of structures which are independent of the particular choice of an order, in this situation, we usually restrict the logic to order–independent formulae: such a formula holds either for all orderings of the structure, or for none (cf., e.g., [Imm99]). In our context order–independence is quite a delicate notion, since subclasses of linear time are not necessarily entirely independent of the input order. When designing an algorithm, we usually assume the input graph to be given in some systematic way, e.g., in form of an adjacency list.

However, the order in which vertices appear in this input should be irrelevant, as long as this systematic form is retained – in the case of adjacency lists this means that edges adjacent to one vertex appear in direct succession. We model this restriction on the input order with the notion of a *compatible* successor function and require the validity of formulae to be invariant under different choices of compatible successors.

2 MonadicNLIN

In this section we first specify the signatures σ of our logic. Then, we describe extensions σ' of these signatures with additional symbols interpreted "compatibly" with the original ones and specify the syntax of a Monadic**NLIN** formula.

Definition 1 (unary functional structure).

- *A* unary functional signature *a finite set of function symbols of arity*[1] ≤ 1.
- *A* unary functional structure *is a finite structure over a unary functional signature.*

In the following "functional signature (structure)" will always mean "unary functional signature (structure)". We presume that in every functional signature there is at least one constant symbol, *nil*, and one unary function symbol. If there is more than one constant we will use the symbols $0, 1, \ldots, n$, with 0 as a synonym for *nil*.

Examples 1
1. We view a directed graph as a functional structure whose universe can be partitioned into vertices, edges and an element nil, with functions head and tail mapping edges to vertices: tail maps an edge to its starting vertex and head maps it to its target vertex. Vertices are characterised by the fact that they are mapped by head and tail to the constant nil, which we assume to be a fixpoint of both head and tail. Hence, a directed graph is a σ_G*–structure* $\mathcal{G}$ *– with* $\sigma_G := \{head, tail, nil\}$ *consisting of unary function symbols head and tail and constant symbol nil – in which the universe is partitioned by the predicates* $NIL(x) :\leftrightarrow head(x)=tail(x)=x=nil$, $E(x) :\leftrightarrow V(head(x)) \wedge V(tail(x))$, *and* $V(x) :\leftrightarrow \neg NIL(x) \wedge NIL(head(x)) \wedge NIL(tail(x))$.

2. We view a Boolean formula in CNF as a functional structure whose universe can be partitioned into clauses, occurrences, variables and elements 0 *and* 1 *and where a function junction maps clauses to* 0, *variables to* 1 *and occurences to clauses, a function var maps occurrences to variables or to* 0 *(FALSE) or* 1 *(TRUE) and a function neg maps occurrences to* $\{0, 1\}$ *(positive or negative occurrence). Thus, a Boolean formula in CNF is a* σ_{CNF}*–structure* $\mathcal{B}$, *with* $\sigma_{CNF} := \{junction, var, neg, 0, 1\}$, *where the partition of the universe into zero, one, variables, occurrences, and clauses can easily be expressed in first order by predicates* $ZERO$, ONE, V, C, O.

[1] By "function symbol of arity 0" we mean a constant symbol.

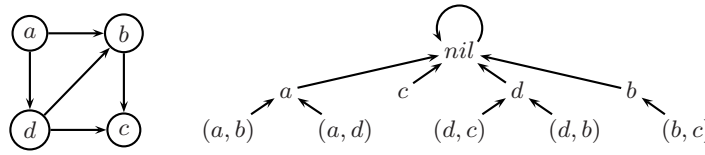


Fig. 1. A graph, and its function graph of *tail*.

$$(\neg x_1 \vee x_2) \wedge (\neg x_3 \vee 1) \wedge (x_1 \vee \neg x_2 \vee x_3)$$

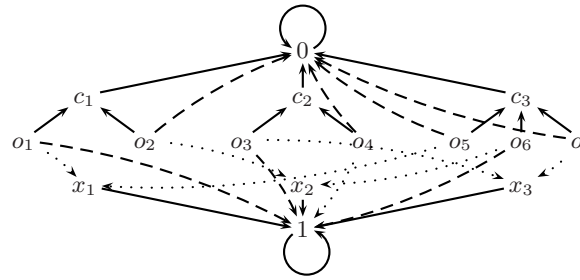


Fig. 2. A CNF–formula and its function graph. The function *junction* is represented by solid, *var* by dotted, and *neg* by dashed lines. Function values not shown are all 0.

3. Boolean formulae in k–CNF, i.e., with precisely k literals per clause, can be represented more easily: Here the universe consists of $0, 1$*, a set* C *of clauses and a set* V *of variables, and we have* k *functions* $f_1 \dots, f_k : C \to V$ *to indicate the variable for each literal, and* k *functions* $neg_1, \dots, neg_k : C \to \{0, 1\}$ *to indicate which variables occur negated. Thus these formulas are* σ_{kCNF}*–structures, with signature* $\sigma_{kCNF} := \{f_1, \dots, f_k, neg_1, \dots, neg_k, 0, 1\}$.

We want to define sets of functional structures by syntactic restrictions on Grandjean's logic – which already is quite restrictive. In order to obtain something computationally meaningful, we have to provide some means of expressing a systematic exploration of the structure. In other contexts, this is usually done by a built–in linear order relation; in our functional setting, we use successor functions instead. However, not every successor will be equally useful: in order to explain our choice of successor, let us look at the example of graphs. Algorithms on graphs often assume the input to be given in the form of an *adjacency list*, i.e. an array (or a linked list) of pairs $\langle v, l(v)\rangle$, where each $v \in V$ appears precisely once, and $l(v)$ is a linked list of all the edges leaving v. Such an adjacency list induces the successor function which starts with the first vertex in the main list, after each vertex v enumerates all of $l(v)$ and at the end of $l(v)$ continues with the vertex which succeeds v in the main list. If we transfer this successor function onto our functional representation of the graph we see that it corresponds to the preorder generated by a depth-first traversal of the rooted

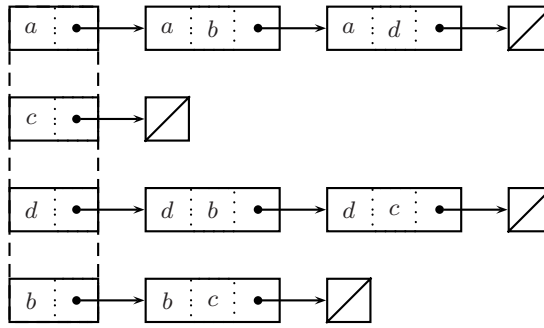


Fig. 3. An adjacency list representation of the graph of Figure 1

tree given by the function *tail*: such a traversal starts at *nil*, next it will visit a vertex, after each vertex v all edges e with $tail(e) = v$ are visited before the traversal moves on to some other vertex, and so on. Over graphs, the compatible successor functions on $V \cup E$ will be just those derived in this way from an adjacency list representation of G.

Now the correctness of an algorithm on adjacency lists should not depend on the order in which the vertices appear in the main list, nor on the order of any of the edge lists. Correspondingly, when describing a graph property by a logical formula ϕ containing a successor function symbol s we want

- s to be interpreted by a successor function which corresponds to an adjacency list representation
- ϕ's validity on a given graph G to be independent of *which* adjacency list representation of G is modeled by s.

When dealing with general functional structures, there is no distinguished function - even on graphs we might as well use *head* instead of *tail*, which would give us a successor corresponding to the dual adjacency list with lists of incoming rather than outgoing edges. A further complication arises from the fact that, in general, function graphs are not trees but unions of components each of which consists of a set of trees whose roots are connected in a cycle.

The following definitions take this into account.

Definition 2 (compatible successors).
Let U be a finite set and let $f : U \to U$ be a function.

- *Let f define a connected graph $F = \{(u, f(u)) \mid u \in U\}$. Since there is exactly one cycle C in F, deleting one pair $(u', f(u'))$ in C leaves a tree, which we denote by $F_{u'}$.*
  A successor function s_f^{pre} on U is pre–compatible *with f if there is a u' such that s_f^{pre} is obtained by a preorder traversal of $F_{u'}$, i.e., a depth first traversal of $F_{u'}$ which lists the elements of U the first time they are encountered during the traversal.*

- *For general f a successor function s_f^{pre} on U is* pre–compatible *with f, if it is pre–compatible with f on every connected component of $F = \{(u, f(u)) \mid u \in U\}$.*
- *Analogously, a successor function s_f^{post} is* post–compatible *with f if, for some u', it is obtained by a depth first traversal of $F_{u'}$ which lists the elements of U the last time they are encountered during the traversal.*

Examples 2
1. A successor function on the graph of Figure 1 precompatible with tail is drawn in Figure 4.

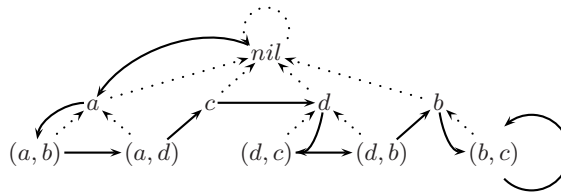


Fig. 4. The successor function is shown by solid arcs.

2. Let $\mathcal{B}$ be a $\{junction, var, neg, 0, 1\}$–structure representing a Boolean formula in CNF. A successor function s pre–compatible with junction starts with one of the elements 0 and 1. If it starts with 1, this element is followed by a list of the variables followed by the element 0; then the clauses follow but with a list of all its occurrences inserted immediately after each clause. If s starts with 0 then the parts (1, variables)(0, clause, occurrences, clause ...) are interchanged.

Definition 3 (Monadic**NLIN**). *Let σ be a unary functional signature.*
1. Set $\sigma^{succ} :=$
$\sigma \cup \{s_f^{pre}, s_f^{post}, min_f^{pre}, max_f^{pre}, min_f^{post}, max_f^{post} \mid f \in \sigma,\ f$ *has arity* $1\}$ *where the $s_f^{pre}, s_f^{post}, min_f^{pre}, max_f^{pre}, min_f^{post}, max_f^{post}$ are function symbols of arities 1,1,0,0,0 and 0, respectively. Let $\sigma \subseteq \sigma' \subseteq \sigma^{succ}$ and set $\sigma^s := \sigma' \setminus \sigma$. We say that a σ'-structure $\mathcal{A}'$ is a* compatible σ'-extension *of a σ-structure $\mathcal{A}$ if*

- *$\mathcal{U}^{\mathcal{A}'} = \mathcal{U}^{\mathcal{A}}$ and $f^{\mathcal{A}'} = f^{\mathcal{A}}$ for $f \in \sigma$ and*
- *the symbols s_f^{pre} resp. s_f^{post} in σ' are interpreted in $\mathcal{A}'$ as successor functions pre–compatible resp. post–compatible with f and the constant symbols $min_f^{pre}, max_f^{pre}, min_f^{post}, max_f^{post}$ as the respective minima and maxima.*

Let $\mathcal{B}$ be a functional structure over the signature σ representing a relational structure $\mathcal{A}$ as in Example 1.1. We will call (cf. Section 4) any compatible σ'–extension $\mathcal{B}'$ of $\mathcal{B}$ an admissible representation *of $\mathcal{A}$.*
2. A σ–property $\mathcal{P}$ (i.e. a class $\mathcal{P}$ of σ–structures which is closed under σ–isomorphisms) is a Monadic**NLIN**$_{\sigma,\sigma'}$*–property, if $\sigma \subseteq \sigma' \subseteq \sigma^{succ}$ and there is a second–order formula $\Phi[\sigma']$ such that*

- *Φ is of the form $\exists \bar{X} \forall x\, \phi$ where $\bar{X}$ is a tupel of* unary *second–order variables and ϕ is a quantifier–free σ'–formula in the X_i and in* one *first–order variable x and*
- *for any σ–structure $\mathcal{A}$ and any compatible σ'–extension $\mathcal{A}'$ of $\mathcal{A}$ we have*

$$\mathcal{A} \in \mathcal{P} \quad \Longleftrightarrow \quad \mathcal{A}' \models \Phi.$$

3. We write
- *$\mathcal{P} \in$ Monadic$\mathbf{NLIN}_{\sigma,\sigma'}$ if $\mathcal{P}$ is a* Monadic$\mathbf{NLIN}_{\sigma,\sigma'}$*–property,*
- *$\mathcal{P} \in$ Monadic$\mathbf{NLIN}_{\sigma}$ if $\mathcal{P}$ is a* Monadic$\mathbf{NLIN}_{\sigma,\sigma'}$*–property for some σ' and*
- *$\mathcal{P} \in$ Monadic$\mathbf{NLIN}$ if $\mathcal{P} \in$ Monadic$\mathbf{NLIN}_{\sigma}$ for some σ.*

The name Monadic**NLIN** and the syntax of our formulae are motivated by the logic $\exists \bar{f} \forall x\, \phi$, with unary function variables $\bar{f}$, which on functional structures characterises **NLIN**, i.e. non-deterministic linear time on Grandjean's RAM–model, [GO96]. The restriction to *linear* time is reflected both by the restricted first-order syntax and the use of *unary functional* structures. We give some evidence that our restriction to monadic second-order quantification still defines an expressive class.

Proposition 1.
The following problems are in Monadic**NLIN***: 3COL, KERNEL, CNF–SAT, 2PPM*[2]*.*

Proof: To illustrate the use of the successors, we sketch the proof for CNF–SAT: Let $\mathcal{F}$ be a Boolean formula in CNF. Denote an assignment of truth values by a partition of the variables into sets P (truth value 1) and $\neg P$ (truth value 0). Given a successor function on the occurrences, $\mathcal{F} \in CNF - SAT$ iff there is P and a predicate $R(x)$ ("Red") on the occurrences such that
1. in each clause, the first occurrence is red if and only if it is satisfying;
2. the successor of an occurence o is red if and only if it is satisfying or o is red;
3. the last occurrence in each clause is red.

We can express that x is a satisfying occurrence wrt P by a formula $sat_P(x)$. Hence, with a *junction*-precompatible successor s we can express 1.–3.:
1. by $\phi_1(x) = [C(x) \wedge O(s(x))] \rightarrow [R(s(x)) \leftrightarrow sat_P(s(x))]$,
2. by $\phi_2(x) = O(x) \rightarrow [R(s(x)) \leftrightarrow (sat_P(s(x)) \vee R(x))]$,
3. by $\phi_3(x) = [(O(x) \wedge C(s(x))) \vee x = max] \rightarrow R(x)$. □

3 Quantifier–free reductions and the completeness of CNF–SAT

In this section we define a way to interpret one functional signature τ in terms of another functional signature σ in analogy to the relational interpretations of [Cos93]. We derive natural notions of reduction and of Monadic**NLIN**–completeness,

[2] 2–partition into perfect matchings, cf. [Cre95]

and show that Monadic**NLIN** is closed under these reductions and that CNF–SAT is Monadic**NLIN**–complete.

We sketch the idea behind our notion of interpretation first: Basically an interpretation I associates to every function symbol g in τ a σ–term t. Evaluated in a given σ–structure $\mathcal{A}$ with universe A, $t(a)$ defines the value of $g(a)$ for any $a \in A$.[3] Thus an interpretation I of τ in σ associates to every σ–structure a τ–structure $I(\mathcal{A})$. We use two generalisations: 1. The values of g in $I(\mathcal{A})$ will be defined by a case–distinction expressed by σ–formulae ϕ^r: If for $a \in A$ we have $\mathcal{A} \models \phi^r(a)$ then we will evaluate a corresponding σ–term $t^r(a)$ to obtain $g(a)$.[4] 2. We will allow interpretations for which the universe of $I(\mathcal{A})$ consists of c copies of the universe of $\mathcal{A}$, i.e. consists of elements (a, i), $k = 1, ..., c$ where the constant c is given by the interpretation[5]. In such a case we have to define g on each of the c "levels" of the universe of $I(\mathcal{A})$ and want to be able to express that (a, k) is mapped to (a', l) on a possibly different level l. Such a definition is formalized by σ–formulae ϕ_{kl} and σ–terms t_{kl} describing that $g(a, k) = (t_{kl}(a), l)$ whenever $\mathcal{A} \models \phi_{kl}(a)$.[6]

Definition 4 (qffi). *Let σ and τ be functional signatures. A* 1–dimensional quantifier–free functional interpretation of τ in σ *or* qffi of τ in σ*, for short, is a tupel $I = (c, \bar{c}, \bar{\phi}, \bar{t})$ where*

- *c is a natural number, the* length *of I*
- *$\bar{c}$ is a $|\tau|c^2$–tupel of natural numbers c_{gkl} ($g \in \tau$, $k, l = 1, \ldots, c$),*
- *$\bar{\phi}$ is a tuple of quantifier–free σ–formulae ϕ^r_{gkl} ($g \in \tau$; $k, l = 1, \ldots, c$; $r = 1, \ldots, c_{gkl}$)*
  - *if g has arity 0, i.e. if g is a constant, then the ϕ^r_{gkl} are variable-free formulae and for any σ–structure $\mathcal{A}$ exactly one of the ϕ^r_{gkk}, $k = 1, \ldots, c$, $r = 1, \ldots, c_{gkk}$ is true, and*
  - *if g has arity 1, then the ϕ^r_{gkl} have one variable and for fixed k and for any σ–structure $\mathcal{A}$, the ϕ^r_{gkl}, $l = 1, \ldots, c$, $r = 1, \ldots, c_{gkl}$ define a partition on the universe of $\mathcal{A}$, i.e. exactly one ϕ^r_{gkl} holds for any one element in the universe of $\mathcal{A}$*[7]*,*
- *$\bar{t}$ is a tupel of σ–terms t^r_{gkl} ($g \in \tau$, $k, l = 1, \ldots, c$; $r = 1, \ldots, c_{gkl}$).*

A 1–dimensional quantifier–free functional interpretation defines for every σ–structure $\mathcal{A}$ a τ–structure $\mathcal{B} := I(\mathcal{A})$ where $U^{\mathcal{B}} := \{(a, i) \mid a \in U^{\mathcal{A}}, 1 \leq i \leq c\}$, a constant symbol $g \in \tau$ is interpreted by $g_{\mathcal{B}} = (a', k)$ iff ($\mathcal{A} \models \phi^r_{gkk}$ and $\mathcal{A} \models a' = t^r_{gkk}(a)$) and a function symbol g by $g_{\mathcal{B}}((a, k)) = (a', l)$ iff ($\mathcal{A} \models \phi^r_{gkl}(a)$ and $\mathcal{A} \models a' = t^r_{gkl}(a)$).

---

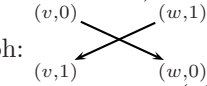
[3] If g has arity 0, t is a closed term.

[4] We will stipulate, of course, that the ϕ^r define a partition on the universe of A, i.e. that every $a \in \mathcal{A}$ satisfies precisely one of the ϕ^r.

[5] Cosmadakis introduced this generalisation in the relational setting when studying Monadic **NP**, [Cos93].

[6] Again, we have to stipulate that for a given k the $\phi^r_{kk'}$ define a partition on the universe of A.

[7] As for the constants, this condition is not meant to contradict the *syntactical* character of the definition: think of the ϕ_i as of a sequence $\phi_i \wedge \bigwedge_{i'<i} \neg\phi_{i'}$ with a final catch–all formula $\bigvee_{i<i_{max}} \neg\phi_i$.

Example 1. Consider the well–known reduction from 2–SAT to graphs which do not contain certain cycles. Given a formula in 2–CNF, a graph is constructed as follows: For every variable v, there are two vertices, $(v,1),(v,0)$, (representing v and $\neg v$, respectively), and for every clause c which contains variables v,w, there are two edges $(c,0),(c,1)$: if both variables occur unnegated in c, then $(c,0)=\langle(v,0),(w,1)\rangle$, $(c,1)=\langle(w,0),(v,1)\rangle$, if v occurs negated, w unnegated, then $(c,0)=\langle(v,1),(w,1)\rangle$, $(c,1)=\langle(w,0),(v,0)\rangle$, etc. (just view these edges as the representation of the two implications equivalent to c). The clause $c=(v\vee\neg w)$, for instance, is modeled by the subgraph: 

This construction is easily modeled as a qffi of length 2 of σ_G in σ_{2CNF}, (cf. Example 1). We let $\phi_{nil,0,0}(x)=(x{=}0)$, and define the functions *head* and *tail* in such a way that, in accordance with the above construction, if both variables in c are unnegated then $tail(c,0)=(f_1(c),0)$, $head(c,0)=(f_2(c),1)$, etc. More precisely, our formulae and terms for *tail* are as follows: (the ones for *head* are formed analogously):
$\phi^1_{tail,0,0}=C(x)\wedge neg_1(x){=}0$, $t^1_{tail,0,0}(x)=f_1(x)$, $\phi^2_{tail,0,0}=\neg C(x)$, $t^2_{tail,0,0}(x)=0$, $\phi^1_{tail,0,1}=C(x)\wedge neg_1(x){=}1$, $t^1_{tail,0,1}(x)=f_1(x)$, $\phi^1_{tail,1,0}=C(x)\wedge neg_2(x){=}0$, $t^1_{tail,0,1}(x)=f_2(x)$, $\phi^2_{tail,1,0}=\neg C(x)$, $t^1_{tail,1,0}(x)=0$, $\phi^1_{tail,1,1}=C(x)\wedge neg_2(x){=}1$, $t^1_{tail,1,1}(x)=f_2(x)$.

Definition 5 (qffr(σ',τ')).
1. Let $\mathcal{P}$ be a property of σ–structures, $\mathcal{Q}$ a property of τ–structures and I a qffi of τ' in σ'.
I is a 1–dimensional quantifier–free functional (σ',τ')–reduction compatible with successors (qffr(σ',τ')) from $\mathcal{P}$ to $\mathcal{Q}$ *if for every σ–structure $\mathcal{A}$ and each compatible σ'–extension $\mathcal{A}'$*

- *the τ'–structure $\mathcal{B}':=I(\mathcal{A}')$ is a compatible τ'–extension of some τ–structure $\mathcal{B}$ and*
- *$\mathcal{A}\in\mathcal{P}$ iff $\mathcal{B}\in\mathcal{Q}$.*

2. $\mathcal{P}$ is qffr(σ',τ')–reducible to $\mathcal{Q}$ *($\mathcal{P}\le^{qffr}_{\sigma',\tau'}\mathcal{Q}$) if there is a qffr$(\sigma',\tau')$ from $\mathcal{P}$ to $\mathcal{Q}$.*

Example 2 (continued). In order to obtain the mentioned reduction from 2–SAT, we extend our graph signature σ_G by an additional function symbol *pair*, and consider graphs on which *pair* maps vertices pairwise onto each other, i.e., $pair(pair(v))=v$, for all vertices v. Interpreting $pair(v)$ by $\phi^1_{pair,0,1}(x)=\phi^1_{pair,1,0}(x)=V(x)$, $t^1_{pair,0,1}(x)=t^1_{pair,1,0}(x)=x$, *pair* maps corresponding vertices onto each other, and setting $\phi^2_{pair,1,0}(x)=\phi^2_{pair,0,0}(x)=\neg V(x)$ $t^2_{pair,1,0}(x)=t^2_{pair,0,0}(x)=0$, we make sure that *pair* maps all other objects onto *nil*. This extends our interpretation to a reduction from the set of satisfiable formulas in 2–CNF to the set of those graphs with pairing function, in which no pair of vertices lies on one cycle.

Next we want to prove a closure property of Monadic**NLIN** wrt qffrs.

Proposition 4 *Let $I = (c, \bar{c}, (\phi^r_{gjk}), (t^r_{gjk}))$ be a qffi of τ in σ, and let $\Phi = \exists X_1 \cdots \exists X_n \forall x \varphi$ be a τ–formula, where φ is quantifier–free. Then there is a σ–formula $\Phi_I = \exists \overline{Y} \forall y \psi$, with quantifier–free ψ, such that for every σ–structure $\mathcal{A}$ it holds that $\mathcal{A} \models \Phi_I \iff I(\mathcal{A}) \models \Phi$.*

Before proving the proposition, we derive the closure property from it:

Theorem 1. *If $\mathcal{Q} \in$ Monadic$\mathbf{NLIN}_{\tau,\tau'}$ and $\mathcal{P}$ is a σ–property such that $\mathcal{P} \leq^{qffr}_{\sigma',\tau'} \mathcal{Q}$ then $P \in$ Monadic$\mathbf{NLIN}_{\sigma,\sigma'}$.*

Proof *of Theorem 1: Let I be a qffi of τ' in σ' and let Φ be a τ'–formula which proves $\mathcal{Q} \in$ Monadic$\mathbf{NLIN}$. Let $\mathcal{A}$ be a σ–structure and $\mathcal{A}'$ be a compatible σ'–extension of $\mathcal{A}$. Then $I(\mathcal{A}')$ is a compatible τ'–extension of some τ–structure $\mathcal{B}$, and it holds that*

$$\mathcal{A} \in \mathcal{P} \iff \mathcal{B} \in \mathcal{Q} \iff I(\mathcal{A}') \models \Phi \iff \mathcal{A}' \models \Phi_I.$$

Thus Φ_I characterises $\mathcal{P}$, hence $\mathcal{P} \in$ Monadic$\mathbf{NLIN}_{\sigma,\sigma'}$. □

Proof of Proposition 4:
We want Φ_I to hold in $\mathcal{A}$ iff Φ holds in $I(\mathcal{A})$, whose universe consists of c copies of A, the universe of $\mathcal{A}$. Therefore, we replace the first–order part of Φ, $\forall x \varphi$, by the formula $\forall y \bigwedge_{j=1}^{c} \psi_j$, with the intention that $\psi_j(a)$ holds in $\mathcal{A}$ iff $\varphi(a, j)$ holds in $I(\mathcal{A})$. In order to construct ψ_j from φ, we replace every atom by the formula which interprets it according to I. Let us first consider an atom without nested terms. This can be of one of the following forms.

- $X_i(x)$. We want to express that $(a, j) \in X_i$. To this end we replace the variable X_i by c variables $Y_{i,1}, \ldots, Y_{i,c}$ with the intended interpretation $(a, j) \in X_i \iff a \in Y_{i,j}$. Therefore the atom $X_i(x)$ is replaced by $Y_{i,j}(y)$.

Thus our formula Φ_I is of the form $\exists Y_{1,1} \cdots \exists Y_{c,n} \forall y \bigwedge_{j=1}^{c} \psi_j$.

- $X_i(fx)$, for some $f \in \tau$. We want to express that $f(a, j) \in X_i$. Since $f(a, j)$ may reside on any level k of $I(\mathcal{A})$, we have to consider all these cases. The formulae ϕ^r_{fjk}, where $k = 1, \ldots, c$, $r = 1, \ldots, c_{fjk}$, partition A, so we can write $\bigvee_{k=1}^{c} \bigvee_{r=1}^{c_{fjk}} (\phi^r_{fjk}(y) \wedge Y_{ik}(t^r_{fjk}(y)))$.
- $fx = x$. This can be true for (a, j) only if $f(a, j)$ also lies on level j, i.e., if one of $\phi^1_{fjj}, \ldots, \phi^{c_{fjj}}_{fjj}$ holds for a. Thus we replace the atom $fx = x$ by the formula $\bigvee_{r=1}^{c_{fjj}} (\phi^r_{fjj}(y) \wedge y{=}t^r_{fjj}(y))$.
- $fx = gx$. This is replaced by $\bigvee_{k=1}^{c} \bigvee_{r=1}^{c_{fjk}} \bigvee_{s=1}^{c_{gjk}} (\phi^r_{fjk}(y) \wedge \phi^s_{gjk}(y) \wedge t^r_{fjk}(y){=}t^s_{gjk}(y))$.

For atoms containing nested terms, the formulae are similar, although more complex, due to the necessary iteration of case distinctions: e.g., in order to express that $e(f(a,j)) = g(h(a,j))$, we will have to consider all possible combinations of intermediate levels, i.e., of the levels of $f(a,j)$ and $h(a,j)$, as well as all possible target levels. The following lemma (proof omitted) handles all this.

Lemma 5. *For every τ–term t and every $j,k \leq c$ there are a finite index set R_{tjk}, σ–formulae χ^r_{tjk} and σ–terms s^r_{tjk}, $r \in R_{tjk}$, such that for every σ–structure $\mathcal{A}$ and every τ–term t:*

1. *for every $j \leq c$, the sets $(\{a \in A \mid \mathcal{A} \models \chi^r_{tjk}(a)\})_{r,k}$ form a partition of A, and*
2. *for every $a \in A$, every r,k*
   if $\mathcal{A} \models \chi^r_{tjk}(a)$ then $I(\mathcal{A}) \models t((a,j)) = (s^r_{tjk}(a), k)$.

We can now derive Proposition 4: for each atom which contains $t(x)$, we proceed in the same way as in the corresponding atom for fx, but instead of $\{1, \ldots, c_{fjk}\}$, ϕ^r_{fjk}, and t^r_{fjk}, we use R_{tjk}, χ^r_{tjk}, and s^r_{tjk}, as given by the lemma. □
Next, we introduce a notion of completeness for Monadic**NLIN**.

Definition 6 (Monadic**NLIN**–**completeness).**
Let $\mathcal{Q}$ be a τ–property. $\mathcal{Q}$ is Monadic**NLIN**–complete wrt qffrs *if there is $\tau \subseteq \tau' \subseteq \tau^{succ}$ such that*

- $\mathcal{Q} \in$ Monadic$\mathbf{NLIN}_{\tau,\tau'}$ *and*
- *for any σ-property $\mathcal{P} \in$* Monadic**NLIN** *it holds that $\mathcal{P} \leq^{qffr}_{\sigma^{succ},\tau'} \mathcal{Q}$.*

The existence of complete problems allows a succinct characterisation of Monadic**NLIN**: denote by $Cl_{\tau'}(\mathcal{Q}) := \{\mathcal{P} \mid \mathcal{P} \leq^{qffr}_{\sigma^{succ},\tau'} \mathcal{Q}\}$ the *τ'-closure of a τ-property $\mathcal{Q}$.*

Proposition 6 *Let $\mathcal{Q}$ be* Monadic**NLIN**–*complete wrt qffrs, the completeness testified by a signature τ'. Then*

$$\text{Monadic}\mathbf{NLIN} = Cl_{\tau'}(\mathcal{Q}).$$

Proof: Directly from Definition 6 and Theorem 1. □

Theorem 2.

1. *CNF–SAT is* Monadic**NLIN**–*complete wrt qffrs.*
2. *This can be testified by $\sigma'_{CNF} := \sigma_{CNF} \cup \{s^{pre}_{junction}, max^{pre}_{junction}\}$.*

Proof:By Proposition 1, CNF–SAT is in Monadic$\mathbf{NLIN}_{\sigma_{CNF},\sigma'_{CNF}}$. For the reductions, we follow the standard method of reducing the model class of an arbitrary Σ^1_1–formula $\Psi = \exists \bar{X} Q^1 x_1 \cdots Q^n x_n \, \psi(\bar{X}, \bar{x})$ (with $Q^i \in \{\exists, \forall\}$ and quantifier–free ψ) in a relational signature σ to SAT: there, to any σ–structure $\mathcal{A}$ is associated a Boolean formula $\psi_{\mathcal{A},\Psi} = J^1_{a \in U_\mathcal{A}} \cdots J^n_{a \in U_\mathcal{A}} \psi(\bar{X_i a}, \bar{a})$ where $J^i \in \{\bigvee, \bigwedge\}$, the $X_i \bar{a}$ are understood to be Boolean variables and the remaining atoms are assigned truth values according to the evaluation in $\mathcal{A}$. Here, we have to show that, if Ψ is a Monadic**NLIN**–formula, an analogous mapping from $\mathcal{A}'$

to $\psi_{\mathcal{A}',\Psi}$ can be given as a qffr(σ^{succ}, τ'), which means in particular that this qffr defines a *junction*–compatible successor function on $\psi_{\mathcal{A}',\Psi}$.

Let $\mathcal{P}$ be defined by $\Psi = \exists X_1 \cdots \exists X_l \forall x \psi(\bar{X}, x)$. We view Ψ as a formula in σ^{succ}, even if not all symbols from σ^{succ} appear in Ψ. Wlog we assume that $\psi(\bar{X}, x)$ is given in CNF, i.e. $\psi(\bar{X}, x) = \bigwedge_{i=1}^{s} \psi^i(\bar{X}, x) = \bigwedge_{i=1}^{s} \bigvee_{j=1}^{p_i} \psi^{ij}(\bar{X}, x)$ with clauses $\psi^i(\bar{X}, x)$ and atomic or negated atomic formulae $\psi^{ij}(\bar{X}, x)$. To define $\psi_{\mathcal{A}',\Psi}$, we have to specify its universe and to define the signature $\{junction, var, neg, 0, 1\} \cup \{s^{pre}_{junction}, max^{pre}_{junction}\}$ in terms of σ^{succ}; the particulars of this qffi will depend, of course, on Ψ.

For each element $a \in A$, the target CNF–formula $\psi_{\mathcal{A}',\Psi}$ will have a Boolean variable $v_q(a)$ for every set variable X_q, and a clause $c_i(a)$ for every disjunction ψ^i. A clause $c_i(a)$ has p_i occurences, one for each ψ^{ij}. Furthermore, there will be elements 0 and 1. We therefore take $c := l + s + 2 + \sum_{i=1}^{s} p_i$ copies of the universe. We now have to provide formulae and terms defining the functions: we only give two examples here, one for the function var and one for the successor $s^{pre}_{junction}$:

In the target structure, an occurrence o is represented as a pair (a, h), where h encodes the pair (i, j) such that o corresponds to $\psi^{ij}(a)$. This occurrence can be a Boolean variable $x_q(b)$, represented by a pair (b, g). This is the case if $\psi^{ij}(x)$ is $X_q(t(x))$, and $t(a) = b$. For those pairs ij and the appropriate g, we thus can write $t_{var,h,g}(x) = t(x)$ without any further case distinction expressed by a quantifier–free formula. On the other hand, if $\psi^{ij}(x)$ is of the form $t_1(x) = t_2(x)$, the case distinction "$t_1(x) = t_2(x)$ or $t_1(x) \neq t_2(x)$ controls if var maps (a, h) to 0 or to 1.

This example illustrates that for the elements (a, h) of the target structure, the index h encodes the syntactical structure of the matrix of Ψ. Accordingly, $t_{s^{pre}_{junction},h,g}(x)$ depends on h and g: if, for example, h encodes $(i, p_i - 1)$ g encodes (i, p_i) then the occurrence (a, g) is the direct successor of the occurrence (a, h) and, therefore, $t_{s^{pre}_{junction},h,g}(x) = x$. If, however, h encodes (s, p_s) and g encodes 1 then $t_{s^{pre}_{junction},h,g}(x) = suc(x)$, for some $suc \in \sigma^{succ} \setminus \sigma$. □

We close this section with a result on the transitivity of qffrs.

Proposition 7

1. *Let σ, τ and ρ be functional signatures and I_1 be a qffi of τ in σ and I_2 be a qffi of ρ in τ. Then the composition of I_1 and I_2 can be given as a qffi, i.e. there is a qffi I_3 from ρ in σ such that for any σ-structure $\mathcal{A}$ it holds that $I_3(\mathcal{A}) = I_2(I_1(\mathcal{A}))$.*
2. *Let $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{R}$ be σ–,τ– and ρ–properties respectively. If $\mathcal{P} \leq^{qffr}_{\sigma',\tau'} \mathcal{Q}$ and $\mathcal{Q} \leq^{qffr}_{\tau'',\rho'} \mathcal{R}$ and $\tau' \subseteq \tau''$ then $\mathcal{P} \leq^{qffr}_{\sigma',\rho'} \mathcal{R}$.*

Proof: 1. can be checked using Lemma 5, 2. follows easily from 1. □

4 Nonexpressibility results

In this section, we will show that neither the class of connected graphs nor the class of Hamiltonian graphs are in Monadic**NLIN**. For the proof we will make use of Schwentick's result that, for relationally represented graphs, connectivity cannot be defined in monadicΣ^1_1, even in the presence of a built–in linear order relation [Sch96]. We have to translate this result into our setting, which differs from the one of [Sch96] mainly in that here, graphs are represented as *functional* structures, in particular, quantification over (sets of) edges is possible. Therefore, we will use Schwentick's result in a more specific form: no formula of monadic$\Sigma^1_1(<, E)$ can distinguish the set of all (directed) cycles from the set of all disjoint unions of (directed) cycles ([Sch96], Corollary 19).[8] In a cycle, we can represent each edge by its starting point, so quantification over edges can be replaced by quantification over vertices.[9]

Theorem 3. *There is no set of directed graphs in* Monadic**NLIN** *which contains all cycles, but no disjoint union of more than one cycle.*

Proof (sketch): Let σ be the functional signature for graphs (cf. Example 1.1). Given a formula $\Phi = \exists M_1 \cdots \exists M_r \, \forall x \, \varphi(x)$ over σ_{succ}, we construct a monadicΣ^1_1–formula $\hat{\Phi}$ over the relational signature $\{E, <\}$, such that the following holds: for every *cycle graph*, i.e. every graph G which is the disjoint union of simple cycles, and for every order relation $<$ on the vertices of G, there is an admissible functional representation G' of G, such that $G' \models \Phi$ iff $\langle G, < \rangle \models \hat{\Phi}$. It follows from [Sch96] that $\hat{\Phi}$ cannot distinguish cycles from unions of cycles, hence neither can Φ.

Given an order relation $<$ on a cycle graph G we take as functional representation of G the unique structure G', in which all successor functions are in accordance with $<$, i.e., they all induce the order $<$ on the vertices of G.

When interpreting Φ over functionally represented graphs, the variable x can be instantiated by vertices and edges (or by *nil*, but we will ignore this special case here). In the target formula (which is to be interpreted over relationally represented graphs), we have to express both, the properties expressed by φ for *vertices*, and those for *edges*. We will use each vertex to represent the egde leaving it – which in a cycle graph is unique. Accordingly, we develop two formulas, $\varphi^V(y)$ and $\varphi^E(y)$ from $\varphi(x)$. In the former, we assume y to represent a vertex, in the latter an edge. For every set variable M_i we introduce two set variables, M_i^V, M_i^E, where $M_i^V(y)$ is intended to indicate that the *vertex* represented by y is in the set M_i, similarly, $M_i^E(y)$ for the *egde* represented by y. For the construction of φ^V and φ^E from φ we form, for every atom α which occurs in $\varphi(x)$, two formulae, α^V and α^E, which, when interpreted in $\langle G, < \rangle$ express the same property as α does over G'. It will then hold that $\langle G, < \rangle \models \exists M_1 \cdots \exists M_r \forall x \varphi(x)$ iff $G' \models \exists M_1^V \cdots \exists M_r^V \exists M_1^E \cdots \exists M_r^E \forall y (\varphi^V(y) \wedge \varphi^E(y))$.

[8] Although the results in [Sch96] are stated for undirected graphs, the proofs also work in the directed setting.

[9] This is the reason why we don't use the inexpressibility result for built–in successor [dR87]: the graphs there are more complicated.

The construction of α^V and α^E from α is not difficult, but, due to the many different possibilities for α, rather tedious. We therefore only demonstrate the idea by way of a few simple examples. We will make free use of abbreviations, such as max or $\lessdot$, the latter denoting the *direct successor* relation relative to $<$.
• Let α be the atom $M_i(tail(x))$. If x is a vertex then $tail(x) = nil$. Since we can assume, without loss of generality, that $nil \notin M_i$, for any i, $M_i(tail(x))$ is false, and we set $\alpha^V = \bot$. If x is an edge, then we want $\alpha^E(y)$ to hold for the vertex y representing x iff α holds for x. Since an edge is represented by its tail vertex, we can choose $\alpha^E(y)$ as $M_i^E(y)$.
• Let $\alpha(x)$ be $s^{post}_{head}(s^{pre}_{tail}(x))$=$max^{pre}_{head}$. Note that all successor functions alternate between vertices and edges, the preorder successors start with a vertex and end with an edge. Since all successors are compatible with $<$, max^{pre}_{head} is the edge arriving at the $<$–maximal vertex, max. Furthermore, in order for $\alpha(x)$ to be true, x must be an edge, and then $s^{pre}_{tail}(x)$ is the vertex b following $tail(x)$ in $<$. $s^{post}_{head}(b)$ is the edge arriving at b's direct successor in $<$, it follows that $b \lessdot max$. Altogether, $\alpha(x)$ expresses that the edge x leaves the predecessor of the predecessor of max, and we obtain $\alpha^E(y) = \exists\, z\, y \lessdot z \lessdot max$ (and $\alpha^V(y) = \bot$).
• For our most complex example, let $\alpha(x)$ be $s^{post}_{head}(s^{pre}_{tail}(x))$=$s^{post}_{tail}(s^{pre}_{head}(x))$. Again, if x is a vertex, $s^{post}_{head}(s^{pre}_{tail}(x))$ is the next vertex in the cycle. $s^{pre}_{head}(x)$ is the edge $e = (u, x)$ which arrives at x, and $s^{post}_{tail}(e)$ is u, the vertex preceding x in the cycle. Thus $\alpha^V(y) = \exists u \exists v(E(u, y) \wedge E(y, v) \wedge u{=}v)$. If x is an edge (y, v_1) then $s^{pre}_{tail}(x)$ is the successor z_1 of y in the order $<$, $s^{post}_{head}(s^{post}_{head}(z_1))$ is z_2, the successor of z_1 in $<$, therefore $s^{post}_{head}(z_1)$ is the edge (z_3, z_2). $s^{pre}_{head}(x)$ is the successor v_2 of v_1 in $<$, $s^{post}_{tail}(v_2)$ is the edge starting from v_2's successor in $<$, v_3. Thus $\alpha^E(y)$ can be chosen as the formula
$\exists z_1 \exists z_2 \exists z_3 \exists v_1 \exists v_2 \exists v_3\, E(y, v_1) \wedge y \lessdot z_1 \lessdot z_2 \wedge E(z_3, z_2) \wedge v_1 \lessdot v_2 \lessdot v_3 \wedge z_3{=}v_3$. □

Corollary 1. *The following sets are not in* Monadic**NLIN***:*
- *CONN, the set of all connected graphs.*
- *HAM, the set of all Hamiltonian graphs,*
- *EULER, the set of all Eulerian graphs.*

5 Discussion

Functional structures seem a appropriate representation of *pointer structures* as used in algorithm design: Every object has a finite number of pointers to other objects; this can conveniently be modeled by a corresponding number of functions. This representation of structures is abstract enough to allow reasoning in terms of combinatorial or algebraic structures. On the other hand, it is explicit enough to allow algorithmic reasoning without going down to the level of string encodings.
With the notion of a *compatible successor* we developed a way of ordering the elements of a functional structure in a meaningful way, without overspecifying the order. Similar to, but somewhat stronger than, the local order on the neighbours of a vertex, as used, e.g., in [Cou97], compatible successors enable us to express

the systematic exploration of a structure and thus to model algorithms even in logics which are too weak to express order. Since some algorithms may use several different orders (e.g., adjacency lists according to *tail* vertices and adjacency lists according to *head* vertices), we equipped our structures with several such successors. Of course, we could have defined Monadic**NLIN** equivalently as Monadic$\mathbf{NLIN}_{\sigma^{succ}}$, however, we have several reasons for the introduction of intermediate signatures σ':

- Although we have no concrete example, we expect that there are problems which can only be expressed in Monadic**NLIN** with both, pre– and post–compatible successor functions over structures in which the function trees have unbounded width and depth.
- The multitude of choices of successor functions (and associated *min* and *max* constants), for every signature σ, reflects the situation in algorithm design, where the input representation may vary according to the suitability for particular algorithms. Since we have no reason to prefer one successor to the other, we leave the choice to the "user", i.e., to whoever writes a formula.
- The introduction of σ' is suggested by the proof of the closure property 1. It means that we only have to interpret those successors which are really used in the Monadic**NLIN**–formula defining the target property.
- With τ^{succ} instead of τ' in the definition of completeness, we would not have been able to prove CNF–SAT complete, since it seems that not all successor functions for CNF–formulas can always be interpreted in a quantifier–free way.

Our notion of quantifier–free functional interpretation and reduction should be of use also in other contexts. Being quantifier–free, these reductions are efficiently computable, and thus can be used within low–complexity classes, such as DLIN, in fact, they are very closely related to the *affine* reductions of [GS99]. On the other hand, they seem powerful enough to simulate most of the *sort–lin–*reductions presented in [Cre95]. Thus our class Monadic**NLIN** is a logically defined analogon to the class of *SAT–easy* problems investigated there.

Monadic**NLIN** is a computationally meaningful class. This is shown by the completeness of CNF–SAT, as well as by the definition: Monadic**NLIN**–algorithms have a certain locality, their only means of exploring the structure are the successor functions, whereas general **NLIN**–algorithms can guess arbitrary unary functions, thus potentially accessing every object from every other object. Beyond that, the study of Monadic**NLIN** suggests notions and methods which seem to be of general interest in the logical investigation of functional structures and linear time.

Our negative result in Section 4 shows that, e.g., Hamiltonian graphs are strictly harder than satisfiable CNF–formulas: they are not in Monadic**NLIN**, hence not quantifier–free reducible to CNF–SAT. The method used in the proof can be generalised into a notion of reduction between sets of relational and sets of functional structures.

We conclude the paper by listing just a few questions that arise directly from our work.

1. What is the precise computational meaning of the class Monadic**NLIN**? Does it coincide with the closure of CNF-SAT under other, computationally defined, reductions?
2. What is the relation of Monadic**NLIN** to other monadic logics such as MonadicΣ^1_1?
3. What about the expressibility of other logics over functional structures with compatible successors?

References

[Cos93] S. Cosmadakis. Logical reducibility and monadic NP. In *Proc. 34th IEEE Symp. on Foundations of Computer Science*, pages 52–61, 1993.

[Cou94] B. Courcelle. The monadic second order logic of graphs vi: On several representations of graphs as relational structures. *Discrete Applied Mathematics*, 54:117–149, 1994.

[Cou97] B. Courcelle. On the expression of graph properties in some fragments of monadic second-order logic. In N.Immerman and P.Kolaitis, editors, *Descriptive Complexity and Finite Models*, pages 33–62. American Mathematical Society, 1997.

[Cre95] Nadia Creignou. The class of problems that are linearly equivalent to Satisfiability or a uniform method for proving NP-completeness. *Theoretical Computer Science*, 145(1–2):111–145, 10 July 1995.

[dR87] M. de Rougemont. Second-order and inductive definability on finite structures. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 33:47–63, 1987.

[Fag74] R. Fagin. Generalized first–order spectra and polynomial–time recognizable sets. In R. M. Karp, editor, *Complexity of Computation, SIAM-AMS Proceedings, Vol. 7*, pages 43–73, 1974.

[Fag75] R. Fagin. Monadic generalized spectra. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 21:89–96, 1975.

[GO94] E. Grandjean and F. Olive. Monadic logical definability of NP–complete problems. *In Proc. 1994 of the Annual Conference of the EACSL*, pages 190–204, 1994. Extended version submitted.

[GO96] E. Grandjean and F. Olive. Monadic logical definability of NP-complete problems. submitted, 1996.

[Gra94a] E. Grandjean. Invariance properties of RAMs and linear time. *Computational Complexity*, 4:62–106, 1994.

[Gra94b] E. Grandjean. Linear time algorithms and NP-complete problems. *SIAM Journal of Computing*, 23:573–597, 1994.

[Gra96] E. Grandjean. Sorting, linear time and the satisfiability problem. *Annals of Mathematics and Artificial Intelligence*, 16:183–236, 1996.

[GS99] E. Grandjean and T. Schwentick. Machine-independent characterizations and complete problems for deterministic linear time. In preparation, 1999.

[Imm99] N. Immerman. *Descriptive and Computational Complexity.* Springer, 1999.

[Sch96] T. Schwentick. On winning Ehrenfeucht games and monadic NP. *Annals of Pure and Applied Logic*, 79:61–92, 1996.

Directed Reachability: From Ajtai-Fagin to Ehrenfeucht-Fraïssé Games *

Jerzy Marcinkowski

jma@tcs.uni.wroc.pl

Institute of Computer Science
University of Wrocław,
ul. Przesmyckiego 20
51-165 Wrocław, Poland

Abstract

In 1974 Ronald Fagin proved that properties of structures which are in $\mathcal{NP}$ are exactly the same as those expressible by existential second order sentences, that is sentences of the form $\exists \vec{P} \phi$, where $\vec{P}$ is a tuple of relation symbols. and ϕ is a first order formula. Fagin was also the first to study monadic $\mathcal{NP}$: the class of properties expressible by existential second order sentences where all quantified relations are unary. In their very difficult paper [AF90] Ajtai and Fagin show that directed reachability is not in monadic $\mathcal{NP}$.

In [AFS97] Ajtai, Fagin and Stockmeyer introduce closed monadic *$\mathcal{NP}$: the class of properties which can be expressed by a kind of monadic second order existential formula, where the second order quantifiers can interleave with first order quantifiers. Among other results they show that directed reachability is expressible by a formula of the form $\exists P \forall x \exists P_1\ \phi$, where P and P_1 are unary relation symbols and ϕ is first order. They state the question if this property is in the positive first order closure of monadic $\mathcal{NP}$, that is if it is expressible by a sentence of the form $\vec{Q}x \exists \vec{P} \phi$, where $\vec{Q}x$ is a tuple of first order quantifiers and $\vec{P}$ is a tuple of unary relation symbols.*

In this paper we give a negative solution to the problem.

1 Introduction

In 1974 Ronald Fagin proved that the properties of structures which are in $\mathcal{NP}$ are exactly the same as those expressible by existential second order sentences, known also as Σ_1^1 sentences. Such a sentence has the form $\exists \vec{P} \phi$, where $\vec{P}$ is a tuple of relation symbols and ϕ is a first order formula.

Fagin was also the first to study *monadic* $\mathcal{NP}$: the class of properties expressible by an existential second order sentence where all the quantified relations are unary. The first reason to study this class was the belief that it could serve as a training ground for attacking the "real problems" like whether $\mathcal{NP}$ equals

* Research supported by the Polish KBN grant 8T11C02913

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 338–349, 1999.

co-$\mathcal{NP}$: it is not hard to show ([S95]) that monadic $\mathcal{NP}$ is different than monadic co-$\mathcal{NP}$. But despite of its simple syntax monadic $\mathcal{NP}$ contains also $\mathcal{NP}$-complete problems, including 3-colorability.

A big part of the research in the area of monadic $\mathcal{NP}$ is devoted to the possibility of expressing different variations of graph connectivity. Already Fagin's proof that monadic $\mathcal{NP}$ is different from monadic co-$\mathcal{NP}$ is based on the fact that connectivity of undirected graphs is not expressible by a sentence in monadic Σ^1_1, while non-connectivity obviously is. Then de Rougemont [dR87], Fagin, Stockmeyer and Vardi [FSV95] and Schwentick [S95] proved that connectivity is not in monadic $\mathcal{NP}$ even in the presence of various built-in relations. A closely related topic is reachability: Consider a graph with two constants *source* and *sink*. Then it has the property of reachability if there is a path from the source to the sink. As observed by Kanellakis this property for undirected graphs is expressible in monadic Σ^1_1. But, as Ajtai and Fagin show in their very difficult paper [AF90] directed reachability is not in monadic $\mathcal{NP}$ (their proof was then simplified in [AF97]).

As we said, connectivity is not in monadic $\mathcal{NP}$. But since reachability is in this class, connectivity is expressible by a formula of the form $\forall x \forall y \exists \vec{P} \phi$. This observation leads to the study of *closed monadic* $\mathcal{NP}$: the class of properties expressible by a sentence of the form $\vec{Q}\phi$ where ϕ is quantifier free, and $\vec{Q}$ is a quantifier prenex, containing alternating first order and monadic second order existential quantifiers.

In [AFS97] and [AFS98] Ajtai, Fagin and Stockmeyer argue that the closed monadic $\mathcal{NP}$ is at least as interesting object of study as monadic $\mathcal{NP}$: it is still a subclass of $\mathcal{NP}$, is defined by simple syntax, and is closed with respect to first order quantification. They consider a hierarchy inside closed monadic $\mathcal{NP}$, with respect to the number of alternations between first order and second order quantification, and define *positive first order closure of monadic* $\mathcal{NP}$, as the class of properties expressible by a sentence of the form $\vec{Q}x \exists \vec{P} \phi$, where $\vec{Q}x$ is a tuple of first order quantifiers and $\vec{P}$ is a tuple of unary relation symbols. They show a (very artificial) graph property which is expressible by a sentence of the form $\exists \vec{P}_1 \vec{Q} x \exists \vec{P}_2 \phi$, but neither is not in positive first order closure of monadic $\mathcal{NP}$ nor is expressible as a Boolean combination of properties from this class. They also show that directed reachability is expressible by a formula of the form $\exists P \forall x \exists P_1 \phi$, and state a question whether it is in the positive first order closure of monadic $\mathcal{NP}$.

In this paper we show how to modify the argument from [AF90] to answer the last question negatively.

2 Ajtai-Fagin Graphs

In this section we give a very brief sketch of Ajtai-Fagin's proof of:

Theorem 1. *[AF90] Directed reachability is not expressible by a formula in monadic* Σ^1_1.

Let $\Psi = \exists \vec{P} \phi$ be a formula of monadic Σ_1^1, where $\vec{P}$ is a tuple of $\bar{r}$ unary relations. We assume that ϕ is a first order formula in the prenex form and that this prenex contains r quantifiers.

In order to show that Ψ does not express directed reachability the authors consider, for a given natural number n, a directed graph G defined by the following probabilistic procedure. The set of vertices of G will be $V = \{v_1, v_2 \dots v_n\}$, with v_1 being the source and v_n being the sink. For each $1 \leq i \leq n-1$ there is an edge in G from v_i to v_{i+1}. Such edges are called *forward edges*. For each pair $1 \leq i < j \leq n$ there exists, with probability p, an edge in G leading from v_j to v_i. The probability p depends only on n and on r. Such edges are called *backedges*.

It is clear that there exists a directed path from the source to the sink in G. Call G_k the graph being the result of removing from G the forward edge from v_k to v_{k+1}. Obviously none of the graphs G_k has the property of directed reachability.

Ajtai and Fagin give a very difficult proof of the following:

Theorem 2. *For every $\varepsilon > 0$ and $\bar{r}$ there exists n such that, with probability at least $1 - \varepsilon$ the constructed above graph G has the following property (*):*

(*) For every coloring of G with $\bar{r}$ colors, there exists a number k such that if we color G_k in the same way as G, then the duplicator has a winning strategy in the r-round Ehrenfeucht-Fraïssé game on G and G_k.

See the next Section for a brief introduction to Ehrenfeucht-Fraïssé games.

Notice, that G and G_k have the same set of vertices so it makes sense to talk about "the same way of coloring". Notice also that it follows from the last theorem that a graph G with the property (*) exists. From now on we will treat G as fixed.

Once Theorem 2 is proved it is straightforward to show Theorem 1. Suppose there is a formula Ψ in monadic Σ_1^1 expressing directed reachability. Since Ψ is valid in G there exists a coloring of G satisfying ϕ. Take G_k given by the property (*) and color it as G is colored. To get a contradiction we only need to show that ϕ is valid in the (colored) G_k. But since ϕ is in a prenex form with r quantifiers this follows from Theorem 2 and Theorem 3. □

3 The Games

The most standard tool in proving results about non-expressibility of properties of structures in various logics are Ehrenfeucht-Fraïssé games. The simplest of them corresponds to formulae of first order logic.

Definition 1. *Consider the following* r-round first order Ehrenfeucht-Fraïssé game.

It is played by two players, the spoiler and the duplicator, on two structures G_0 and G_1 (it is good to think that G_0 and G_1 are colored graphs, possibly with constants $c_1, c_2 \dots c_l$). There are r rounds. In the i-th round the spoiler selects

one point in one of the graphs, and calls this point p_i if it is selected from G_0 or q_i if it is selected from G_1. Then, still in the i-th round, the duplicator selects one point in the other graph and calls it q_i, or p_i, respectively.

We say that the duplicator wins, *if after r rounds the structures* $G_0 \cap \{p_1, p_2 \dots p_r, c_1^0, c_2^0, \dots c_l^0\}$ *and* $G_1 \cap \{q_1, q_2 \dots q_r, c_1^1, c_2^1, \dots c_l^1\}$ *where c_i^0 (c_i^1) is the interpretation of the constant c_i in G_0 (G_1) are isomorphic under the function that maps each p_i onto q_i. This means that:*

(i) $q_i = q_j$ ($q_i = c_j^1$ or $c_i^1 = c_j^1$) if and only if $p_i = p_j$ (respectively $p_i = c_j^0$ or $c_i^0 = c_j^0$) for each i, j,

(ii) for each i the color of p_i is the same as the color of q_i, and the color of c_i^0 is the same as the color of c_i^1,

(iii) for each i, j, there is an edge from p_i to p_j (from p_i to c_j^0 and so on) in G_0 if and only if there is an edge from q_i to q_j (respectively from p_i to c_j^0 and so on) in G_1.

We say that the duplicator has a winning strategy *if he can guarantee that he will win, no matter how the spoiler plays. Otherwise we say that* the spoiler has a winning strategy.

Theorem 3. *The property $\mathcal{P}$ is expressible by a first order formula ϕ, with quantifier depth r if and only if, for each choice of structures G_0 and G_1, such that G_0 has the property $\mathcal{P}$ and G_1 does not have, the spoiler has a winning strategy in an r-round first order Ehrenfeucht-Fraïssé game on G_0 and G_1.*

Definition 1 and Theorem 3 come from [E61] and [Fr54].

The strategy of the Ajtai-Fagin's proof above is based on the idea of *Ajtai-Fagin game*: The duplicator selects a graph H with the property $\mathcal{P}$ and the spoiler colors it with some fixed number $\bar{r}$ of colors. In his next move the duplicator provides a graph F, without the property $\mathcal{P}$ and colors it. Then they play an r-round first order Ehrenfeucht-Fraïssé game on the colored graphs H and F. The winner of the game is the winner of the final Ehrenfeucht-Fraïssé game.

Lemma 1 is a simple consequence of Theorem 3:

Lemma 1. *A property $\mathcal{P}$ is expressible by a formula in monadic Σ_1^1 if and only if there is a number $\bar{r}$ of colors and a number r of rounds such that the spoiler has a winning strategy in the Ajtai-Fagin game with $\bar{r}$ colors and r rounds.*

Of course there exists also a version of Ehrenfeucht-Fraïssé game good for monadic Σ_1^1. In such a game on structures H and F the spoiler colors H, the duplicator colors F and then they play the standard first order Ehrenfeucht-Fraïssé game. With the use of Theorem 3 it is easy to prove the Lemma, which comes from [F75]:

Lemma 2. *A property $\mathcal{P}$ in expressible by a formula in monadic Σ_1^1 if and only if there is a number $\bar{r}$ of colors and a number r of rounds such that for each choice of structures F and H such that H has the property $\mathcal{P}$ and F does not have it the spoiler has a winning strategy in the Ehrenfeucht-Fraïssé game on H and F with $\bar{r}$ colors and r rounds.*

Notice that the last game, unlike the first order Ehrenfeucht-Fraïssé game, is not symmetric: in the game on H and F it is only the structure H that the spoiler is allowed to color.

Since in the Ajtai-Fagin game the spoiler colors the structure H before he learns what is F it is much easier to show that duplicator has a winning strategy in Ajtai-Fagin game than in Ehrenfeucht-Fraïssé game. In particular it is very easy to see that for every k it is rather the spoiler than the duplicator who has a winning strategy in the Ehrenfeucht-Fraïssé game on the graphs G and G_k. But the disadvantage of the Ajtai-Fagin game is that it is unclear how it can be extended to the situation where the second order existential quantifiers alternate with first order quantifiers. In particular the technique of [AF90] and [AF97] cannot be used directly even for the proof of the fact that directed reachability cannot be expressed by a sentence of the form $\forall x \exists \vec{P} \phi$, where ϕ is first order.

We will show how to modify the technique of [AF90] so that it can be used together with the following Lemma 3, which is a variation of lemmas from [AFS97].

Definition 2. *By a $l - \bar{r} - r-$Ehrenfeucht-Fraïssé game on structures H and F we will mean the following game. First there are l rounds, almost like in the first order Ehrenfeucht-Fraïssé game, with the only difference that for future convenience we assume that the spoiler selects points in H in odd rounds and points in F in even rounds. Then the spoiler colors H with $\bar{r}$ colors and the duplicator responds by coloring F with $\bar{r}$ colors. Finally they play an r-round first order game on colored H and F, where the points picked in the first l rounds are understood as constants. The winner is the winner of the final first order game.*

Lemma 3. *A property P is expressible in first order closure of monadic Σ^1_1 if and only if there are $l, \bar{r}$ and r such that for each choice of structures F and H such that H has the property $\mathcal{P}$ and F does not have it, the spoiler has a winning strategy in the $l - \bar{r} - r-$Ehrenfeucht-Fraïssé game on H and F.*

4 Ehrenfeucht-Fraïssé Game for Directed Reachability

In this section we will show how to prove that directed reachability is not expressible by a formula in monadic Σ^1_1 using Ehrenfeucht-Fraïssé game instead of Ajtai-Fagin game. In fact we are going to prove a little bit more:

Lemma 4. *Directed reachability is not expressible by a sentence of the form $\forall z \exists x \forall y \exists \vec{P} \phi$, where ϕ is first order. We assume here that $\vec{P}$ is a tuple of $\bar{r}$ unary relations and ϕ is in prenex form, with quantifier depth r.*

There are two reasons why we need the Lemma. First is that it will be used in Section 5 as the first step of induction (see also Section 6). But for this it would be enough to prove the result for sentences of the form $\forall y \exists \vec{P} \phi$. The second reason is that this restricted case can be considered as an example of almost all the tricks needed in Section 5. There we will combine the proof of Lemma 4 with induction to prove Theorem 4, the main result of this paper.

In order to prove Lemma 4 it is enough to show:

Lemma 5. *For every $\bar{r}, r$ there exist two graphs E_1 and F_1 such that E_1 has the property of directed reachability, F_1 does not have this property, and the duplicator has a winning strategy in the $3-\bar{r}-r-$Ehrenfeucht-Fraïssé game on E_1 and F_1.*

Let us define some notations:

Definition 3. *Let H and I be two graphs, each of them with specified source and sink. Let c be a natural number.*

1. *$H+I$ is the graph being a union of disjoint copies of H and I, where the source of H and the source of I are identified and the sink of H and the sink of I are identified. The source of $H+I$ is the common source of H and I and the sink of $H+I$ is the common sink of H and I.*
2. *cH is $H+H+\ldots+H$, where c copies are added.*
3. *HI is the graph being a union of disjoint copies of H and I, where the sink of H and the source of I are identified. The source of HI is the source of H and the sink of HI is the sink of I.*

Now we are ready to define the graphs F and E, the bricks of all our future constructions:

Definition 4. *$F = c(G_1 + G_2 + \ldots G_n)$ where c is huge enough with respect to n, r and $\bar{r}$ (it will soon be clear how huge it must be) and $E = F + G$.*
G and G_i are the graphs from Section 2.

Clearly, E has the property of directed reachability while F does not. Notice our mnemonic notation: the letter E is F with an additional edge. One should remember that the graph G (and so the graphs G_i and graphs E and F) depends on the choice of $\bar{r}$ and r.

To explain what are E and F good for we will prove:

Lemma 6. *The duplicator has a winning strategy in the $0-\bar{r}-r-$Ehrenfeucht-Fraïssé game on E and F.*

This Lemma gives a translation of the original proof of the Ajtai-Fagin result into the language of Ehrenfeucht-Fraïssé games.

Proof:

We describe the strategy of the duplicator. In the first round the spoiler colors the graph E. The duplicator analyzes the colored copy of G in E, and finds k whose existence is guaranteed by Theorem 2. He takes one of the copies of G_k in F and copies on it the coloring of G in E. Then for each copy of G_i in E such that $i \neq k$, he copies its coloring to F. Now he still has many copies of G_k to be colored in F. He cannot just copy the coloring of the respective copies of G_k in H_1 because he has one of them less in F than in E: we mean the one that is already colored as G.

To overcome this problem the duplicator finds the coloring which is the most common among the copies of G_k in E. There are some $m > r$ copies of G_k

colored like this (here one can compute c). He colors $m-1$ copies of G_k in F with this most common color. Then for each copy of G_k colored in E in a way different than the most common one he creates one copy of it in F.

It is now simple to show (with the use of Theorem 2) that the duplicator has a winning strategy in the r-round Ehrenfeucht-Fraïssé game on the colored graphs F and E. □

Now we define the graphs whose existence is claimed in Lemma 5:

Definition 5. *Let F_1 be $c_1(FE+EF)$ and let E_1 be F_1+EE. Again c_1 is a constant which is "huge enough" with respect to n, r and $\bar{r}$.*

Clearly, E_1 has the property of directed reachability and F_1 does not. We will need names to call parts of E_1 and F_1:

Definition 6. *Each copy of EF, or FE or EE in E_1 or F_1 will be called* segment. *In each segment its* first half, *and* second half *are defined in the natural way. The point which is the sink of the first half and the source of the second half will be called* middle *of the segment.*

It is time now for:

Proof of Lemma 5. We will show a winning strategy for the duplicator in the $3-\bar{r}-r-$Ehrenfeucht-Fraïssé game on E_1 and F_1. The first to move is the spoiler. He picks a point $q_0 \in F_1$. By symmetry we may think that it is in one of the segments of the form EF. The duplicator finds the same point in E_1 and names it p_0. Here we use the fact that F_1 is a subset of E_1, so he can take "the same" point.

Now the spoiler picks the point p_1 in E_1. There are three cases:
case 1. The spoiler picks an element p_1 in the segment EE of E_1.
case 2. The spoiler picks an element p_1 in one of the segments EF or in one of the segments FE of E_1, but not in the one where p_0 is located.
case 3. The spoiler picks an element p_1 in the same segment where p_0 is located.

We only consider the case 1 which seems to be the most difficult one for the duplicator. No new arguments are needed for case 2. Also case 3 is easy for the duplicator: he takes a proper q_1 in the same segment where q_0 is.

There are two (almost symmetric) subcases of case 1: either p_1 is in the first half (including the middle) of EE, or in the second half. Let us consider the situation when p_1 is in the first half of EE. Since p_1 is an element of a copy of graph E the duplicator wants q_1 to be also an element of some copy of E. So he takes one of the segments in F_1 which is of the form EF and chooses q_1 to be an element of the first half of it. More precisely, the duplicator chooses q_1 to be the same element of (this another copy of) E as p_1 is.

Now the spoiler picks a point q_2 in F_1. There are three subsubcases: either he takes a point in the same segment where q_0 was picked, or in the one where q_1 is, or in still another one. In the last case the duplicator finds in E_1 a new copy of the segment where q_2 was taken from, and chooses p_2 to be the same

as q_2 element of (different copy of) the segment. In the first (subsub) case the duplicator chooses as q_2 a proper element of the copy of EF where q_0 is located. In the second (subsub)case there are again two possibilities. First is that q_2 is in the first half (including the middle) of the segment. So q_2 is an element of the same copy of E as q_1 is. Then the duplicator chooses as p_2 the same as q_2 element of the copy of E in which p_1 was picked. The second possibility is that q_2 is taken by the spoiler from the second half of the segment where q_1 is. Since q_1 and q_2 are in the same segment the duplicator would like to keep p_2 in the same segment as p_1. So he must pick a point in the second half of the segment where p_1 is located. The half where q_2 is picked is a copy of F and the half where the duplicator wants to place p_2 is a copy of E. Since F is a subset of E it makes sense to say that duplicator chooses p_2 to be the same point in this half as q_2 is.

Now the coloring round. All the (sub)cases are similar. We show how the duplicator can win if p_1 and p_2 are in two halves of the segment EE. The first to move is the spoiler who colors the graph E_1. Now the duplicator colors F_1. First, he copies to the segment of F_1 where q_0 is located the coloring of the segment in E_1 where p_0 is. It is easy since the two segments are isomorphic. Also, he copies to F_1 the coloring of the segments of the form FE in E_1 (one copy for each of them). Then he colors the copy of E where q_1 is, in the same way as the copy of E in which p_1 is located is colored in E_1. To color the second half of the segment with q_1, q_2 he needs something different than just copying: this is because this second half is F in F_1 and E in E_1. He starts from copying the coloring of the copy of G_k where p_2 is located to the copy of G_k where q_2 is. Then, for the remaining nodes of the half he uses the method from the proof of Lemma 6. To finish he needs to color the remaining segments of the form EF in F_1. Like in Lemma 6 he cannot just copy the colorings from the respective segments in E_1: there is one less non-colored segment in F_1 now. So again he finds the most common coloring of segments of the form EF in E_1. Since c_1 is huge enough there are $m > r$ segments colored with the coloring. The duplicator colors like this $m-1$ copies of EF in F_2. Then he makes in F_1 one copy of each of the EF segments colored in F_1 in a way different than the most common.

It is easy to show (with the use of Theorem 2) that after the described coloring the duplicator has a winning strategy in the remaining $r-$round first order Ehrenfeucht-Fraïssé game. □

5 Positive First Order Closure of Monadic Σ^1_1

In this section we prove the main result of this paper:

Theorem 4. *Directed reachability is not expressible by a formula in the positive first order closure of monadic* Σ^1_1.

Proof: Fix $l, \bar{r}$ and r. By Lemma 3 it is enough to show that there exist structures $E_{l,\bar{r},r}$ and $F_{l,\bar{r},r}$, such that $E_{l,\bar{r},r}$ has the property of directed reachability, $F_{l,\bar{r},r}$ does not and that the duplicator has a winning strategy in the $l-\bar{r}-r-$Ehrenfeucht-Fraïssé game on $E_{l,\bar{r},r}$ and $F_{l,\bar{r},r}$.

Definition 7. *Let E_0 be the E from Section 4 and F_0 be the F from Section 4. For $i \geq 1$ we define F_{i+1} as $c_{i+1}(E_iF_i + F_iE_i)$, and E_{i+1} as $F_{i+1} + E_iE_i$, where c_{i+1} is huge enough with respect to n, r and $\bar{r}$.*

As we said in Section 2, the graph G, and so the graphs E_0 and F_0, depend on r and $\bar{r}$.

Again it is clear that E_{i+1} has the property of directed reachability and that F_{i+1} does not have it.

Definition 8. *Each copy of E_iF_i, or F_iE_i or E_iE_i in E_{i+1} or F_{i+1} will be called* i-segment*. In each i-segment its* first half*, and* second half *are defined in the natural way.*

Definition 9. *Assume a tuple of points $p_1, p_2, \ldots p_k$ is selected in E_0, and a tuple of points $q_1, q_2, \ldots q_k$ is selected in F_0. We say that the two tuples are* 1*-corresponding if there is an automorphism f of F_0, such that $f(q_i) = p_i$ for each i. In other words, this means that if q_i is located in some copy of G_s in F_0, for some i and s, then p_i is the same point of some copy of G_s in E_0, and that q_i and q_j belong to the same copy of some G_s in F_0 if and only if p_i and p_j belong to the same copy of some G_s in E_0.*

Definition 10. *Assume a tuple of points $p_1, p_2, \ldots p_k$ is selected in E_i (or F_i), and a tuple of points $q_1, q_2, \ldots q_k$ is selected in another copy of E_i (respectively F_i). We say that the two tuples are i-corresponding if there exists an isomorphism f between the two copies, such that $f(p_j) = q_j$ for each $1 \leq j \leq k$.*

Definition 11. *Assume a tuple of points $p_1, p_2, \ldots p_k$ is selected in E_{i+1}, and a tuple of points $q_1, q_2, \ldots q_k$ is selected in F_{i+1}. We say that the two tuples are $i+1$-corresponding if the conjunction of the following conditions holds:*

1. *If p_j is in the first (second) half of some i-segment then q_j also is in the first (second) half of its i-segment.*
2. *The points p_{j_1} and p_{j_2} are in the same i-segment if and only if q_{j_1} and q_{j_2} are in the same i-segment.*
3. *If $p_{j_1}, p_{j_2}, \ldots p_{j_s}$ are in the same half of some i-segment (that is in some copy of E_i or F_i) then $q_{j_1}, q_{j_2}, \ldots q_{j_s}$ are also in the same half of some i-segment (this follows from (2)) and the tuples $p_{j_1}, p_{j_2}, \ldots p_{j_s}$ and $q_{j_1}, q_{j_2}, \ldots q_{j_s}$ are i-corresponding.*

Lemma 7. *Suppose $k \leq l$ and the tuples of points $p_1, p_2, \ldots p_k$ in E_i, and $q_1, q_2, \ldots q_k$ in F_i are i-corresponding. Then for every coloring of E_i there exists a coloring of F_i, such that the duplicator has a winning strategy in the r-rounds Ehrenfeucht-Fraïssé game on the colored structures E_i and F_i with l constants interpreted in E_i as $p_1, p_2, \ldots p_k$ and in F_i as $q_1, q_2, \ldots q_k$.*

Proof: Induction on i.

If $i = 1$ then one can use the argument from the end of the proof of Lemma 5. We leave it to the reader. Notice that this argument requires an easy modification

to take care about the constants. One should remark here that the "huge enough" c_1 depends on l, the number of constants in the tuple. This is the same l as in $E_{l,\bar{r},r}$ and $F_{l,\bar{r},r}$ in the beginning of this section.

Now the induction step. Assume that the claim holds for some i and consider the graphs E_{i+1} and F_{i+1} with $i+1$-corresponding tuples $p_1, p_2, \ldots p_k$ and $q_1, q_2, \ldots q_k$

This is how to color F_{i+1} for a given coloring of E_{i+1}:

For any i-segment in E_{i+1} which contains some $p_{j_1}, p_{j_2}, \ldots p_{j_s}$ color the i-segment in F_{i+1} containing $q_{j_1}, q_{j_2}, \ldots q_{j_s}$ in the way given by the induction hypothesis. This is possible since the subset of $p_{j_1}, p_{j_2}, \ldots p_{j_s}$ in the first (second) half of the i-segment and the subset of $q_{j_1}, q_{j_2}, \ldots q_{j_s}$ in the first (second) half of the i-segment are i-corresponding.

Then, if none of p_j was in the segment E_iE_i of E_{i+1} then take one of not yet colored segments E_iF_i of F_{i+1}, copy to it the coloring of the first half of E_iE_i and use the hypothesis to color the second half.

What remains to be colored is some number of segments of the form E_iF_i and F_iE_i in F_{i+1}. For this we use the fact that c_{i+1} is huge enough and the trick with the "most common color" from the proof of Lemma 6.

Now the strategy for the duplicator in the r-rounds Ehrenfeucht-Fraïssé game is inherited from the strategies existing by the hypothesis for each of the games on each half of an i-segment respectively. □

Lemma 8. *Consider the following game between the spoiler and the duplicator: in every odd round the spoiler picks a point in F_i and the duplicator responds with a point in E_i and in every even round the spoiler picks a point in E_i and the duplicator responds with a point in F_i. The duplicator wins if the tuples $p_1, p_2, \ldots p_k$ in E_i, and $q_1, q_2, \ldots q_k$ in F_i, of points picked in rounds $1, 2, \ldots k$ respectively, are i-corresponding.*

Then for every i the duplicator has a winning strategy in a game with $2i+1$ rounds.

Proof: Induction on i.

If $i = 1$ then the claim follows from the proof of Lemma 5.

For the induction step, assume that the claim holds for some i and consider the graphs E_{i+1} and F_{i+1}. We want to show a winning strategy for the duplicator in a game with $2i+3$ rounds. In the first round the spoiler takes a point q_1 in F_{i+1}, for example in one of the i-segments of the form E_iF_i. The duplicator responds with p_1 being the same point in E_{i+1}. Then the spoiler can pick a point p_2 in E_{i+1}. The only interesting case is when the point is in the i-segment E_iE_i. Suppose p_2 is in the first half of E_iE_i. Then the duplicator finds a copy of E_iF_i in F_{i+1}, but a different one than where q_1 is located in, and chooses q_2 to be the same point in the first half of the copy of E_iE_i as p_2 is in the first half of E_iE_i.

In the remaining $2i+1$ rounds the duplicator can in most of the cases answer the moves of the spoiler in isomorphic halves of i-segments. In those cases he easily wins. The only action of the spoiler which could require more care from the duplicator is when he takes his points from the second halves of the i-segments

where p_2 and q_2 are located. This is since the half is a copy of E_i in E_{i+1} and a copy F_i in F_{i+1}. But there are only $2i+1$ rounds left now, so winning strategy for the duplicator exists by the hypothesis. □

Now, take $E_{l,\bar{r},r} = E_l$ and $F_{l,\bar{r},r} = F_l$. Theorem 4 follows from Lemmas 7 and 8 □

6 Remark

The existence of graphs H_1 and H_2 such that H_1 has the property of directed reachability, H_2 does not have it, and the duplicator has a winning strategy in the $0-\bar{r}-r-$Ehrenfeucht-Fraïssé game on H_1 and H_2 follows from Theorem 1 and Lemma 2. Moreover, as it was proved by R. Fagin [F97] we can take H_1 to be a path (from source to sink) with backedges, like in the proof of Theorem 1. Then H_2 can be defined as the result of removing one forward edge from H_1.

This fact could be used as induction basis in our proof, instead of Lemma 4. Definition 4 would be skipped then: the bricks of the whole construction would be the unknown graphs H_1 and H_2 instead of E and F. It is funny to see how little we need to know about the actual structure of H_1 and H_2. We do not even need to require that they have the same sets of vertices.

The remark above is almost obvious, and I am ashamed to admit I only have learned it from a letter that Ronald Fagin wrote to me after reading the first version of this paper. Also one of the anonymous referees suggested removing Lemma 4 and using H_1 and H_2 as the basis of induction.

Nevertheless I decided not no reorganize this paper. This is first of all because I find the construction of E and F in Definition 4 very nice. Another reason is that this would require concentrating all difficulties inside the induction step, what would make the paper harder to read.

7 Acknowledgment

The author would like to thank Ronald Fagin and Larry Stockmeyer for their comments on the preliminary version of the paper. I also benefited a lot from discussions with Leszek Pacholski, Lidka Tendera and Wiesiek Szwast.

References

[AF90] M.Ajtai, R.Fagin *Reachability is harder for directed than for undirected finite graphs*, Journal of Symbolic Logic, 55(1):113-150, 1990;

[AF97] S. Arora, R. Fagin *On winning strategies in Ehrenfeucht-Fraïssé games*, Theoretical Computer Science , 174:97-121, 1997;

[AFS97] M.Ajtai, R.Fagin, L.Stockmeyer *The Closure of Monadic $\mathcal{NP}$*, IBM Research report RJ 10092, November 1997;

[AFS98] M.Ajtai, R.Fagin, L.Stockmeyer *The Closure of Monadic $\mathcal{NP}$*, Proc. of 13th STOC, pp 309-318, 1998;

[dR87] M. de Rougemont *Second-order and inductive definiability on finite structures*, Zeitschrift fuer Mathematische Logik und Grundlagen der Mathematik, 33:47-63, 1987;

[E61] A. Ehrenfeucht *an application of games to the completeness problem for formalized theories*, Fund. Math. 49:129-141,1961;

[F75] R. Fagin *Monadic Generalized spectra*, Zeitschrift fuer Mathematische Logik und Grundlagen der Mathematik, 21;89-96, 1975;

[F97] R. Fagin *Comparing the power of games on graphs*, Math. Logic Quarterly 43, 1997, pp. 431-455;

[Fr54] R. Fraïssé *Sur quelques classifications des systemes de relations*, Publ. Sci. Univ. Alger. Ser. A, 1:35-182, 1954;

[FSV95] R. Fagin, L. Stockmeyer, M. Vardi *On monadic $\mathcal{NP}$ vs. monadic co-$\mathcal{NP}$*, Information and Computation, 120(1):78-92, July 1995;

[S95] T. Schwentick *Graph connectivity, monadic $\mathcal{NP}$ and built-in relations of moderate degree*, Proceedings of 22nd ICALP: 405-416,1995;

Fixpoint Alternation and the Game Quantifier

J.C. Bradfield[1,2]

[1] BRICS⋆⋆⋆,
Department of Computer Science, Aarhus University,
Bygning 540, Ny Munkegade,
8000 Århus C, Denmark
[2] LFCS†, Division of Informatics, University of Edinburgh,
Edinburgh, United Kingdom,
EH9 3JZ (email: jcb@dcs.ed.ac.uk)

Abstract. Drawing on an analogy with temporal fixpoint logic, we relate the arithmetic fixpoint definable sets to the winning positions of certain games, namely games whose winning conditions lie in the difference hierarchy over Σ^0_2. This both provides a simple characterization of the fixpoint hierarchy, and refines existing results on the power of the game quantifier in descriptive set theory.

1 Introduction

For several decades, games have been an essential tool for the study of logic, both in mathematical logic and more recently in computer science. Perhaps the most developed application in computer science logic is the use of Ehrenfeucht–Fraïssé games for first-order logic, and the refinements such as pebble games which correspond to finite variable fragments. However, games are also useful in temporal logic, and in particular for the modal mu-calculus. The ability to switch one's point of view between logics, automata and games facilitates many results. In particular, the semantics of the modal mu-calculus can be described by means of a *parity game*, that is, a game in which the winning condition concerns the parity of the highest *index* seen infinitely often in the game. This presentation is equivalent to a presentation via alternating Rabin automata, or via tableaux. In modal mu-calculus, a key issue is the alternation of minimal and maximal fixpoints; in automata, this corresponds to the Rabin index, and in normal form parity games it corresponds to the number of indices.

Games, in the form of Gale–Stewart games, also play an important role in descriptive set theory: they provide a tool with which many of the structure theorems of the classical and effective Borel and Lusin hierarchies can be obtained. The *game quantifier* $\mathcal{G}$ takes a game, defined by its set of winning plays, and returns the set of winning positions; the power of this quantifier is the object

⋆⋆⋆ Danish National Research Foundation Centre for Basic Research in Computer Science

† Laboratory for Foundations of Computer Science

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 350–361, 1999.

of our study. Kechris and Moschovakis showed that a Σ^0_1 game has a Π^1_1 set of winning positions; and Robert Solovay showed that the set of winning positions of a Σ^0_2 game is Σ^1_1-inductive. Thomas John studied Σ^0_3 games, and the characterization is complex, involving higher-type recursion and certain levels of the constructible universe.

If one looks at the previous paragraph with fixpoint glasses on, one notices that Π^1_1 is Π^0_1-inductive, that is, on the first level Σ^μ_1 of the fixpoint alternation hierarchy of arithmetic with fixpoints; and that Σ^1_1-inductive is the second level. It is then natural to ask whether this is coincidence, or whether there is, in arithmetic, a nice relationship between fixpoint alternation and some hierarchy of games, mediated by the game quantifier. One might initially speculate that Σ^0_n games have Σ^μ_n winning positions, but alas this cannot be true. However, the world of Rabin automata and modal mu-calculus provides a suggestion: the complexity of the Rabin or parity condition corresponds to the complexity of the winning plays, and since the modal fixpoint alternation *is* correlated nicely to that, the next obvious thing to do is to try to find a notion of complexity in arithmetic that corresponds to the Rabin index, and then hope that the correlation still holds in the rather different world of arithmetic. The result of such an exploration is that fixpoint complexity of winning positions does indeed correspond to a natural fine hierarchy of arithmetic, in a way that matches well with the finite automata world; and although the result is pure descriptive set theory, the games used in its proof are natural analogues of games developed for the automata and temporal logic world. In fact, the games were formulated in order to establish the modal mu-calculus fixpoint alternation hierarchy with the assistance of descriptive set theory; so it is pleasing to go back and use them to obtain results in descriptive set theory.

2 Preliminaries

2.1 Notations and Basic Definitions

ω is the set of non-negative integers; variables $i, j, \ldots, n$ range over ω. The set of finite sequences of integers is denoted ω^*; finite sequences are identified with integers via standard codings; variables u, v range over ω^*. The set of infinite sequences of integers is ${}^\omega\omega$; variables α, β range over ${}^\omega\omega$. For $\alpha \in {}^\omega\omega$, $\alpha(i)$ is the i'th element of α, and $\alpha(<i)$ is the finite sequence $\langle \alpha(0), \ldots, \alpha(i-1) \rangle$. Concatenation of finite and infinite sequences is written with concatenation of symbols or with $\cdot$, and extended to sets pointwise.

We consider (only) spaces that are the product of copies of ${}^\omega\omega$ and ω; they are given the product topology, where ω carries the discrete topology and ${}^\omega\omega$ itself carries the infinite product topology (in which the basic open sets are the sets $u \cdot {}^\omega\omega$ for every finite sequence u). A *pointset* P is a subset of such a space X; we write variously $P(i, \alpha)$ or $(i, \alpha) \in P$ if X is, for example, $\omega \times {}^\omega\omega$. A pointset is *semi-recursive or* Σ^0_1 iff it is a recursive union of basic opens, in other words given by a semi-recursive set of prefixes: in particular, a pointset $P \subseteq {}^\omega\omega$ is Σ^0_1 iff $P = \bigcup_i N_{\varepsilon(i)}$, where N_k denotes the basic neighbourhood $u_k \cdot {}^\omega\omega$ for some

recursive enumeration $k \mapsto u_k$ of the finite sequences w^*, and ε is a recursive function $\omega \to \omega$, otherwise known as a recursive element of ${}^{\omega}\omega$. A *pointclass* is a set of pointsets; if Λ and Λ' are pointclasses, then $\Lambda \wedge \Lambda'$ is the pointclass $\{ P \cap P' \mid P' \in \Lambda, P' \in \Lambda' \}$ and similarly for $\vee$ and $\neg$; if Λ is a pointclass on $\omega \times \mathrm{X}$, then $\exists^{\omega}\Lambda$ is the pointclass $\{ Q \subseteq \mathrm{X} \mid \exists P \in \Lambda.\, x \in Q \Leftrightarrow \exists i.\, (i,x) \in P \}$; similarly for $\exists^{{}^{\omega}\omega}$. The *Kleene pointclasses* (the arithmetical and analytical hierarchies) are defined by $\Pi^i_j = \neg\Sigma^i_j$; $\Sigma^0_{j+1} = \exists^{\omega}\Pi^0_j$; $\Sigma^1_0 = \Sigma^0_1$; $\Sigma^1_{j+1} = \exists^{{}^{\omega}\omega}\Pi^1_j$; $\Delta^i_j = \Sigma^i_j \cap \Pi^i_j$. Pointsets in the Kleene pointclasses are definable by formulae of first- and second-order arithmetic in the usual prenex normal form.

For completeness we recall that the boldface classes are given by: Σ^0_1 is the class of open sets, and then similarly to the lightface classes; however, we are here concerned mainly with the lightface classes.

Ordinals are ranged over by variables ζ, ξ.

An *(ω-)tree* is a prefix-closed subset of ω^*. If T is a tree, α is an *infinite branch* of T iff $\forall i.\, \alpha(<i) \in T$. The *body* $[T]$ of T is the set of its infinite branches. T is *recursive (etc.)* iff it is recursive (etc.) as a subset of ω via the coding of sequences. The following standard fact will be useful:

Lemma 1. *If $P \subseteq {}^{\omega}\omega$ is Π^0_1, then there is a Π^0_1 tree T such that $\alpha \in P \Leftrightarrow \alpha \in [T]$.*

Proof. If P is Π^0_1, by definition it is $\neg \bigcup_j N_{\varepsilon(j)} = \bigcap_j \overline{N_{\varepsilon(j)}}$ for some recursive ε. Put $T = \{ v \mid \forall j.\, v \cdot {}^{\omega}\omega \subseteq \overline{N_{\varepsilon(j)}} \}$. If $\alpha \in P$, then $\forall i.\, \alpha(<i) \in T$; conversely, if $\alpha \notin P$, then $\alpha \in N_k$ some k, and then there is a prefix $\alpha(<i) \in N_k$. Finally, T is Π^0_1 since the test '$v \cdot {}^{\omega}\omega \subseteq \overline{N_{\varepsilon(j)}}$' is recursive, reducing to 'v does not have $u_{\varepsilon(j)}$ as a prefix', where $u_{\varepsilon(j)}$ is as above.

2.2 Gale–Stewart Games

An infinite game of perfect information, or Gale–Stewart game, on ω, is played between two players, Abelard and Eloise. The players take turns, starting with Eloise, to choose a number, so defining a *play* as an infinite sequence $\alpha \in {}^{\omega}\omega$. The game is defined by a *winning condition* $P \subseteq {}^{\omega}\omega$, a set of sequences; if $\alpha \in P$ (we write also $P(\alpha)$), then Eloise wins the play, otherwise Abelard. A *strategy* for Eloise is a function from partial plays where she is due to move, i.e. finite sequences of even length, to integers, telling Eloise her next move. A *winning strategy* for Eloise is one such that if she follows it, she is guaranteed to win the game no matter how Abelard plays. If u is a partial play in the game P, then $P[u]$ denotes the game $\{ \alpha \mid u\alpha \in P \}$. A *winning position* for Eloise is a partial play u of even length from which Eloise has a winning strategy for $P[u]$; thus Eloise has a winning strategy for the game, or *wins the game*, iff $\langle\rangle$ is a winning position for her.

It is frequently convenient to relax the definition to allow games with *rules* which constrain the choices of the players, and games where the players' turns need not strictly alternate. This is harmless provided that the rules and the turn function are recursive in the partial plays.

For our purposes, it is also useful to permit finite plays where Eloise wins outright at a particular point. A game with such plays can easily be modified to a game with only infinite plays; an important point for us is that we shall always have recursive winning conditions on finite plays.

A game is *determined* if one or other player wins it. By a celebrated theorem of Martin, all games with Δ^1_1 winning conditions are determined. However, Wolfe much earlier proved determinacy for Σ^0_2 games, and it is a generalization of this proof, far easier than Martin's theorem, that will give us our result.

It is convenient to define *cogames* in which Abelard moves first. Now we can extend the definition of Eloise winning position to partial plays u of odd length by saying that u is a winning position in the game P iff Eloise wins the cogame $P[u]$.

If $P \subseteq {}^{\omega}\omega \times \mathrm{X}$, then for each $x \in \mathrm{X}$ the set $\{\, \alpha \mid (\alpha, x) \in P \,\}$ defines a game (we call it $P(\alpha, x)$). The *game quantifier*[1] is defined thus: $\mathcal{G}\alpha.P(\alpha, x)$ is the set $\{\, x \mid$ Eloise wins the game $P(\alpha, x) \,\}$. Although formally defined in terms of strategies, it is intuitively understood as an infinite string of first-order quantifiers:

$$\exists a_0.\, \forall a_1.\, \exists a_2.\, \forall a_3.\, \ldots P(a_0 a_1 \ldots, x).$$

Let Γ be a pointclass on ${}^{\omega}\omega \times \mathrm{X}$; then $\mathcal{G}\Gamma$ is the pointclass

$$\{\, Q \subseteq \mathrm{X} \mid \exists P \in \Gamma.\, Q = \mathcal{G}\alpha.P(\alpha, x) \,\}.$$

The following standard fact (see [6]) will be required:

Lemma 2. *If Γ is a determined pointclass closed under recursive substitution, then $\neg\mathcal{G}\Gamma = \mathcal{G}\neg\Gamma$.*

2.3 Mu-Arithmetic

In [5] Robert Lubarsky studies the logic given by adding fixpoint constructors to first-order arithmetic. This logic is also known as LFP in finite model theory, where it is most studied. The logic ('mu-arithmetic' for short) has as basic symbols the following: function symbols f, g, h; predicate symbols P, Q, R; first-order variables x, y, z; set variables X, Y, Z; and the symbols $\vee, \wedge, \exists, \forall, \mu, \nu, \neg, \in$. As with the modal mu-calculus, $\neg$ can be pushed inwards to apply only to atomic formulae, by De Morgan duality.

The language has expressions of three kinds, individual terms, set terms, and formulae. The individual terms comprise the usual terms of first-order logic. The set terms comprise set variables and expressions $\mu(x, X).\, \phi$ and $\nu(x, X).\, \phi$, where X occurs positively in ϕ. Here μ binds both an individual variable and a set variable; henceforth we shall often write just $\mu X.\, \phi$, and assume that the individual variable is the lower-case of the set variable. We also use $\mu\nu$ to mean 'μ or ν as appropriate'. The formulae are built by the usual first-order construction,

[1] As Springer now like to have LaTeX source, the notation in this version of the paper is constrained by the commonly available packages. Readers who want real notation should see the version on my home page `http://www.dcs.ed.ac.uk/home/jcb/`.

together with the rule that if τ is an individual term and Ξ is a set term, then $\tau \in \Xi$ is a formula.

This language is interpreted over the structure ω of first-order arithmetic. The semantics of the first-order connectives is as usual; $\tau \in \Xi$ is interpreted naturally; and the set term $\mu X.\, \phi(x, X)$ is interpreted as the least fixpoint of the functional $X \mapsto \{\, m \in \omega \mid \phi(m, X) \,\}$ (where $X \subseteq \omega$).

Mu-arithmetic has a prenex normal form [5,2] of the following shape:

$$\tau_n \in \mu X_n . \tau_{n-1} \in \nu X_{n-1} . \tau_{n-2} \in \mu X_{n-2} . \ldots \tau_1 \in \mu\nu X_1 .\, \phi$$

where ϕ is first-order—that is, a string of alternating fixpoint quantifiers, and a first-order body. If we refer to a formula in normal form, we shall refer to its components by this notation.

We define levels of the fixpoint alternation hierarchy similarly to the Kleene hierarchies: first-order formulae are Σ^μ_0 and Π^μ_0, as are set variables. The Σ^μ_{n+1} formulae and set terms are formed from the $\Sigma^\mu_n \cup \Pi^\mu_n$ formulae and set terms by closing under (i) the first-order connectives and (ii) the formation of $\mu X.\, \phi$ for $\phi \in \Sigma^\mu_{n+1}$.

A set $X \in \omega$ is Σ^μ_n if $x \in X \Leftrightarrow \tau(x) \in \Xi(x)$ for some Σ^μ_n set term Ξ. Note that this does *not* mean that X is a fixpoint, only that X is definable via a fixpoint.

A Σ^μ_1 set corresponds to a set definable by an inductive definition over an arithmetic predicate; hence by Kleene's theorem, Σ^μ_1 is equal to Π^1_1. The higher levels of the fixpoint hierarchy have been characterized by Lubarsky [5] in terms of large admissible ordinals involving a generalized reflection principle, devised for the purpose, and whose essential content is the iteration of the idea $\Pi^1_1 = \Pi^0_1$-*IND* $= (\Sigma_1$ on $\Delta^1_1)$; however, there has not been a simple characterization in terms of existing notions.

2.4 Rabin Conditions and Parity Games

Consider a (non-deterministic) finite automaton on which Eloise and Abelard play a Gale–Stewart style game by alternately choosing next states. A *Rabin condition* is a winning condition for the game of the following form:

$$\bigvee_{1 \le i \le n} (\infty G_i \wedge \neg \infty R_i)$$

where G_i and R_i are subsets of states, and ∞X means that the set X is met infinitely often during the play. n is the *Rabin index*.

These *alternating Rabin automata* are important in temporal logic, as they are one characterization of modal mu-calculus. They are equivalent to alternating *parity* automata: a *parity* condition has the form 'for given sets X_i, $1 \le i \le n$, of states, the greatest j such that X_j occurs infinitely often in the play, is even'. The relationship between parity automata and modal mu-calculus is direct [4], as the parity condition corresponds to the statement 'the highest fixpoint variable regenerated infinitely often is a maximal fixpoint'.

The Rabin index, or the number of sets in a parity condition, correspond to the fixpoint alternation in modal mu-calculus, and it is this that inspires both our question and its solution. (Niwiński [7] gives a survey of all the concepts mentioned here, as part of a study of fixpoint operators on trees.)

3 Games for Mu-Arithmetic

The familiar Ehrenfeucht–Fraïssé games for first-order logic are used to distinguish structures; one can also define *model-checking games* or semantic games where the object is to determine whether $\phi(x)$ holds for a formula ϕ and element x of a given structure. For first-order logic, the game may be defined thus: given a formula ϕ and a structure $\mathcal{T}$, a position in the game is a subformula ψ of ϕ and a valuation of the free variables of ψ by elements of $\mathcal{T}$. If ψ is a conjunction, then it is Abelard's turn, and he chooses a conjunct; if a disjunction, then Eloise chooses a disjunct. If $\psi = \exists x.\, \psi'$, then Eloise chooses a value for x and play moves to ψ'; and dually for $\forall x.\, \psi'$. Play terminates at atomic formulae $P(\boldsymbol{x})$; Eloise wins the play if $P(\boldsymbol{x})$ holds of the chosen values for $\boldsymbol{x}$, Abelard otherwise. It is a standard theorem that Eloise wins the game iff ϕ is true. (If ϕ itself has free variables, there is one game for each valuation of them.)

Remark 3. Traditionally, the valuation of variables is not explicitly encoded in the position, but read off from the history. This is a vital distinction in the finite model theory use of games, since we do not wish to assume that we can keep arbitrary amounts of information. The notion of pebble game was invented in order to encode the variable assignment in the position, and then the number of pebbles (variables) can be limited, so that we can ask 'is there a winning strategy using only the bounded history information available?'. However, we are working in arithmetic, so we have all the coding apparatus we want, and may as well carry the assignment with us; it is purely a matter of convenience.

Games have been extended to LFP in the world of finite model theory, by choosing a candidate fixpoint set X when one passes through a fixpoint operator. Uwe Bosse [1] has used such games to obtain expressivity results on fragments of LFP. However, such a game is undesirable in arithmetic, since it has second-order positions. A more useful game for mu-arithmetic is defined by adapting the game for parity automata or modal mu-calculus: instead of finite plays, one now has infinite plays, and the winning condition is given by a parity condition.

Given a formula of mu-arithmetic (or indeed FOL with fixpoints in general), the model-checking game has moves as for first-order logic together with the following rules for the fixpoints: if the position is $\tau \in \boldsymbol{\mu}(x, X).\, \phi$, then play moves to the position ϕ with x valued at the current value of τ. It does not matter who moves, but for definiteness say Eloise moves for μ and Abelard for ν. If the position is $\tau \in X$, where X is bound by $\boldsymbol{\mu}(x, X).\, \phi$, then again play moves to ϕ with x valued at the current value of τ, and we say that we have seen X. It remains to define the winning conditions: if play terminates, the play is won as for first-order logic; if the play is infinite, then Eloise wins iff the outermost

fixpoint variable seen infinitely often is maximal. If we start with a formula in normal form, then this is exactly a parity condition. We have

Theorem 4. *A formula $\phi(\boldsymbol{x})$ of mu-arithmetic holds for some valuation of $\boldsymbol{x}$ exactly if Eloise has a winning strategy for the model-checking game for ϕ with the given initial valuation.*

Proof. A full proof of this theorem is quite long; however it is strategically the same as the corresponding proof for modal mu-calculus and parity games. It is also the essential content of Theorem 5 of [2]. Therefore I shall not give the proof again here. □

4 The Power of the Game Quantifier

Suppose that a winning condition $P(\alpha, x)$ for a Gale–Stewart game has a given descriptive complexity: what is the descriptive complexity of $\mathcal{G}\alpha.P(\alpha, x)$? If the winning conditions are in the analytical hierarchy, then this is a question intimately related to the structure theory of the hierarchy, and a question that depends on hypotheses outside *ZFC*, in particular, the hypothesis of Projective Determinacy (that all projective games are determined). Given *PD*, the answer is quite simple for analytical games: $\mathcal{G}\Sigma^1_n = \Pi^1_{n+1}$ and dually, so $\mathcal{G}$ is a 'hermaphrodite' second-order quantifier.

If the winning conditions are below Δ^1_1, then determinacy is not an issue, and one can expect unequivocal answers. However, the game quantifier turns out to be quite delicate. The first answer was:

Theorem 5 (Kechris–Moschovakis). $\mathcal{G}\Sigma^0_1 = \Pi^1_1$

Later, Robert Solovay (unpublished, cited in [6], *q.v.* also for the previous theorem) characterized Σ^0_2 games, based on Wolfe's proof of the determinacy of Σ^0_2 games.

Theorem 6 (Solovay). $\mathcal{G}\Sigma^0_2 = \Sigma^1_1$*-IND (that is, sets given via an inductive definition over a* Σ^1_1 *predicate).*

A decade or so later, the next step was taken by Thomas John, who studied Σ^0_3 games. Unfortunately, the characterization is complex: it involves capturing the levels of Gödel's L at which winning strategies can be found, and is given in terms of higher-type recursion. This appears to be inevitable: the proof of determinacy for Σ^0_3 proceeds via games in which the positions are themselves games.

There are no published results on Σ^0_4 or beyond.

As was remarked in the introduction, Π^1_1 is also Π^0_1*-IND*; or in terms of the fixpoint hierarchy, Σ^μ_1. Then Σ^1_1*-IND* is just Σ^μ_2. We then naturally ask whether $\Sigma^\mu_n = \mathcal{G}X$ for some natural class X. The conjecture that immediately comes to mind is $\Sigma^\mu_n = \mathcal{G}\Sigma^0_n$; unfortunately, the complexities of higher-type recursion are not so easily banished. In fact, the game semantics of mu-arithmetic shows that

Theorem 7. *For all n, $\Sigma^\mu_n \subseteq \mathcal{G}\nabla^0_2$, where ∇^0_2 denotes the boolean closure of $\Sigma^0_2 \cup \Pi^0_2$.*

Proof. Consider a mu-arithmetic formula in normal form; wlog, consider the case of odd n. The winning condition for the associated game comprises a recursive part dealing with the finite plays, and a parity condition dealing with the infinite plays. The parity condition says that in a play α, the highest X_i seen infinitely often is maximal, i.e. i is even. In other words, the condition is

$$\neg\infty X_n \wedge (\infty X_{n-1} \vee (\neg\infty X_{n-2} \wedge \infty X_{n-3} \vee (\ldots \infty X_2 \vee \neg\infty X_1)\ldots)).$$

Now, the statement 'X_i is seen at position j of the play α' is a recursive predicate of α; and the statement ∞X_i is just $\forall j.\, \exists k > j.$ 'X_i is seen at k', which is Π^0_2. Therefore the entire condition is a boolean combination of Π^0_2 and Σ^0_2 statements, Q.E.D. □

One may equally well use a Rabin condition, although this is less natural.

At this point, it seems 'obvious' that the argument should also run backwards. However, model-checking games are a very restricted format of games, and the statement ∞X_i is apparently a rather restricted form of Π^0_2 statement about a play α; we wish to make a statement about arbitrary ∇^0_2 winning conditions. Thus the obvious statement requires some work to prove. The first step is to choose the appropriate fine hierarchy within ∇^0_2. One may here choose to follow the pattern of Rabin conditions: by using disjunctive normal form, it is trivial that formulae of the shape $\bigvee_i(\Sigma^0_2 \wedge \Pi^0_2)$ give a normal form for ∇^0_2. However, it is easier and more elegant to follow the pattern of parity conditions, and use a hierarchy known as the *difference hierarchy* (over Σ^0_2). Difference hierarchies over open sets have been studied long ago in classical descriptive set theory; more recently Victor Selivanov has, in a series of papers, made a study of an abstract fine hierarchy which subsumes, in a certain sense, difference hierarchies: applications include simpler proofs and refinements of Wagner's hierarchies of ω-regular languages [8]. However, we shall not need any of this more general theory; let us just define the hierarchy we need.

The difference hierarchy over Σ^0_2 is defined thus: $\Sigma^\partial_1 = \Sigma^0_2$; $\Pi^\partial_n = \neg\Sigma^\partial_n$; $\Sigma^\partial_{n+1} = \Sigma^0_2 \wedge \Pi^\partial_n$. To provide a simpler base case, let us also define $\Sigma^\partial_0 = \Sigma^0_1$ (which fits into the induction, since $\Sigma^0_2 \wedge \Pi^0_1 = \Sigma^0_2$). The main result is now

Theorem 8. $\mathcal{G}\Sigma^\partial_n = \Sigma^\mu_{n+1}$ *for* $n \geq 0$.

Proof. First consider the easier direction, that $\Sigma^\mu_{n+1} \subseteq \mathcal{G}\Sigma^\partial_n$. This is not trivial: by inspection, the parity condition of rank n is in Σ^∂_n, but this is not tight enough. However, if we consider more carefully the winning condition for the game of a Σ^μ_1 formula $\tau \in \mu X.\phi$, it says simply 'X is seen finitely often'. Since the only way a play can be infinite is to pass infinitely often through X, this is equivalent to saying that the play is really finite (and therefore terminates on an outright Eloise win). Hence the winning condition is really just $\exists i.$ 'Eloise wins outright at $\alpha(i)$', and since the outright winning conditions are recursive, this is a Σ^0_1 statement.

Hence $\Sigma^\mu_1 \subseteq \mathcal{G}\Sigma^0_1 = \mathcal{G}\Sigma^\partial_0$. Now an induction following the proof of Theorem 4 gives the rest. (Of course, we already know from Theorem 5 that $\Sigma^\mu_1 = \mathcal{G}\Sigma^0_1$; however, the above direct argument of the base case has the advantage of being easy to fit directly into the induction.)

The harder direction is showing that $\mathcal{G}\Sigma^\partial_n \subseteq \Sigma^\mu_{n+1}$. For convenience we shall let Theorem 5 deal with the base case; it is an easy exercise to write down the direct proof using a simplified version of the strategy here. The inductive step is a generalization of Solovay's result, using a generalization of Wolfe's determinacy proof. We shall follow, more or less, the presentation of Wolfe's proof by Moschovakis [6], extending as necessary.

The approach is to define inductively 'easy winning positions', and show that all winning positions are easy. We then inspect the inductive definition, and see that it has the required form.

Suppose we have a Σ^∂_n winning condition $P(\alpha, \boldsymbol{x})$; for notational convenience we omit the parameters $\boldsymbol{x}$. Then it has the form

$$(\exists i.\, Q(i,\alpha)) \wedge R(\alpha)$$

where Q is Π^0_1 and R is Π^∂_{n-1}. In Solovay's result, the winning condition is $\Sigma^0_2 = \Sigma^\partial_1$, and so there is no R term; we have to show that the argument still goes through with this additional term, so allowing us to use the proof in an induction on n.

We start with a trivial but critical observation:

$$(\exists i.\, Q(i,\alpha)) \wedge R(\alpha) \Leftrightarrow \exists i.\, (Q(i,\alpha) \wedge R(\alpha)).$$

The second observation is that (by lemma 1) since, for a given i, $Q(i,\alpha)$ is a Π^0_1 predicate of α, there is a Π^0_1 tree $T_i \subseteq \omega^*$ such that $Q(i,\alpha) \Leftrightarrow \alpha \in [T_i]$.

We shall build the set of winning positions by a transfinite induction; to explain the technique let us first consider the base case on its own. We can define a set of *really* easy winning positions: let

$$W^0 = \{\, u \mid \exists i.\, \text{'Eloise wins the game } H^0_i[u] = (Q(i,\alpha) \wedge R(\alpha))[u]\text{'} \,\}$$

Strictly, if u is an odd length sequence, we mean the cogame $H^0_i[u]$; we will assume henceforth that 'game' means 'game' for even length u and 'cogame' for odd length u. Now, it is clear that if $u \in W^0$, then Eloise wins the game $P[u]$.

To extend this base case into an inductive step, we first reformulate this definition using the second observation:

$$W^0 = \{\, u \mid \exists i.\, \text{'Eloise wins the game } H^0_i[u] = (R(\alpha) \wedge \forall k.\, \alpha(<k) \in T_i)[u]\text{'} \,\}$$

So the 'really easy' winning positions can be thought of as the places where Eloise knows how to win R while also staying within T_i. Now the inductive step is to look at places where Eloise knows how to win R while staying within the winning positions for easier games. That is, if W^ξ is defined for $\xi < \zeta$, let $W^{<\zeta} = \bigcup_{\xi<\zeta} W^\xi$, and define the game

$$H^\zeta_i(\alpha) = R(\alpha) \wedge \forall k.\, \alpha(<k) \in W^{<\zeta} \cup T_i.$$

Then we define

$$W^\zeta = \{\, u \mid \exists i.\text{`Eloise wins the game } H_i^\zeta[u]\text{'} \,\}.$$

We show by induction that if $u \in W^\zeta$, then u is a winning position in the original game P. So, let $u \in W^\zeta$. Then for some i, Eloise wins $H_i^\zeta[u]$; let Eloise play according to her winning strategy to produce a play α. Then by definition, $R(\alpha) \wedge \forall k.\, \alpha(<k) \in W^{<\zeta} \cup T_i$. If play ever reached a position $v = \alpha(<k) \in W^{<\zeta}$, then by induction Eloise could have won $P[v]$, so by switching to her winning strategy there, instead of continuing with α, she can win $P[u]$. If not, then $R(\alpha) \wedge \forall k.\, \alpha(<k) \in T_i$; but then $\alpha \in [T_i]$, so $Q(i, \alpha) \wedge R(\alpha)$, so Eloise wins the play α in the game P.

Now, W^ζ is an increasing chain, and so (by cardinality) closes at some $W = W^\kappa = W^{<\kappa}$. We now show that if $u \notin W^\kappa$, then Abelard wins $P[u]$. So, let $u = a_0 \ldots a_j \notin W^\kappa$. By definition, for all i, Abelard wins $H_i^\kappa[u]$. (Note: see Remark 9.) Let Abelard continue to play according to his winning strategy for $H_0^\kappa[u]$, generating a play $\alpha = u a_{j+1} \ldots$. First suppose that $\forall k.\, \alpha(<k) \in W^{<\kappa} \cup T_o$; then we must have $\alpha \notin R$, so $\alpha \notin P$, and so Abelard has won P. On the other hand, suppose at some j_0 we have $u_0 = \alpha(<j_0) \notin W^{<\kappa} \cup T_0$. Then firstly $u_0 \notin T_0$, and since T_0 is a tree, any extension of u_0 is also $\notin T_0$. Secondly, $W^{<\kappa} = W^\kappa$, so $u_0 \notin W^\kappa$, so Abelard wins all $H_i^\kappa[u_0]$. So now let Abelard switch to his strategy for H_1^κ. Now repeat the argument: either Abelard plays and wins with R, or there is a $u_1 \notin W^{<\kappa} \cup T_1$. If the process of finding $u_0, u_1, \ldots$ continues for ever, then the final play α is not an infinite branch of any T_i, and so $\neg\exists i.\, Q(i, \alpha)$ and again Abelard has won the play.

We have now shown that $u \in W$ iff Eloise wins the game $P[u]$, in other words that $W = \mathcal{G}\alpha.P(u \cdot \alpha)$. All that remains is to recast the inductive definition in terms of $\mathcal{G}$ and mu-arithmetic:

$$W = \mu(w, W).\exists i.\, \mathcal{G}\alpha.(R(w \cdot \alpha) \wedge \forall k.\, ((w \cdot \alpha)(<k) \in W \vee (w \cdot \alpha)(<k) \in T_i))$$

Now, T_i is a Π_1^0 set, and therefore $\forall k. \ldots$ is also Π_1^0; R is Π_{n-1}^∂, and so the body of the game quantified expression is also Π_{n-1}^∂. Therefore by the induction hypothesis and duality, $\mathcal{G}\alpha. \ldots$ is equal to some Π_n^μ expression ϕ, and so W is indeed Σ_{n+1}^μ; Q.E.D. □

Remark 9. At the point referring to this Remark, we seem to be assuming the determinacy of H_i^κ. At first sight, this seems odd, since the set W^κ occurring in the definition is rather complex; however, the determinacy theorems involve the boldface classes, not the lightface, and any subset of the integers is Δ_1^0, so the determinacy theorems apply. In fact, as I have mentioned, this proof is mostly Wolfe's, and was devised to show determinacy. This works because at the end we have constructed a set W such that Eloise wins P iff $\langle\rangle \in W$, and Abelard wins P iff $\langle\rangle \notin W$. To show determinacy, we use Σ_2^0 rather than Σ_2^0; the only difference this makes is that Q is Π_1^0 instead of Π_1^0, so the trees T_i are not necessarily Π_1^0. The argument goes through to produce the set W; thus we have an inductive proof of the determinacy of Σ_n^∂ games, and then we look at the lightface version in order to obtain the complexity results we really want.

It is worth mentioning, as already noted in [4], that the use of fixpoint notation makes Wolfe's proof itself rather more transparent. The determinacy of (boldface) $\mathbf{\nabla}^0_2$ games was studied by Büchi [3] in the context of monadic second-order logic; again the use of fixpoint notation allows a transparent presentation, since the formulation of Theorem 8 works also in the boldface case.

As was mentioned above, we could as well use the Rabin style hierarchy as the difference hierarchy; indeed, Σ^∂_{2n} is equal to the Rabin class $\bigvee_{1\leq i\leq n}(\Sigma^0_2 \wedge \Pi^0_2)$, and the odd levels of the difference hierarchy correspond to Rabin conditions with one disjunct being simply Σ^0_2.

Since the above proof is also defining a winning strategy, it follows from Lubarsky's characterization that

Corollary 10. *A Σ^∂_n game has a winning strategy in the $\omega^{+(n+1)}$'th level of Gödel's L, where ω^{+n} is the first n-reflecting admissible [5] after ω.*

A result that is already known, but which is much more easily seen from this approach, is

Corollary 11. *The fixpoint definable sets of integers are strictly contained in Δ^1_2.*

Proof. Because the game quantifier is self-dual for reasonable point classes (lemma 2) and because if $U(i,\alpha,x) \subseteq \omega\times{}^\omega\omega\times \mathrm{X}$ is universal for Γ on ${}^\omega\omega\times\mathrm{X}$ then $\mathcal{G}\alpha.U(i,\alpha,x) \subseteq \omega\times\mathrm{X}$ is universal for $\mathcal{G}\Gamma$ on X, the game quantifier preserves the strictness of reasonable hierarchies. In particular, it preserves the arithmetic and hyperarithmetic hierarchies. By the main theorem, the fixpoint definable sets are contained in $\mathcal{G}\nabla^0_2$; but Δ^1_2 contains $\mathcal{G}\Delta^1_1$, a much larger set.

The fact that fixpoint definable sets are contained in Δ^1_2 follows from the classical closure of Δ^1_2 under inductive definitions; the strictness of the containment is established by Lubarsky's analysis, as the ordinals ω^{+n} are all less (and in some sense much less) than the first non-Δ^1_2 ordinal. However, the game characterization is technically simpler, and gives a more transparent meaning to 'much less': ∇^0_2 is 'much smaller' than Δ^1_1 in a well understood sense.

5 Final Remarks

The characterization we have established here is interesting in both directions. The fact that $\mathcal{G}\Sigma^\partial_n$ is characterized by a natural and useful pointclass extends a little the point at which games become inherently difficult. Perhaps the other direction is more interesting: fixpoint alternation is notoriously incomprehensible, so characterizing it in terms of a simple hierarchy of games is helpful—and arguably more useful than the admissible-recursion-theoretic characterization. It also reinforces a slightly different view on the traditional world of automata with Rabin and parity conditions: 'infinitely often' is a fundamental concept in temporal logics, but really it is a fundamental concept because it is Π^0_2. Indeed, within the framework of recursion theory, any Π^0_2 statement about α is of the

form 'infinitely often something happens at $\alpha(i)$', where in general 'something' includes statements about the previous and future elements of α.

There are some obvious directions in which to extend this investigation. As I have mentioned, Selivanov has made a detailed study of difference-like hierarchies, and I hope that further results may emerge from applying his work. One question of particular interest is that of transfinite extensions. The fixpoint hierarchies can be extended into the transfinite in the usual way, and Selivanov's hierarchies are also transfinite. One can then ask whether our characterization here extends at all. A possibly interesting issue here is that there is something of a mis-match between the hierarchies: the first natural stopping point for a transfinite extension of the difference hierarchy is ω_1^{CK}, whereas the transfinite fixpoint hierarchy has no natural stopping point before an otherwise unknown and extremely large ordinal. I conjecture that an extension may be possible up to ω_1^{CK}; beyond there it is not clear that the question is meaningful.

Another natural question is whether this descriptive set theoretic approach can be used directly to establish results such as the non-collapse of the modal mu-calculus alternation hierarchy, avoiding the passage through arithmetic that was used in [2]. Here again Selivanov's work is relevant: he has shown how an approach from descriptive set theory provides elegant proofs of results on ω-regular expressions.

6 Acknowledgements

I am supported by an Advanced Fellowship (AF/97/0322) from the United Kingdom Engineering and Physical Sciences Research Council; also BRICS, the Danish National Research Foundation Centre for Basic Research In Computer Science, is supporting my visit to Aarhus.

References

1. U. Bosse, An 'Ehrenfeucht-Fraïssé game' for fixpoint logic and stratified fixpoint logic. *Computer science logic* , LNCS **702** (San Miniato, 1992) 100–114.
2. J. C. Bradfield, The modal mu-calculus alternation hierarchy is strict, *Theor. Comput. Sci.* **195** 133–153 (1997).
3. J. R. Büchi, Using determinancy of games to eliminate quantifers, *Proc. FCT '77*, LNCS **56** 367–378 (1977).
4. E. A. Emerson and C. S. Jutla, Tree automata, mu-calculus and determinacy, in: *Proc. FOCS 91.* (1991)
5. R. S. Lubarsky, μ-definable sets of integers, *J. Symbolic Logic* **58** (1993) 291–313.
6. Y. N. Moschovakis, *Descriptive Set Theory* (North-Holland, Amsterdam, 1980).
7. D. Niwiński, Fixed point characterization of infinite behavior of finite state systems. *Theoret. Comput. Sci.* **189** (1997) 1–69.
8. V. Selivanov, Fine hierarchy of regular ω-languages, *Theoret. Comput. Sci.* **191** (1998) 37–59.

Lower Bounds for Space in Resolution

Jacobo Torán*

Abt. Theoretische Informatik
Universität Ulm
Oberer Eselsberg
D-89069 Ulm
toran@informatik.uni-ulm.de

Abstract. Resolution space measures the maximum number of clauses that need to be simultaneously active in a resolution refutation. This complexity measure was defined by Kleine Büning and Lettmann in [8] and slightly modified recently [6] to make it suitable for comparisons with other measures. Since its definition, only trivial lower bound for the resolution space, measured in terms of the number of initial clauses were known. In this paper we prove optimal lower bounds for the space needed in the resolution refutation of two important families of formulas. We show that Tseitin formulas associated to a certain kind of expander graphs of n nodes need resolution space $n - c$ for some constant c. Measured on the number of clauses, this result is best possible since the mentioned formulas have $O(n)$ clauses, and the number of clauses is an upper bound for the resolution space. We also show that the formulas expressing the general Pigeonhole Principle with n holes and more than n pigeons, need space $n + 1$ independently of the number of pigeons. Since a matching space upper bound of $n + 1$ for these formulas exist, the obtained bound is exact. These results point to a possible connection between resolution space and resolution width, another measure for the complexity of resolution refutations.

Keywords: Resolution, complexity measures, space, lower bounds, pebbling game.

1 Introduction

Resolution is perhaps the most studied propositional refutation system. This is on one hand because its importance in many automatic theorem proving procedures, and on the other hand because its simplicity. Resolution acts on clauses and contains only one inference rule. If $A \vee x$ and $B \vee \bar{x}$ are clauses, then the clause $A \vee B$ (the resolvent) may be inferred by the resolution rule resolving the variable x. A resolution refutation of a non-satisfiable conjunctive normal form (CNF) formula φ is a sequence of clauses $C_1 \dots C_s$, where each C_i is either a

* Partially supported by the DFG

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 362–373, 1999.

clause of φ or is inferred from earlier clauses in the refutation by the resolution rule, and C_s is the empty clause, $\Box$.

Resolution refutations can be viewed as directed acyclic graphs, where the clauses are the nodes, and if two clauses are resolved, there is a directed edge from each of these two clauses to to their resolvent. Resolution refutations can be restricted to be *tree-like*, considering only trees as possible underlying graphs. In this case, each clause in the refutation is used in an inference of a new clause at most once. The same clause may appear more than once in the tree-like refutation.

Because of the importance of resolution, several measures for the complexity of resolution refutations have been introduced. The most natural one is the *size*, that is, the number of clauses that are needed in the resolution refutation of a formula (in the case of tree-like proofs, each appearance of a clause in counted). Using the family of formulas expressing the Pigeonhole Principle, Haken [7] proved for the first time an exponential lower bound on the size of a resolution refutation. This proof was subsequently simplified and extended to other families of formulas [14,5,2].

In spite of these results, there are still many questions about the complexity of resolution that remain unsolved. In an attempt to better understand this refutation system, other complexity measures like *width* and *space* have been introduced.

Recently Ben-Sasson and Wigderson [3] unified all the existing exponential lower bounds for resolution size using the concept of *width*. The width of a resolution refutation is the maximal number of literals in any clause of the refutation. The authors relate in [3] width and size showing that lower bounds for resolution width imply lower bounds for resolution size.

Another measure for the complexity of a resolution refutation is the amount of *space* it needs. This concept was initially defined by Kleine Büning and Lettman in [8], and slightly modified in [6] in order to make it more natural and suitable for comparisons with other complexity measures[1].

Definition 1. *Let $k \in I\!N$, we say that an unsatisfiable CNF formula φ has resolution refutation bounded by space k if there is a series of CNF formulas $\varphi_1, \ldots, \varphi_n$, such that $\varphi_1 \subseteq \varphi$, $\Box \in \varphi_n$, in any φ_i there are at most k clauses, and for each $i < nr$, φ_{i+1} is obtained from φ_i by deleting (if wished) some of its clauses, adding the resolvent of two clauses of φ_i, and adding (if wished) some of the clauses of φ (initial clauses).*

The space needed for the resolution of an unsatisfiable formula φ, is the minimum k for which the formula has a refutation bounded by space k.

Intuitively, this expresses the idea of keeping a set of *active* clauses in the refutation, and producing from this set a new one until the empty clause is included in the set. The new set is produced by copying clauses from the previous

[1] The original definition [8] differs from the one given in [6] and here in the fact that when an initial clause has been deleted from the list of active clauses, it cannot be included again.

set or from the initial set of clauses, and resolving one pair of clauses. The space used is the maximum number of clauses that are simultaneously active in the refutation.

An upper bound on the space needed for the resolution of a formula φ with n variables is $n+1$, [6], and there are formulas for which this is also a lower bound. On the other hand, some unsatisfiable CNF formulas (like for example those with at most 2 literals in each clause) can be resolved using only constant space [6]. However, the question of the existence of nontrivial space lower bounds measured on the number of initial clauses of the formula was open. We address this question here obtaining optimal space lower bounds for the two important families of Tseitin formulas and formulas expressing the Pigeonhole Principle. Lower bounds for the space needed in resolution and Polynomial Calculus, have also been recently considered in [1].

Very similar results also hold for these families of formulas if the width instead of the space of a resolution refutation is used [3]. This is surprising since both measures seem unrelated, and suggest that there might be a relationship between the concepts of width and space.

We show in Section 2 space lower bounds for the refutation of Tseitin formulas. This family of formulas was first defined by Tseitin [13], and express the principle that the sum of the degrees of the vertices in a graph must be even. Tseitin proved in [13] super-polynomial lower bounds on the size of regular resolution refutations for them. Later Urquhart [14] improved these bounds to exponential lower bounds for general resolution. We prove that the space needed for the resolution of a Tseitin formula with associated graph G is at least $ex(G) - \lfloor \frac{d}{2} \rfloor + 1$, where $ex(G)$ is the expansion of G and d its maximum degree. For Tseitin formulas corresponding to expander graphs with n nodes, this means that the space needed is at least $n - c$ for some constant c. These formulas have $O(n)$ variables and clauses, and because of the general space upper bound mentioned above, the space needed is $\Theta(n)$, and this linear lower bound on the number of initial clauses is optimal up to a constant factor[2].

The family of formulas for the general Pigeonhole Principle PHP_n^m express the fact that it is not possible to fit m pigeons in n pigeonholes (for $m > n$). As mentioned above, for the case $m = n+1$, this was the first example of a family of formulas with an exponential resolution size lower bound [7]. We show that the negation of PHP formulas need refutation space $n+1$, independently of the number of pigeons [3]. In this case we have an exact bound since Messner [10] has proven that $n+1$ is also an upper bound for the space needed for the refutation of PHP formulas with n pigeonholes.

This lower bound result is also interesting due to the fact that the complexity of resolution refutations of the general Pigeon Hole Principle is not known. For example, only trivial lower bounds on the size are known when the number of

[2] A linear space lower bound in the number of initial clauses for Tseitin formulas have been independently proven in [1]

[3] A $\Omega(n)$ lower bound for the resolution space of PHP_n^m have been obtainded independently in [1]

pigeons m is greater than n^2. Buss and Pitassi [4] have shown that for the case of tree-like resolution, for any $m > n$, $\neg\text{PHP}_n^m$ needs tree-like resolution refutation of size at least 2^n. This result can also be proven using a lower bound on the width of refutations for $\neg\text{PHP}_n^m$ from [3]. Due to the fact that tree-like resolution refutations of size S require at most space $\lceil \log S \rceil + 1$, [6], the above mentioned space lower bound for $\neg\text{PHP}_n^m$ also provides the lower bound 2^n on the size of tree-like resolution refutations for these formulas.

We show then that for the case of tree-like resolution, the space needed in a refutation of a formula is at least as large as the refutation width minus the initial with of the formula. Again here we find a connection between the concepts of space and width.

The space lower bound results are proven using an alternative characterization of the space measure based on a pebble game on graphs. Resolution refutations can be represented as directed acyclic graphs of in-degree two, in which the nodes are the clauses used in the refutation, and a vertex (clause) has outgoing edges to the resolvents obtained using this clause. In case that in the refutation no derived clauses are reused, that is, when all the nodes in the refutation graph (except maybe the sources) have out-degree one, the proof is called tree-like.

The space required for the resolution refutation of a CNF formula φ (as expressed in Definition 1) corresponds to the minimum number of pebbles needed in the following game played on the graph of a refutation of φ. Observe that in a resolution refutation graph the sources are the initial clauses, and the unique sink is the empty clause.

Definition 2. *Given a connected directed acyclic graph with one sink the aim of the pebble game is to put a pebble on the sink of the graph (the only node with no outgoing edges) following this set of rules:*

1) *A pebble can be placed in any initial node, that is, a node with no predecessors.*
2) *Any pebble can be removed from any node at any time.*
3) *A node can be pebbled provided all its parent nodes are pebbled. For doing so, one can place a new pebble on the node, or one can shift a pebble from a parent node.*

Because of the equivalence of both definitions [6], we will indistinctly talk about the space needed in the refutation of a formula, or about the number of pebbles needed on a game played on its refutation graphs.

The space lower bounds are obtained reasoning on the pebble game. The idea is the following: a critical stage in any pebbling strategy of the refutation graphs is defined, and then it is proved that this stage must exist, and that it must contain a large number of pebbles. In a critical stage it is required that there is a partial assignment of the variables that on one hand includes variables from all the pebbled clauses in that stage, and on the other hand does not satisfy a combinatorial property related to the input formula.

2 Lower Bounds on Tseitin Formulas

In this section we study the space used in resolution refutations of some formulas related to graphs. This formulas were defined originally by Tseitin [13], and have also been used in order to prove lower bounds on the size of resolution refutations in [14] and [12].

Let $G = (V, E)$ be a connected undirected graph with n vertices, and let $m : V \to \{0, 1\}$ be a marking of the vertices of G satisfying the property

$$\sum_{x \in V} m(x) = 1 (\text{mod } 2).$$

For such a graph we can define an unsatisfiable formula in conjunctive normal form $F(G, m)$ in the following way: The formula has E as set of variables, and is a conjunction of the formulas F_x for $x \in V$, where

$$F_x = \begin{cases} e_1(x) \oplus \ldots \oplus e_d(x) & \text{if } m(x) = 1 \\ \overline{e_1(x) \oplus \ldots \oplus e_d(x)} & \text{if } m(x) = 0 \end{cases}$$

Here $e_1(x) \ldots e_d(x)$ are the edges (variables) incident with vertex x. If d is the maximum degree of a node in G, $F(G, m)$ contains at most $n2^{d-1}$ many clauses, each one with at most d many literals. The number of variables of the formulas is bounded by $\frac{dn}{2}$.

$F(G, m)$ captures the combinatorial principle that for all graphs the sum of the degrees of the vertices is even. When the marking m is odd, $F(G, m)$ is unsatisfiable. Suppose on the contrary that there were a satisfying assignment $\varphi : E \to \{0, 1\}$. For every vertex x, the number of edges of x that have been assigned value 1 by φ has the same parity as $m(x)$, and therefore

$$\sum_{x \in V} \sum_{(x,y) \in E} \varphi((x, y)) \equiv \sum_{x \in V} m(x) \equiv 1 (\text{ mod } 2)$$

but in the left hand sum in the equality, every edge is counted twice and therefore this sum must be even, which is a contradiction.

Fact 1 *For an odd marking m, for every $x \in V$ there exists an assignment φ with $\varphi(F_x) = 0$, and $\varphi(F_y) = 1$ for all $y \neq x$. If the marking is even, then $F(G, m)$ is satisfiable.*

Consider a a partial truth assignment t of some of the variables. We refer to the following process as applying t to (G, m): Setting a variable (x, y) in t to 0 corresponds to deleting the edge (x, y) in the graph, and setting it to 1 corresponds to deleting the edge from the graph and toggling the value of $m(x)$ and $m(y)$ in G. Observe that the formula $F(G', m')$ for the graph and marking (G', m') resulting after applying t to (G, m) is still unsatisfiable.

In order to prove the lower bound we will consider the last stage in any pebbling strategy in which two properties are satisfied. On the one hand, the set of pebbled clauses must be simultaneously satisfiable. The other property needed is based on non-splitting assignments, a concept that we define next.

Definition 3. *We say that a partial truth assignment t of some of the variables in $F(G,m)$ is non-splitting for (G,m), if applying it to (G,m) produces a pair (G',m') so that G' has a connected component of size $> \frac{2}{3}n$ with an odd number of 1's in its marking, and an even number of 1's in the markings of all other connected components.*

Definition 4. *Let $G=(V,E)$ be an undirected graph with with $|V|=n$. The expansion of G, $ex(G)$ is defined as:*

$$ex(G) = \max k: \ \forall S \subseteq V, |S| \in [\frac{n}{3}, \frac{2n}{3}], \ |\{(x,y) \in E: \ x \in S, y \notin S\}| \geq k.$$

Intuitively the expansion of a graph is the minimum size of a cut produced when the vertices are partitioned into two subsets that do not differ too much in size. As shown in the next theorem, the expansion of a graph is a lower bound on the space required in the resolution of its associated Tseitin formula.

Theorem 2. *Let $G=(V,E)$ be an undirected and connected graph with $|V|=n$ and maximum degree d, and let m be an odd marking of G. Any resolution refutation of $F(G,m)$ requires space at least $ex(G) - \lfloor \frac{d}{2} \rfloor + 1$.*

Proof. Let Π be a resolution refutation of the formula, and consider the last stage s in a pebbling strategy of the graph of Π in which there is a partial assignment t fulfilling the following two properties:

i) t simultaneously satisfies all the pebbled clauses at stage s,
ii) t is *non-splitting* for (G,m).

This stage in the pebbling must exist: Before the initial step, no clause has a pebble. Since G is connected, the empty truth assignment is trivially a non-splitting partial assignment satisfying the set of pebbled clauses. At the end, the set of pebbled clauses contains the empty clause which cannot be satisfied by any assignment. Stage s must exist in between.

The clause pebbled in stage $s+1$ must be an initial one. The only other clause that could be pebbled at stage $s+1$ would be a clause C_3 whose parents C_1 and C_2 already have a pebble, but any partial assignment satisfying C_1 and C_2 also satisfies C_3, and the non-splitting partial assignment from stage s would also work for stage $s+1$. For some vertex x in G, this last initial pebbled clause corresponds to the formula F_x.

Let t be a partial assignment satisfying properties *i*) and *ii*) at stage s. There is an extension of t that satisfies F'_x, the formula for x after applying t. To see this, observe that after applying t to (G,m), the graph has a connected component of size at least $\frac{2n}{3}$ with an odd marking, and the rest of the components have even markings. By Fact 1, for every vertex x, the formula F'_x can therefore be satisfied by an extension of t. Moreover, the initial clause C pebbled at stage $s+1$ corresponds to a vertex x in the big connected component with odd marking since otherwise there would be also non-splitting partial assignments satisfying all the pebbled clauses at stage $s+1$.

Let t be a non-splitting partial truth assignment of minimal size satisfying the clauses at stage s, and (G', m') the graph and marking resulting after applying t. It suffices to extend t giving some value to one or more of the variables in the last pebbled clause to obtain an assignment t' satisfying all the clauses pebbled at stage $s+1$. However, t' is a splitting assignment and applying it to (G, m) does not produce a connected component larger than $\frac{2}{3}n$ with odd marking. We will show that there is always a way to extend t to t' by assigning some new variables in the last pebbled clause C, in such a way that t' satisfies all the pebbled clauses and produces a subgraph disconnected from the rest and with a number of nodes in the interval $[\frac{n}{3}, \frac{2n}{3}]$.

Let C be the initial clause pebbled at stage $s+1$, corresponding to a node x and let d' be the degree of x in G' ($d' \leq d$). $F'(x)$ is the formula $e_1(x) \oplus \ldots \oplus e_{d'}(x) = m'(x)$. We have shown that this formula is satisfiable. d' is at least 1, since otherwise t would also satisfy F'_x.

x is connected in G' to d' components $A_1, \ldots, A'_d$, and there is no edge between any two of such components A_i, A_j. Otherwise, satisfying the clause C by satisfying the literal corresponding to the edge connecting x and A_i, would provide a non-splitting extension of t.

We consider different cases depending on the size of the A components.

Case 1: Some component A_i has size within the interval. Deleting the edge connecting x and A_i, this component is isolated from the rest of the graph.

Case 2: The size of all the A_i components lie outside the interval. This implies that they all have size smaller than $\frac{n}{3}$, since otherwise, by Fact 1, there would be an extension of t that satisfies C, and disconnects all the components form node x producing an odd marking in the component of size greater that $\frac{2n}{3}$, and an even marking in all the other ones. This would provide a non-splitting assignment satisfying all the pebbled clauses at stage $s+1$. The size of all the components A_i is therefore smaller than $\frac{n}{3}$ and the sum of all their sizes is greater than $\frac{2n}{3}$. There is a set of at most $\lfloor \frac{d'}{2} \rfloor$ components such that the sum of their sizes lie within the interval. This set of components can be isolated from the rest of the graph just by deleting the edges connecting them to x.

In both cases, by deleting at most $\lfloor \frac{d}{2} \rfloor$ edges from G' we have isolated a set of nodes S of size within $[\frac{n}{3}, \frac{2n}{3}]$ from the rest of the graph. There are at least $ex(G)$ edges $\{y, z\}$ in G with $y \in S$ and $z \notin S$. All these edges, except at most $\lfloor \frac{d}{2} \rfloor$ of them have been removed by the partial assignment t. Since t was chosen to be an assignment of minimal size satisfying all the pebbled clauses at stage s, there are at least $ex(G) - \lfloor \frac{d}{2} \rfloor$ pebbled clauses at this stage and $ex(G) - \lfloor \frac{d}{2} \rfloor + 1$ pebbled clauses at stage $s+1$.

■

There exist expander graphs G with n nodes constant degree d and with $ex(G) > n$ [9]. In [11] it is shown that the degree for such expander graphs can be reduced to $d = 8$. For an odd marking of such a graph the formula $F(G, m)$ has at most $\frac{dn}{2}$ variables and $n2^{d-1}$ clauses. By the above result, the space needed in a resolution refutation of $F(G, m)$ is at least $n - 3$ as stated in the next corollary:

Corollary 1. *For the constant $d = 8$ there is a family of unsatisfiable formulas $F_1, F_2, \ldots$ (corresponding to expander graphs) such that for every n F_n has at most $256n$ clauses and $4n$ variables, and any resolution refutation of F_n requires at least space $n - 3$.*

The number of variables of a formula is an upper bound for its resolution space [6]. For the family of formulas mentioned in the corollary, the space needed is therefore $\Theta(n)$. Observe that this bound is linear, measured in terms of the number of clauses of the formula.

An interesting fact is that Theorem 2 (even with the lower bound $ex(G)$ instead of $ex(G) - \lfloor \frac{d}{2} \rfloor + 1$) also holds if the width of the refutation instead of the space is considered [3].

3 The Pigeonhole Principle

Let $m > n$. The tautology PHP_n^m expresses the Pigeonhole Principle that there is no one-one mapping from a domain of size m (the set of pigeons) into a range of size n (the set of holes). We study the space needed in a resolution refutation of the contradiction $\neg\mathrm{PHP}_n^m$. This contradiction can be written as a CNF formula in the following way: The variables of the formula are $x_{i,j},\ 1 \leq i \leq m, 1 \leq j \leq n$. $x_{i,j}$ has the intuitive meaning that pigeon i is mapped to hole j. There are mn variables. The clauses of the formula are:

(1) $x_{i,1} \vee x_{i,2} \vee \ldots \vee x_{i,n}$ for $1 \leq i \leq m$, and
(2) $\overline{x}_{i,k} \vee \overline{x}_{j,k}$ for $1 \leq i, j \leq m,\ 1 \leq k \leq n,\ i \neq j$.

Clauses of type (1) express the fact that every pigeon is mapped to some hole, while the clauses of type (2) indicate that at most one pigeon can be mapped to any hole.

The number of clauses in $\neg\mathrm{PHP}_n^m$ is $m + \binom{m}{2} n < m^2 n$.

Theorem 3. *For any $m > n$, the space needed in a resolution refutation of $\neg PHP_n^m$ is at least $n + 1$.*

Proof. Let Π be a resolution refutation of $\neg\mathrm{PHP}_n^m$ and consider the last stage s in a pebbling strategy of the graph of Π in which there is a partial assignment t fulfilling the following two properties:

i) t simultaneously satisfies all the pebbled clauses at stage s, and
ii) t does not assign value false to any of the initial clauses.

At stage $s = 0$ in the pebbling process, such a partial assignment t exists since there are no pebbled clauses. Also, at the end of the pebbling, the empty clause has a pebble on it and therefore there is no t fulfilling property i). Because of this, the stage s defined above must exist.

The pebble from stage $s + 1$ is placed in an initial clause. Otherwise the two parents of the pebbled clause at stage $s + 1$ contain a pebble in stage s and any partial assignment satisfying the pebbled clauses at stage s also satisfies the clauses at stage $s + 1$.

Let t be a partial assignment simultaneously satisfying all the pebbled clauses at stage s. t can be extended to a partial assignment t' that satisfies the last pebbled clause C. We have seen that C must be an initial clause. If no extension of t can satisfy clause C is because t assigns value false to all the literals in C, but this is a contradiction since C is an initial clause, and by condition ii) t cannot give value false to any initial clause.

Let t be a partial assignment of minimal size satisfying all the pebbled clauses at stage s and not giving value false to any initial clause, and let t' be any extension of t satisfying the clause C pebbled at stage $s+1$. By hypothesis, t' falsifies some initial clause.

If C is of type (1) for some pigeon i, C can be satisfied by giving value true to some variable $x_{i,k}$ that has not been assigned by t. This makes some initial clause $C_{i,k}$ false, and therefore $C_{i,k}$ must be of type (2), $C_{i,k} = \overline{x_{i,k}} \vee \overline{x_{j,k}}$ for some j. This implies that for any hole k, t assigns variable $x_{i,k}$ value false, or variable $x_{j,k}$ value true (for some $j \neq i$), and therefore t assigns at least as many variables as holes. Since t was a partial assignment of minimal size satisfying all the pebbled clauses at stage s, in this stage at least n clauses were pebbled, and in $s+1$ at least $n+1$.

If C is of type (2), $C = \overline{x_{i,k}} \vee \overline{x_{j,k}}$, assigning value true to any literal in C that has not been assigned by t, falsifies some initial clause of type (1). If t has not assigned value to any of the variables in C, this means that the number of variables assigned by t is at least $2n-2$. Otherwise t has assigned at least n variables. For $n \geq 2$, this implies that the number of variables assigned by t is at least n, which means that the number of pebbled clauses at stage $s-1$ is at least n, and at stage s, $n+1$. ■

Jochen Messner [10] has proved that $n+1$ pebbles suffice in a resolution refutation of the Pigeonhole Principle with n holes and $m > n$ pigeons. This means the the above space lower bound is exact.

Although only trivial lower bounds for the size of a resolution refutation of the general Pigeonhole Principle $\neg\text{PHP}_n^m$ are known for the case $m > n^2$, the situation is better when restricted to tree-like resolution. In [4] it is shown that for any $m > n$ $\neg\text{PHP}_n^m$ requires tree-like resolution refutations of size 2^n. Using the following result from [6] we can derive this bound as a corollary of the above space lower bound.

Theorem 4. *[6] Let φ be an unsatisfiable CNF formula with a tree-like resolution of size S, then φ has a resolution refutation of space $\lceil \log S \rceil + 1$.*

The same bound for tree-like refutations of $\neg\text{PHP}_n^m$ has also obtained in [3] using a lower bound on the width of the refutations of $\neg\text{PHP}_n^m$.

4 Tree-Like Refutations, Space and Width

The relationship between the two complexity measures of space and width is not clear. Recall that width of a refutation denotes the maximum number of literals of a clause appearing in the refutation. Formally:

Definition 5. *[3] The width of a clause C, $w(C)$, is defined as the number of literals in C. The width of a set of clauses in the maximal width of a clause in the set. The width of deriving a clause C from the formula φ, denoted $w(\varphi \vdash C)$ is defined by $min_{\Pi}\{w(\Pi)\}$ where the minimum is taken over all resolution derivations Π of C from φ.*

In the case of tree-like resolution we can show a connection between the concepts of size and width. For an unsatisfiable formula φ, define *tree-space*(φ) to be the minimum number of pebbles needed in a tree-like resolution refutation of φ. For any unsatisfiable formula φ, the difference between the width in a refutation of φ minus the initial width of the formula, is bounded by the space in any tree-like refutation of the formula. The proof of this fact relies on the following lemma from Ben-Sasson and Wigderson:

Lemma 1. *[3] Let φ a CNF unsatisfiable formula, and for a literal a, let φ_0 and φ_1 be the formulas resulting from assigning a the truth values 0 and 1 respectively. If for some value k, $w(\varphi_0 \vdash \square) \leq k-1$ and $w(\varphi_1 \vdash \square) \leq k$ then $w(\varphi \vdash \square) \leq max\{k, w(\varphi)\}$*

Theorem 5. *Tree-space*$(\varphi) - 1 \geq w(\varphi \vdash \square) - w(\varphi)$.

Proof. Let φ be an unsatisfiable CNF formula, and s the minimum number of pebbles needed in any tree-like refutation of φ, Π. We prove by induction on the depth of Π, d, that $w(\varphi \vdash \square) \leq w(\varphi) + s - 1$. For $d = 0$, we have that $\square$ is an initial clause, and the results holds trivially. For $d > 0$, let Π be a tree-like refutation of φ of depth d and let x be the last variable being resolved. Let T_0 and T_1 be the subtrees in the refutation deriving the literals x and $\overline{x}$ from initial clauses, and let s_0 and s_1 be the number of pebbles needed to pebble these subtrees reaching the literals x and $\overline{x}$. Since we are dealing with a tree-like refutation, either s_0 or s_1 must be smaller than s, this is because in order to place a pebble on the empty clause the two subtrees must be previously pebbled, and the pebbles in one of the subtrees do not affect the pebbling of the other one. W.l.o.g. let us consider $s_0 < s$. Also, T_0 and T_1 have depth smaller than d.

Applying the partial assignment $x = 0$ to all the clauses in T_0 (respectively the partial truth assignment $x = 1$ to the clauses in T_1), we obtain two refutation trees deriving the empty clause from two sets of clauses φ_0, φ_1. By induction, $w(\varphi_0 \vdash \square) \leq w(\varphi_0) + s_0 - 1 \leq w(\varphi) + s - 2$, and $w(\varphi_1 \vdash \square) \leq w(\varphi_1) + s_1 - 1 \leq w(\varphi) + s - 1$. Applying Lemma 1 we obtain $s - 1 \geq w(\varphi \vdash \square) - w(\varphi)$ ■

This result shows that when the width of the initial clauses is small with respect to the width of some internal clause, width lower bounds can be used to obtain space lower bounds for the restricted case of tree-like resolution. For example, for the case of a Tseitin formula related to an undirected graph G with odd marking, Ben-Sasson and Wigderson showed that the width is at least the expansion of G. By the above result, this implies a space lower bound for tree like resolution of at least the expansion of G minus the maximal degree of the graph, which is a little worse than the space lower bound for general resolution for these formulas obtained in Theorem 2.

5 Discussion

We have shown lower bounds for the resolution space of Tseitin and Pigeonhole formulas. These lower bounds are optimal since matching upper bounds exist. Besides the interest the bounds have on their own for a better understanding of the mentioned classes of formulas, these result point to a possible connection between the seemingly unrelated measures of resolution width and space. Similar lower bounds to the ones shown here, hold also for the case of width, and besides, it is known that for the case of tree-like resolution both width and space lower bounds imply exponentially larger size lower-bounds. However, the question of whether space lower bounds imply size lower bounds for other restrictions of resolution is still open [4]. The question of non-trivial lower bounds for the resolution space of random CNF formulas is another interesting open problem.

Acknowledgment: The author would like to thank Jochen Messner for helpful discussions on earlier versions of the paper.

References

1. Alekhnovich, M., Ben-Sasson, E., Razborov A. and Wigderson, A.: Space complexity in Propositional Calculus. Preliminary draft, July 1999.
2. Beame, P. and Pitassi, T.: Simplified and Improved Resolution Lower Bounds. In *Proc. 37th IEEE Symp. on Foundations of Computer Science*, (1996) 274–282.
3. Ben-Sasson, E. and Widgerson, A.: Short proofs are narrow, resolution made simple. In *Proc. 31st ACM Symp. on Theory of Computing* (1999) 517–527.
4. Buss, S. and Pitassi, T.: Resolution and the Weak Pigeonhole Principle. In *Computer Science Logic 97*, (1997), proceedings to appear in Springer Verlag.
5. Chvátal, V. and Szemerédi, E.: Many hard examples for resolution. *Journal of the ACM*, **35** (1988) 759–768.
6. Esteban, J.L. and Torán, J.: Space bounds for resolution. In *Proc. 16th STACS*, Springer Verlag LNCS **1563** (1999) 551–561.
7. Haken, A: The intractability of resolution *Theoretical Computer Science* **35** (1985) 297–308.
8. Kleine Büning, H. and Lettman, T.: *Aussangenlogik: Deduktion und Algorithmen*, B.G. Teubner Stuttgart (1994).
9. Margulis, A.: Explicit construction of concentrators. *Problems Information Transmission* **9** (1973) 71–80.
10. Messner, J.: Personal communication 1999.
11. Schöning, U.: Better expanders and superconcentrators by Kolmogorov complexity. In *Proceedings 4th International Colloquium on Structural Information and Communication Complexity, Sirocco 97* Carleton Scientific (1997) 138–151.
12. Schöning, U.: Resolution proofs, exponential bounds and Kolmogorov complexity. In *Proc. 22nd MFCS Conference*, Springer Verlag LNCS **1295** (1997) 110–116.

[4] Recently in [1] it has been shown that this is not true for the case of general resolution

13. Tseitin, G.S. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, Part 2. Consultants Bureau (1968) 115–125.
14. Urquhart, A.: Hard examples for resolution. *Journal of the ACM* **34** (1987) 209–219.

Program Schemes, Arrays, Lindström Quantifiers, and Zero-One Laws

Iain A. Stewart*

Department of Mathematics and Computer Science,
Leicester University, Leicester LE1 7RH, U.K.
i.a.stewart@mcs.le.ac.uk
WWW home page: http://www.mcs.le.ac.uk/~istewart

Abstract. We characterize the class of problems accepted by a class of program schemes with arrays, NPSA, as the class of problems defined by the sentences of a logic formed by extending first-order logic with a particular uniform sequence of Lindström quantifiers. We prove that our logic, and consequently our class of program schemes, has a zero-one law. However, we show that there are problems definable in a basic fragment of our logic, and so also accepted by basic program schemes, which are not definable in bounded-variable infinitary logic. Hence, the class of problems NPSA is not contained in the class of problems defined by the sentences of partial fixed-point logic even though in the presence of a built-in successor relation, both NPSA and partial fixed-point logic capture the complexity class **PSPACE**.

1 Introduction

This paper is a continuation of the study of the classes of problems captured by different classes of program schemes (in this study, the particular emphasis is on a comparison with the classes of problems defined by the sentences of well-known logics from finite model theory). *Program schemes* form a model of computation that is amenable to logical analysis yet is closer to the general notion of a program than a logical formula is. Program schemes were extensively studied in the seventies (for example, see [3,7,11,17]), without much regard being paid to an analysis of resources, before a closer complexity analysis was undertaken in, mainly, the eighties (for example, see [12,14,22]). There are connections between program schemes and logics of programs, especially dynamic logic [8,15]. One might also view many query languages from database theory as classes of program schemes, although query languages tend to operate on relations as opposed to individual elements (for example, see the *while* language from [1,4,5] and the language BQL from [4,16]).

The results in [2,6,18,21] testify that the study of program schemes is intimately related with more mainstream logics from finite model theory. In [18], program schemes allowing assignments, while instructions with quantifier-free

* Supported by EPSRC Grants GR/K 96564 and GR/M 12933.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 374–388, 1999.

tests, non-determinism and access to arrays were studied but only in the presence of a built-in successor relation (the class of problems accepted by such program schemes was shown to be **PSPACE**). It is with these program schemes and their extensions, obtained by allowing universally quantified program schemes to appear as tests in while instructions, that we are concerned in this paper but in the absence of any built-in relations; that is, the class of program schemes NPSA (non-deterministic program schemes with arrays). Our class of program schemes NPSA is quite natural. It consists of the union of an infinite hierarchy of classes of program schemes

$$\mathrm{NPSA}(1) \subseteq \mathrm{NPSA}(2) \subseteq \mathrm{NPSA}(3) \subseteq \ldots$$

The program schemes of NPSA(1) are built by allowing assignments, while instructions with quantifier-free tests, non-determinism and access to arrays (full details follow later). The program schemes of NPSA(2) are built from program schemes of NPSA(1) by universally quantifying free variables. The program schemes of NPSA(3) are built as are the program schemes of NPSA(1) except that tests in while instructions can be program schemes of NPSA(2). The program schemes of NPSA(4) are built from program schemes of NPSA(3) by universally quantifying free variables; and so on.

What is crucial is our definition of the semantics. Consider, for example, a while instruction in a program scheme ρ of NPSA(3) where the test is a program scheme ρ' of NPSA(2). In order to evaluate whether the test is true or not, the arrays from ρ are not 'passed over' to the program scheme ρ': the evaluation of ρ' has no access to the arrays of ρ. After evaluation of ρ' has been completed, the computation of the program scheme ρ resumes accordingly with its arrays having exactly the same values as they had immediately prior to the evaluation of ρ'. It is essentially our semantic definition that enables us to characterize the class of problems accepted by the program schemes of NPSA as the class of problems defined by the sentences of a logic $(\pm\Omega)^*[\mathrm{FO}]$ formed by extending first-order logic with a particular uniform (or vectorized) sequence of Lindström quantifiers (where this uniform sequence of Lindström quantifiers corresponds to a **PSPACE**-complete problem Ω). Moreover, we show that the logic $(\pm\Omega)^*[\mathrm{FO}]$ has a zero-one law; but not because it is a fragment of bounded-variable infinitary logic, as is so often the case in finite model theory, for we show that there are problems definable in NPSA (in NPSA(1) even) which are not definable in bounded-variable infinitary logic. Consequently, whilst both NPSA and partial fixed-point logic capture the complexity class **PSPACE** in the presence of a built-in successor relation, there are problems in NPSA which are not definable in partial-fixed point logic. If our semantics were such as to allow for universal quantification over arrays then we could simply guess a successor relation and hold our guesses in an array, use universal quantification to verify that the guessed relation was indeed a successor relation and subsequently use this guessed relation as our successor relation throughout. Consequently, we would have captured **PSPACE** and not the interesting logics (with zero-one laws but which are not fragments of bounded-variable infinitary logic) encountered in this paper.

2 Preliminaries

Ordinarily, a *signature* σ is a tuple $\langle R_1, \ldots, R_r, C_1, \ldots, C_c \rangle$, where each R_i is a relation symbol, of arity a_i, and each C_j is a constant symbol. However, we sometimes consider signatures in which there are no constant symbols; that is, *relational signatures*. A *finite structure* $\mathcal{A}$ *over the signature* σ, or σ*-structure*, consists of a finite *universe* or *domain* $|\mathcal{A}|$ together with a relation R_i of arity a_i, for every relation symbol R_i of σ, and a constant $C_j \in |\mathcal{A}|$, for every constant symbol C_j (by an abuse of notation, we do not distinguish between constants or relations and constant or relation symbols). A finite structure $\mathcal{A}$ whose domain consists of n distinct elements has *size* n, and we denote the size of $\mathcal{A}$ by $|\mathcal{A}|$ also (this does not cause confusion). We only ever consider finite structures of size at least 2, and the class of all finite structures of size at least 2 over the signature σ is denoted $\mathrm{STRUCT}(\sigma)$. A *problem* over some signature σ consists of a subset of $\mathrm{STRUCT}(\sigma)$ that is closed under isomorphism; that is, if $\mathcal{A}$ is in the problem then so is every isomorphic copy of $\mathcal{A}$. Throughout, all our structures are finite.

We are now in a position to consider the class of problems defined by the sentences of first-order logic, FO: we denote this class of problems by FO also, and do likewise for other logics. It is widely acknowledged that, as a means for defining problems, first-order logic leaves a lot to be desired especially when we have in mind developing a relationship between computational complexity and logical definability. In particular, every first-order definable problem can be accepted by a logspace deterministic Turing machine yet there are problems in the complexity class **L** which can not be defined in first-order logic (one such being the problem consisting of all those structures, over any signature, that have even size). One way of increasing the expressibility of FO is to augment FO with a uniform or vectorized sequence of Lindström quantifiers, or operator for short (the reader is referred to [10] for details). The archetypal example of such an extension is Immerman's transitive closure logic [13], and we shall see another such extension of FO soon.

An alternative and more computational means for defining classes of problems is to use program schemes. A *program scheme* $\rho \in \mathrm{NPSA}(1)$ involves a finite set $\{x_1, x_2, \ldots, x_k\}$ of *variables*, for some $k \geq 1$, and is over a signature σ. It consists of a finite sequence of *instructions* where each instruction, apart from the first and the last, is one of the following:

- an *assignment instruction* of the form '$x_i := y$', where $i \in \{1, 2, \ldots, k\}$ and where y is a variable from $\{x_1, x_2, \ldots, x_k\}$, a constant symbol of σ or one of the special constant symbols 0 and *max* which do not appear in any signature;
- an *assignment instruction* of the form '$x_i := A[y_1, y_2, \ldots, y_d]$' or '$A[y_1, y_2, \ldots, y_d] := y_0$', for some $i \in \{1, 2, \ldots, k\}$, where each y_j is a variable from $\{x_1, x_2, \ldots, x_k\}$, a constant symbol of σ or one of the special constant symbols 0 and *max* which do not appear in any signature, and where A is an array symbol of dimension d;
- a *guess instruction* of the form 'GUESS x_i', where $i \in \{1, 2, \ldots, k\}$; or

– a *while instruction* of the form 'WHILE φ DO $\alpha_1; \alpha_2; \ldots; \alpha_q$ OD', where φ is a quantifier-free formula of FO($\sigma \cup \{0, max\}$), whose free variables are from $\{x_1, x_2, \ldots, x_k\}$, and where each of $\alpha_1, \alpha_2, \ldots, \alpha_q$ is another instruction of one of the forms given here (note that there may be nested while instructions).

The first instruction of ρ is 'INPUT$(x_1, x_2, \ldots, x_l)$' and the last instruction is 'OUTPUT$(x_1, x_2, \ldots, x_l)$', for some l where $1 \leq l \leq k$. The variables $x_1, x_2, \ldots, x_l$ are the *input-output variables* of ρ, the variables $x_{l+1}, x_{l+2}, \ldots, x_k$ are the *free variables* of ρ and, further, any free variable of ρ never appears on the left-hand side of an assignment instruction nor in a guess instruction. Essentially, free variables appear in ρ as if they were constant symbols.

A program scheme $\rho \in$ NPSA(1) over σ with s free variables, say, takes a σ-structure $\mathcal{A}$ and s additional values from $|\mathcal{A}|$, one for each free variable of ρ, as input; that is, an expansion $\mathcal{A}'$ of $\mathcal{A}$ by adjoining s additional constants. The program scheme ρ computes on $\mathcal{A}'$ in the obvious way except that:

– execution of the instruction 'GUESS x_i' non-deterministically assigns an element of $|\mathcal{A}|$ to the variable x_i;
– the constants 0 and max are interpreted as two arbitrary but distinct elements of $|\mathcal{A}|$; and
– initially, every input-output variable and every array element is assumed to have the value 0.

Note that throughout a computation of ρ, the value of any free variable does not change. The expansion $\mathcal{A}'$ of the structure $\mathcal{A}$ is *accepted* by ρ, and we write $\mathcal{A}' \models \rho$, if, and only if, there exists a computation of ρ on this expansion such that the output-instruction is reached with all input-output variables having the value max. (We can easily build the usual 'if' and 'if-then-else' instructions using while instructions: see, for example, [18]. Henceforth, we shall assume that these instructions are at our disposal.)

We want the sets of structures accepted by our program schemes to be problems, i.e., closed under isomorphism, and so we only ever consider program schemes ρ where a structure is accepted by ρ when 0 and max are given two distinct values from the universe of the structure if, and only if, it is accepted no matter which pair of distinct values is chosen for 0 and max. This is analogous to how we build a successor relation or 2 constant symbols into a logic (see [10]). Indeed, we can build a successor relation into our program schemes of NPSA(1) so as to obtain the class of program schemes NPSA$_s$(1). As with our logics, we write NPSA(1) and NPSA$_s$(1) to also denote the class of problems accepted by the program schemes of NPSA(1) and NPSA$_s$(1), respectively. It was proven in [18] that a problem is in the complexity class **PSPACE** if, and only if, it is in NPSA$_s$(1).

Henceforth, we think of our program schemes as being written in the style of a computer program. That is, each instruction is written on one line and while instructions (and, similarly, if and if-then-else instructions) are split so that 'WHILE φ DO' appears on one line, 'α_1' appears on the next, 'α_2' on the

next, and so on (of course, if any α_i is a while, if or if-then-else instruction then it is split over a number of lines in the same way). The instructions are labelled 1, 2, and so on, according to the line they appear on. In particular, every instruction is considered to be an assignment, a guess or a test. An *instantaneous description* (*ID*) of a program scheme on some input consists of a value for each variable, the number of the instruction about to be executed and values for all array elements. A *partial ID* consists of just a value for each variable and the number of the instruction about to be executed. One *step* in a program scheme computation is the execution of one instruction, which takes one ID to another, and we say that a program scheme can *move* from one ID to another if there exists a sequence of steps taking the former ID to the latter.

3 Complete Problems

Definition 1. *Let the signature $\sigma_{TR} = \langle E, P, T, C, D \rangle$, where E is a binary relation symbol, P and T are unary relation symbols and C and D are constant symbols. We can envisage any σ_{TR}-structure $\mathcal{A}$ as a digraph (possibly with self-loops) whose edge relation is E and with distinguished vertices C, the* source*, and D, the* sink*. The relation P can be seen as providing a partition of the vertices and the relation T a subset of the vertices upon which* tokens *are initially placed. All tokens are indistinguishable and any vertex has upon it at most one token. Let us call a σ_{TR}-structure $\mathcal{A}$ a* token digraph*.*

Just as one can traverse a path in a digraph by moving along edges, so one can traverse a path in a token digraph $\mathcal{A}$. However, as to how edges can be traversed is different from the usual notion. Consider an edge $(u, v) \in E$ for which both u and v are in P and such that a traveller *is at vertex u (the traveller traverses a path of edges in the digraph). The edge (u, v) can only be traversed by the traveller moving as follows.*

- *The traveller moves from u via the edge (u, u') to a vertex u' not in P upon which exactly one token resides;*
- *then from u' via the edge (u', v') to a vertex v' not in P upon which no token resides, if $v' \neq u'$, and at the same time taking the token previously at u' to v', or by moving from u' via the edge (u', u') (if it exists) to u' (so that the token remains at u'); and finally*
- *by moving from the vertex v' or u', whichever is the case, via the edge (v', v) or (u', v) to v.*

This is called a compound move *(such a move is illustrated in Fig. 1), and compound moves are the only moves the traveller is allowed to make. Any tokens which happen to initially lie in P are ignored and play no part in any path traversal. Also, the traveller only ever makes compound moves from a vertex of P (u above) to a vertex of P (v above). The problem* Token Reachability *is defined as all those σ_{TR}-structures, i.e., token digraphs, for which a path can be traversed starting at the source and ending at the sink where the edges are traversed only by compound moves. Any instance for which $C = D$, no matter*

whether C is in P or not, is a yes-instance (note that if $C \notin P$ then the traveller can not move). □

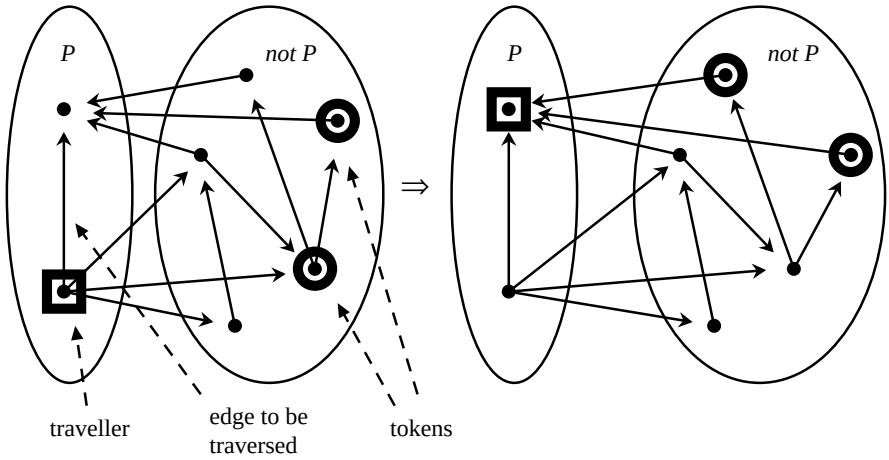


Figure 1. A compound move in a token digraph.

Theorem 1. *There is a quantifier-free first-order translation with* 2 *constants from any problem in NPSA*(1) *to the problem Token Reachability. Hence, Token Reachability is complete for NPSA*(1) *via quantifier-free first-order translations with* 2 *constants.*

Proof. (*Sketch*) Let ρ be a program scheme of NPSA(1) over the signature σ, possibly in which if instructions and if-then-else instructions occur. W.l.o.g. (by introducing more variables if needs be), we may assume that: every array symbol only appears in assignment instructions; no constant symbol appears in any assignment instruction involving an array symbol; and there is only one array symbol B, and this array symbol has dimension $d \geq 1$. Suppose that ρ involves the variables $x_1, x_2, \ldots, x_k$ and that there are l instructions in ρ, numbered $1, 2, \ldots, l$.

Let $\mathcal{A}$ be a σ-structure of size $n \geq 2$. An element $\mathbf{u} = (u_0, u_1, \ldots, u_k)$ of $\{1, 2, \ldots, l\} \times |\mathcal{A}|^k$ encodes a partial ID of the program scheme ρ on input $\mathcal{A}$ via: a computation of ρ on $\mathcal{A}$ is about to execute instruction u_0 and the variables $x_1, x_2, \ldots, x_k$ currently have the values $u_1, u_2, \ldots, u_k$, respectively. Henceforth, we identify partial IDs of ρ and the elements of $\{1, 2, \ldots, l\} \times |\mathcal{A}|^k$.

Let the digraph G_0 have vertex set $U_0 = \{1, 2, \ldots, l\} \times |\mathcal{A}|^{k+2}$ and edge set $E_0 = E_0^1 \cup E_0^2 \cup E_0^3$, where E_0^1, E_0^2, and E_0^3 are defined as follows (in our eventual token digraph, the vertices of U_0 will play the role of the vertices of P from Definition 1).

- $E_0^1 = \{((\mathbf{u}, 0, 0), (\mathbf{u}, max, t)), ((\mathbf{u}, max, t), (\mathbf{v}, 0, 0)) \in U_0 \times U_0$: instruction u_0 is of the form $x_j := B[x_{i_1}, x_{i_2}, \ldots, x_{i_d}]$ and it is possible for ρ on input $\mathcal{A}$ to move from partial ID $\mathbf{u}$ to partial ID $\mathbf{v}$ in one step, and $v_j = t\}$.

– $E_0^2 = \{((\mathbf{u},0,0),(\mathbf{v},0,0)) \in U_0 \times U_0$: instruction u_0 is of the form $B[x_{i_1}, x_{i_2}, \ldots, x_{i_d}] := x_j$ and it is possible for ρ on input $\mathcal{A}$ to move from partial ID $\mathbf{u}$ to partial ID $\mathbf{v}$ in one step$\}$.

– $E_0^3 = \{((\mathbf{u},0,0),(\mathbf{v},0,0)) \in U_0 \times U_0$: instruction u_0 does not involve the array symbol B and it is possible for ρ on input $\mathcal{A}$ to move from partial ID $\mathbf{u}$ to partial ID $\mathbf{v}$ in one step$\}$.

Of course, whether ρ on input $\mathcal{A}$ actually moves from partial ID $\mathbf{u}$ to partial ID $\mathbf{v}$ in one step at some point in a computation depends upon whether $\mathbf{u}$ can be reached from the initial ID and it might also depend upon the actual value of the array B at that time. The edges of $E_0^1 \cup E_0^2$ reflect *potential* one-step moves from partial ID $\mathbf{u}$ to partial ID $\mathbf{v}$ (moves which are dependent upon the value of B).

For each $\mathbf{w} \in |\mathcal{A}|^d$, define the digraph $G_{\mathbf{w}}$ to have vertex set $V_{\mathbf{w}} = |\mathcal{A}|$ (all vertex sets of such digraphs are disjoint). The edge set $E_{\mathbf{w}}$ of $G_{\mathbf{w}}$ consists of every possible edge between vertices of $V_{\mathbf{w}}$ including self-loops. For each vertex $(\mathbf{u},0,0) \in U_0$, let $H_{\mathbf{u}}$ be a digraph with one vertex $z_{\mathbf{u}}$ and one self-loop $(z_{\mathbf{u}}, z_{\mathbf{u}})$ (again, all such digraphs are disjoint). Let the digraph $\mathcal{G}$ consist of the disjoint union of the digraphs G_0, $\{G_{\mathbf{w}} : \mathbf{w} \in |\mathcal{A}|^d\}$ and $\{H_{\mathbf{u}} : (\mathbf{u},0,0) \in U_0\}$, together with the following additional edges between the vertices of these digraphs.

– If instruction u_0 is of the form $x_j := B[x_{i_1}, x_{i_2}, \ldots, x_{i_d}]$ then there are edges $\{((\mathbf{u},0,0), z_{\mathbf{u}}), (z_{\mathbf{u}}, (\mathbf{u}, max, t)) : (\mathbf{u},0,0) \in U_0, t \in |\mathcal{A}|\}$, and for every edge $((\mathbf{u}, max, t), (\mathbf{v},0,0))$ of E_0^1, there are edges $((\mathbf{u}, max, t), t^{\mathbf{w}})$ and $(t^{\mathbf{w}}, (\mathbf{v},0,0))$, where $\mathbf{w} = (u_{i_1}, u_{i_2}, \ldots, u_{i_d})$ and $t^{\mathbf{w}}$ is vertex t of $V_{\mathbf{w}}$.
– If instruction u_0 is of the form $B[x_{i_1}, x_{i_2}, \ldots, x_{i_d}] := x_j$ then for every edge $((\mathbf{u},0,0),(\mathbf{v},0,0))$ of E_0^2, there are edges $\{((\mathbf{u},0,0), t^{\mathbf{w}}), (u_j^{\mathbf{w}}, (\mathbf{v},0,0)) : t \in |\mathcal{A}|\}$, where $\mathbf{w} = (u_{i_1}, u_{i_2}, \ldots, u_{i_d})$ and $t^{\mathbf{w}}$ (resp. $u_j^{\mathbf{w}}$) is vertex t (resp. u_j) of $V_{\mathbf{w}}$.
– If instruction u_0 does not involve the array symbol B then for every edge $((\mathbf{u},0,0),(\mathbf{v},0,0))$ of E_0^3, there are edges $((\mathbf{u},0,0), z_{\mathbf{u}})$ and $(z_{\mathbf{u}}, (\mathbf{v},0,0))$.

That portion of the digraph $\mathcal{G}$ corresponding to a one-step move of ρ on input $\mathcal{A}$ from partial ID $\mathbf{u}$ to partial ID $\mathbf{v}$ can be visualized as in Figs. 2 and 3 when instruction u_0 is of the form $x_j := B[x_{i_1}, x_{i_2}, \ldots, x_{i_d}]$ and $B[x_{i_1}, x_{i_2}, \ldots, x_{i_d}] := x_j$, respectively.

We can now extend $\mathcal{G}$ to a token digraph: let the source be the vertex $(1, \mathbf{0}, 0, 0)$ of U_0 and the sink be the vertex $(l, \mathbf{max}, 0, 0)$ of U_0; let $P = U_0$; and let $T = \{0^{\mathbf{w}} : 0^{\mathbf{w}}$ is the vertex 0 of $V_{\mathbf{w}}$, where $\mathbf{w} \in |\mathcal{A}|^d\} \cup \{z_{\mathbf{u}} : (\mathbf{u},0,0) \in U_0\}$. It is not difficult to see that $\mathcal{A}$ is accepted by the program scheme ρ if, and only if, $\mathcal{G}$ is a yes-instance of the problem *Token Reachability*.

The token digraph $\mathcal{G}$ can easily be described in terms of $\mathcal{A}$ by a quantifier-free first-order formula with 2 constants (see comparable constructions in, for example, [19]), and so the problem *Token Reachability* is hard for NPSA(1) via quantifier-free first-order translations with 2 constants. Moreover, *Token Reachability* can be accepted by the following program scheme of NPSA(1).

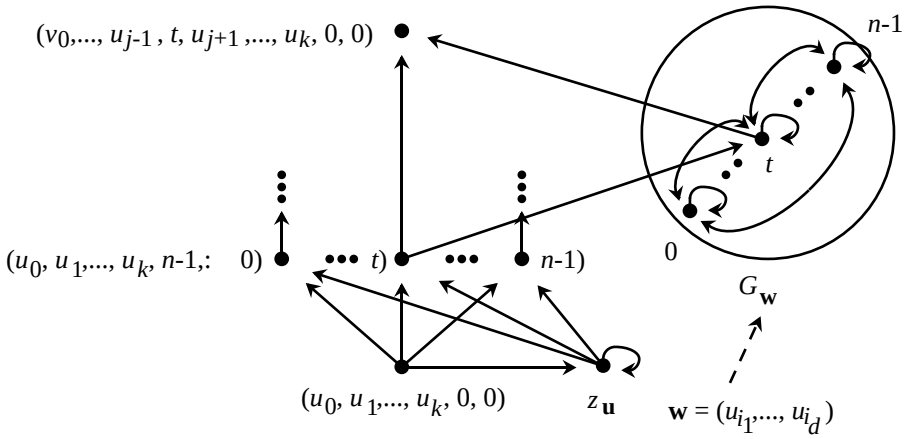


Figure 2. A portion of the digraph $\mathcal{G}$ corresponding to an instruction of the form $x_j := B[x_{i_1}, x_{i_2}, \ldots, x_{i_d}]$.

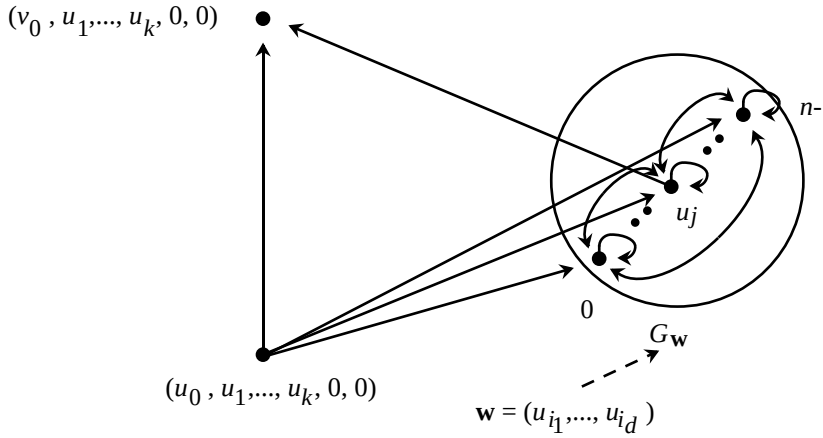


Figure 3. A portion of the digraph $\mathcal{G}$ corresponding to an instruction of the form $B[x_{i_1}, x_{i_2}, \ldots, x_{i_d}] := x_j$.

```
input(u, v, u′, v′, w)
guess w
while w = max do
   guess w
   if T(w) then B[w] := max fi
   guess w
   od
if ¬P(C) ∧ C ≠ D then 'loop forever' fi
u := C
while u ≠ D do
   guess v
   if ¬P(v) ∨ ¬E(u, v) then 'loop forever' fi
   guess u′
```

```
      if P(u') ∨ ¬E(u,u') ∨ B[u'] = 0 then 'loop forever' fi
      guess v'
      if P(v') ∨ ¬E(u',v') ∨ (B[v'] = max ∧ u' ≠ v')
                                  ∨(T(v') ∧ M[v'] = 0 ∧ u' ≠ v') then
         'loop forever' fi
      if E(v',v) then
         B[u'] := 0; B[v'] := max; M[u'] := max; u := v
      else
         'loop forever' fi
      od
   (u,v,u',v',w) := (max,max,max,max,max)
   output(u,v,u',v',w)
```

Some explanation is in order (beyond the obvious short-hand we use in our description). Our program scheme ρ involves two array symbols, B and M, both of dimension 1. Suppose that some σ_{TR}-structure $\mathcal{A}$ is accepted by ρ. The first part of ρ guesses a set of vertices upon which tokens initially lie: this set is $\{w : B[w] = max\}$. Throughout the execution, the array B details the locations of these tokens as they are moved about: call these tokens the B-type tokens. No other token is ever moved (and by 'moved' we include tokens which are moved along self-loops). The array M is initially identically 0 but whenever a vertex w upon which a B-type token lies is involved in a compound move, the array element $M[w]$ is set at max. Consequently, at any particular time we know where the B-type tokens are (the elements w for which $B[w] = max$) and we know where the other tokens are (the elements w for which $T(w) \wedge M[w] = 0$).

The code within the second while-loop is the code associated with making a compound move. The variable v holds the vertex $v \in P$ (see Definition 1), the variable u' holds the vertex $u' \notin P$ and the variable v' holds the vertex $v' \notin P$ (or possibly u'). Note that in choosing u', we must ensure that a B-type token currently lies on u' (as these are the only tokens which we are allowed to move). This is done by checking whether $B[u'] = max$. Also, in choosing v' (if it is to be different from u') we must ensure that no B-type token lies on v' nor no non-B-type token. Thus, $\mathcal{A}$ is in *Token Reachability*.

Conversely, suppose that $\mathcal{A}$ is in *Token Reachability*. In the initial phase of the execution of ρ on $\mathcal{A}$, we can guess every token to be a B-type token. The result follows. □

Thus we obtain the following corollary.

Corollary 1. *Token Reachability is complete for* **PSPACE** *via quantifier-free first-order translations with successor.*

Proof. By [18], any problem in **PSPACE** can be accepted by some program scheme of $\mathrm{NPSA}_s(1)$. Hence, the result follows from Theorem 1. □

4 Logics and Program Schemes

An important point to note is that whereas the usual existential quantifier is catered for in program schemes of NPSA(1) via the guess instruction (intuitively speaking), there is no such analogous modelling of the universal quantifier. Consequently, we extend our program schemes by introducing universal quantification in the following manner.

Definition 2. *Let σ be some signature. For some $m \geq 1$, let the program scheme $\rho \in NPSA(2m-1)$ be over the signature σ and involve the variables $x_1, x_2, \ldots, x_k$. Suppose that the variables $x_1, x_2, \ldots, x_l$ are the input-output variables of ρ, the variables $x_{l+1}, x_{l+2}, \ldots, x_{l+s}$ are the free variables, and the remaining variables are the bound variables (note that if $m = 1$ then ρ has no bound variables but that this may not be the case if $m > 1$). Let $x_{i_1}, x_{i_2}, \ldots, x_{i_p}$ be free variables of ρ, for some p such that $1 \leq p \leq s$. Then*

$$\forall x_{i_1} \forall x_{i_2} \ldots \forall x_{i_p} \rho$$

is a program scheme of $NPSA(2m)$, which we denote by ρ', with no input-output variables, with free variables those of $\{x_{l+1}, x_{l+2}, \ldots, x_{l+s}\} \setminus \{x_{i_1}, x_{i_2}, \ldots, x_{i_p}\}$ and with the remaining variables of $\{x_1, x_2, \ldots, x_k\}$ as its bound variables.

A program scheme such as ρ' takes expansions $\mathcal{A}'$ of σ-structures $\mathcal{A}$ by adjoining $s - p$ constants as input (one for each free variable), and ρ' accepts such an expansion $\mathcal{A}'$ if, and only if, for every expansion $\mathcal{A}''$ of $\mathcal{A}'$ by p additional constants (one for each variable x_{i_j}), $\mathcal{A}'' \models \rho$. □

Note that the different computations of ρ on expansions $\mathcal{A}''$ of $\mathcal{A}'$, in Definition 2, are all such that all arrays are initialised to 0.

Definition 3. *Let σ be some signature. A program scheme $\rho' \in NPSA(2m-1)$, for some $m \geq 2$, over the signature σ and involving the variables of $\{x_1, x_2, \ldots, x_k\}$, is formed exactly as are the program schemes of $NPSA(1)$, with the input-output and free variables defined accordingly, except that the test in some while instruction is a program scheme $\rho \in NPSA(2m-2)$ whose free and bound variables are all from $\{x_1, x_2, \ldots, x_k\}$ (note that ρ has no input-output variables). However, there are further stipulations:*

- *all free variables in any test $\rho \in NPSA(2m-2)$ in any while instruction are input-output or free variables of ρ';*
- *the bound variables of ρ' consist of all bound variables of any test $\rho \in NPSA(2m-2)$ in any while instruction (and no bound variable is ever an input-output or free variable of ρ); and*
- *this accounts for all variables of $\{x_1, x_2, \ldots, x_k\}$.*

Of course, any free variable of ρ' never appears on the left-hand side of an assignment instruction or in a guess instruction.

A program scheme $\rho' \in NPSA(2m-1)$ takes expansions $\mathcal{A}'$ of σ-structures $\mathcal{A}$ by adjoining s constants as input, where s is the number of free variables,

and computes on $\mathcal{A}'$ in the obvious way except that when some while instruction is encountered, the test, which is a program scheme $\rho \in NPSA(2m-2)$, is evaluated according to the expansion of $\mathcal{A}'$ by the current values of any relevant input-output variables of ρ' (which may be free in ρ). In order to evaluate this test, all arrays in ρ are initialised to 0 and when the test has been evaluated the computation of ρ' resumes accordingly with its arrays having exactly the same values as they had immediately before the test was evaluated. □

In a program scheme such as ρ' in Definition 3, the only information which can be 'passed' to a test evaluation is the current values of the relevant input-output or free variables. Arrays can not be used to pass information across. If our semantics were such as to allow universal quantification over arrays then we could build our own successor relation.

Theorem 1 allows us to relate the class of problems accepted by the program schemes of NPSA with the class of problems defined by the sentences of the logic $(\pm\mathrm{TR})^*[\mathrm{FO}]$. For each $m \geq 1$, we define the fragment $\pm\mathrm{TR}(m)$ of $(\pm\mathrm{TR})^*[\mathrm{FO}]$ as follows (see [2] for similarly defined fragments of other logics).

- $\pm\mathrm{TR}(1)$ consists of all formulae of the form $\mathrm{TR}[\lambda\mathbf{x},\mathbf{y}\psi_E,\mathbf{x}\psi_P,\mathbf{x}\psi_T](\mathbf{u},\mathbf{v})$, where ψ_E, ψ_P and ψ_T are quantifier-free first-order formulae and where $\mathbf{u}$ and $\mathbf{v}$ are tuples of constant symbols or variables.
- $\pm\mathrm{TR}(m+1)$, for odd $m \geq 1$, consists of the universal closure of $\pm\mathrm{TR}(m)$; that is, the set of formulae of the form $\forall z_1 \forall z_2 \ldots \forall z_k \psi$, where ψ is a formula of $\pm\mathrm{TR}(m)$.
- $\pm\mathrm{TR}(m+1)$, for even $m \geq 2$, consists of the set of formulae of the form

$$\mathrm{TR}[\lambda\mathbf{x},\mathbf{y}(\psi_E^1 \vee \neg\psi_E^2),\mathbf{x}(\psi_P^1 \vee \neg\psi_P^2),\mathbf{y}(\psi_T^1 \vee \neg\psi_T^2)](\mathbf{u},\mathbf{v}),$$

  where ψ_E^1, ψ_E^2, ψ_P^1, ψ_P^2, ψ_T^1 and ψ_T^2 are formulae of $\pm\mathrm{TR}(m)$ and where $\mathbf{u}$ and $\mathbf{v}$ are tuples of constant symbols or variables.

A straightforward induction yields that:

- for every odd $m \geq 1$, every formula in the closure of $\pm\mathrm{TR}(m)$ under $\wedge$, $\vee$ and $\exists$ is logically equivalent to a formula of $\pm\mathrm{TR}(m)$; and
- for every even $m \geq 1$, every formula in the closure of $\pm\mathrm{TR}(m)$ under $\wedge$, $\vee$ and $\forall$ is logically equivalent to a formula of $\pm\mathrm{TR}(m)$.

Consequently, $(\pm\mathrm{TR})^*[\mathrm{FO}] = \cup\{\mathrm{TR}(m) : m \geq 1\}$. Another easy induction yields the following result.

Corollary 2. *In the presence of 2 built-in constant symbols, $\pm TR(m) = NPSA(m)$, for each $m \geq 1$; and so $(\pm TR)^*[FO] = NPSA$.* □

The following is immediate from Corollaries 1 and 2.

Corollary 3. *In the presence of a built-in successor relation, $(\pm TR)^*[FO_s] = TR^1[FO_s] =$* **PSPACE**. □

Let us now focus on a comparison of NPSA with other classes of logically-defined problems. *Bounded-variable infinitary logic*, $\mathcal{L}^{\omega}_{\infty\omega} = \cup\{\mathcal{L}^{k}_{\infty\omega} : k \geq 1\}$, plays a prominent role in finite model theory (see [10]). In particular, it subsumes many logics from finite model theory, notably transitive closure logic, path system logic, least-fixed point logic and partial-fixed point logic.

Let $\sigma_2 = \langle E \rangle$, where E is a binary relation symbol. We can think of a σ_2-structure $\mathcal{A}$ as an undirected graph via 'there is an edge (u, v) if, and only if, $u \neq v \wedge (E(u, v) \vee E(v, u))$ holds in $\mathcal{A}$'. Define the problem CUB as

$$\text{CUB} = \{\mathcal{A} \in \text{STRUCT}(\sigma_2) : \text{the graph } \mathcal{A} \text{ has a subset of edges inducing a regular subgraph of degree 3}\},$$

where the subgraph induced by a set of edges F of a graph is that subgraph whose vertex set consists of all those vertices incident with at least one edge of F and whose edge set consists of F. Of concern to us is the result from [20] that the problem CUB is not definable in $\mathcal{L}^{\omega}_{\infty\omega}$ (even when we allow *counting quantifiers* in $\mathcal{L}^{\omega}_{\infty\omega}$: see [20]).

Proposition 1. *Any problem definable by a sentence of the form*

$$CUB[\lambda \mathbf{x}, \mathbf{y} \psi(\mathbf{x}, \mathbf{y})],$$

where $|\mathbf{x}| = \mathbf{y}| = k$, *for some* k, *and* ψ *is a quantifier-free first-order formula with* 2 *constants, can be accepted by a program scheme of NPSA*(1).

Proof. We assume throughout that $k = 2$: the general case is similar. Let $\mathcal{A}$ be some structure, of size n, over the underlying signature σ. Our program scheme $\rho \in \text{NPSA}(1)$ proceeds as follows. We begin by 'guessing' a set of (at most $n^2(n^2 - 1)/2$) distinct potential edges in the graph $\mathcal{G}_{\mathcal{A}}$ described by ψ (interpreted in $\mathcal{A}$). We use the 4-dimensional arrays A_1, A_2, B_1 and B_2 in order to store the guessed list of edges as follows. We guess elements u_1^1, u_2^1, v_1^1 and v_2^1 of $|\mathcal{A}|$, ensuring that it is not the case that all of these elements are equal to 0, and we set

$$A_1[0,0,0,0] := u_1^1, A_2[0,0,0,0] := u_2^1, B_1[0,0,0,0] := v_1^1, B_2[0,0,0,0] := v_2^1.$$

Next we guess elements u_1^2, u_2^2, v_1^2 and v_2^2 of $|\mathcal{A}|$, and check that $(u_1^2, u_2^2, v_1^2, v_2^2)$ is different from $(0,0,0,0)$ and $(u_1^1, u_2^1, v_1^1, v_2^1)$. If so then we set

$$A_1[u_1^1, u_2^1, v_1^1, v_2^1] := u_1^2, A_2[u_1^1, u_2^1, v_1^1, v_2^1] := u_2^2,$$
$$B_1[u_1^1, u_2^1, v_1^1, v_2^1] := v_1^2, B_2[u_1^1, u_2^1, v_1^1, v_2^1] := v_2^2.$$

We stop if each of u_1^2, u_2^2, v_1^2 and v_2^2 is equal to max. Next, we guess elements u_1^3, u_2^3, v_1^3 and v_2^3 of $|\mathcal{A}|$ and check that $(u_1^3, u_2^3, v_1^3, v_2^3)$ is different from $(0,0,0,0)$, $(u_1^1, u_2^1, v_1^1, v_2^1)$ and $(u_1^2, u_2^2, v_1^2, v_2^2)$. If so then we set

$$A_1[u_1^2, u_2^2, v_1^2, v_2^2] := u_1^3, A_2[u_1^2, u_2^2, v_1^2, v_2^2] := u_2^3,$$
$$B_1[u_1^2, u_2^2, v_1^2, v_2^2] := v_1^3, B_2[u_1^2, u_2^2, v_1^2, v_2^2] := v_2^3.$$

We stop if each of u_1^3, u_2^3, v_1^3 and v_2^3 is equal to max; and so on.

Any computation of ρ which completes this first phase is such that the arrays now encode the (non-empty) list of $m-1$ distinct 'potential edges'

$$((u_1^1, u_2^1), (v_1^1, v_2^1)), ((u_1^2, u_2^2), (v_1^2, v_2^2)), \ldots, ((u_1^{m-1}, u_2^{m-1}), (v_1^{m-1}, v_2^{m-1})).$$

We now check that the potential edges on this list are indeed edges of $\mathcal{G}_{\mathcal{A}}$ by verifying that

$$(u_1^i \neq v_1^i \vee u_2^i \neq v_2^i) \wedge (\psi(u_1^i, u_2^i, v_1^i, v_2^i) \vee \psi(v_1^i, v_2^i, u_1^i, u_2^i))$$

holds in $\mathcal{A}$, for $i = 1, 2, \ldots, m-1$.

Finally, we check that each vertex incident with some edge in our list of edges is incident with exactly 3 such edges: if so, we accept the input structure $\mathcal{A}$ otherwise we reject it. It is clear that all of the above can be implemented in a program scheme of NPSA(1); and that the resulting program scheme accepts exactly the problem defined by the sentence CUB$[\lambda \mathbf{x}, \mathbf{y} \psi(\mathbf{x}, \mathbf{y})]$. □

Using the facts that the problem CUB is not definable in $\mathcal{L}^{\omega}_{\infty\omega}$ and that there are non-recursive problems which are definable in $\mathcal{L}^{\omega}_{\infty\omega}$, we immediately obtain the following corollary.

Corollary 4. *There are problems in NPSA*(1) *which are not definable in* $\mathcal{L}^{\omega}_{\infty\omega}$, *and there are problems in* $\mathcal{L}^{\omega}_{\infty\omega}$ *which are not definable in NPSA*(1). □

Note that whilst we know that there are problems in NPSA which are not definable in partial fixed-point logic (a fragment of bounded-variable infinitary logi, remember), we do not as yet know whether there are problems in partial fixed-point logic which are not definable in NPSA (although we suspect that there are).

5 Zero-One Laws

Let σ be a relational signature and let Ω be a problem over σ. Define the fraction

$$l_n(\Omega) = \frac{|\{\mathcal{A} : \mathcal{A} \in \text{STRUCT}(\sigma) \text{ has size } n \text{ and } \mathcal{A} \in \Omega\}|}{|\{\mathcal{A} : \mathcal{A} \in \text{STRUCT}(\sigma) \text{ has size } n\}|}$$

and define the (*labelled*) *asymptotic probability* of Ω, $l(\Omega)$, as

$$\lim_{n \to \infty} l_n(\Omega),$$

if it exists. We say that a logic or a class of program schemes has a *zero-one law* if every problem Ω (over a relational signature) definable by a sentence of the logic or accepted by a program scheme from the class is such that the asymptotic probability $l(\Omega)$ exists and is equal to either 0 or 1. For any logical sentence or program scheme ϕ (over a relational signature), we define $l(\phi)$ to be $l(\Omega)$ where Ω is the problem defined by ϕ.

A problem Ω over some (not necessarily relational) signature σ is *closed under extensions* if whenever a σ-structure $\mathcal{A}$ has a sub-structure in Ω then $\mathcal{A} \in \Omega$ (a σ-structure $\mathcal{A}'$ is a *sub-structure* of $\mathcal{A}$ if the universe of $\mathcal{A}'$ is contained in the universe of $\mathcal{A}$, any relation of $\mathcal{A}'$ is the restriction of the corresponding relation of $\mathcal{A}$ to $|\mathcal{A}'|$ and every constant of $\mathcal{A}'$ is the same as the corresponding constant of $\mathcal{A}$). The following theorem is essentially a generalization of Theorem 7.4 of [9] to extensions of first-order logic using a uniform sequence of Lindström quantifiers corresponding to a problem closed under extensions, where this problem need not just involve graphs but can be over any (not necessarily relational) signature.

Theorem 2. *Let Ω be a problem closed under extensions. Then the logic $(\pm\Omega)^*[FO]$ has a zero-one law.* □

The following corollary is immediate from Corollary 2 and Theorem 2, as the problem *Token Reachability* is closed under extensions.

Corollary 5. *The class of program schemes NPSA has a zero-one law.* □

6 Conclusions

In this paper we have investigated a naturally-defined class of program schemes, NPSA, which take finite structures as inputs, and proven that the class of problems accepted by the program schemes of NPSA coincides with the class of problems defined by the sentences of an extension of first-order logic using a uniform sequence of Lindström quantifiers (corresponding to a **PSPACE**-complete problem). Moreover, we have shown that the class of problems NPSA has a zero-one law and also that there are problems in NPSA which are not definable in bounded-variable infinitary logic. We feel that our general approach of investigating more 'computational versions of logics' (that is, classes of program schemes) than is often the case in finite model theory is completely natural, interesting and novel; and the results presented here and obtained in [2,6,21] further testify to this belief.

There remain many unanswered questions concerning classes of program schemes. The most notable ones arising from this paper are: '*Are there problems definable in partial fixed-point logic which are not accepted by any program scheme of NPSA*?'; and '*Is the hierarchy of program schemes NPSA*$(1) \subseteq$ *NPSA*$(2) \subseteq$ *NPSA*$(3) \subseteq \ldots$ *proper*?'. We conjecture that the answer to both of these questions is 'Yes'; although so far we have been unable to apply or extend the techniques of [2] to answer either of these questions. (In [2], the class of program schemes NPS, where array symbols are not allowed, was shown to be none other than transitive closure logic and an infinite proper hierarchy $\mathrm{NPS}(1) \subset \mathrm{NPS}(2) \subset \ldots$ was exhibited; and the class of program schemes NPSS, where array symbols are not allowed but access to a stack is, was shown to be none other than path system logic, which in turn is stratified Datalog, and an infinite proper hierarchy $\mathrm{NPSS}(1) \subset \mathrm{NPSS}(2) \subset \ldots$ was exhibited.)

References

1. Abiteboul, S., Vianu, V.: Generic computation and its complexity, Proc. 23rd Ann. ACM Symp. on Theory of Computing, ACM Press (1991) 209–219.
2. Arratia-Quesada, A.A., Chauhan, S.R., Stewart, I.A.: Hierarchies in classes of program schemes, J. Logic Computat. (to appear).
3. Brown, S., Gries, D., Szymanski, T.: Universality of data retrieval languages, Proc. 6th Ann. ACM Symp. on Principles of Programming Languages, ACM Press (1979) 110–117.
4. Chandra, A.: Programming primitives for database languages, Proc. 8th Ann. ACM Symp. on Principles of Programming Languages, ACM Press (1981) 50–62.
5. Chandra, A., Harel, D.: Structure and complexity of relational queries, J. Comput. System Sci. **25** (1982) 99–128.
6. Chauhan, S.R., Stewart, I.A.: On the power of built-in relations in certain classes of program schemes, Inform. Process. Lett. **69** (1999) 77-82.
7. Constable, R., Gries, D.: On classes of program schemata, SIAM J. Comput. **1** (1972) 66–118.
8. Courcelle, B.: Recursive applicative program schemes, Handbook of Theoretical Computer Science Vol. B (ed. van Leeuwen, J.), Elsevier (1990) 459–492.
9. Dawar, A., Grädel, E.: Generalized quantifiers and 0-1 laws, Proc. 10th IEEE Ann. Symp. on Logic in Computer Science, IEEE Press (1995) 54–64.
10. Ebbinghaus, H.D., Flum, J.: Finite Model Theory, Springer-Verlag (1995).
11. Friedman, H.: Algorithmic procedures, generalized Turing algorithms and elementary recursion theory, Logic Colloquium 1969 (ed. Gandy, R.O., Yates, C.M.E.), North-Holland (1971) 361–390.
12. Harel, D., Peleg, D.: On static logics, dynamic logics, and complexity classes, Inform. Control **60** (1984) 86–102.
13. Immerman, N.: Languages that capture complexity classes, SIAM J. Comput. **16** (1987) 760–778.
14. Jones, N.D., Muchnik, S.S.: Even simple programs are hard to analyze, J. Assoc. Comput. Mach. **24** (1977) 338–350.
15. Kozen, D., Tiuryn, J.: Logics of programs, Handbook of Theoretical Computer Science Vol. B (ed. van Leeuwen, J.), Elsevier (1990) 789–840.
16. Neven, F., Otto, M., Tyszkiewicz, J., Van den Bussche, J.: Adding for-loops to first-order logic, Lecture Notes in Computer Science Vol. 1540, Springer-Verlag (1999) 58–69.
17. Paterson, M., Hewitt, N.: Comparative schematology, Proc. Project MAC Conf. on Concurrent Systems and Parallel Computation, ACM Press (1970) 119–128.
18. Stewart, I.A.: Logical and schematic characterization of complexity classes, Acta Informat. **30** (1993) 61–87.
19. Stewart, I.A.: Context-sensitive transitive closure operators, Ann. Pure App. Logic **66** (1994) 277–301.
20. Stewart, I.A.: Logics with zero-one laws that are not fragments of bounded-variable infinitary logic, Math. Logic Quart. **41** (1997) 158–178.
21. Stewart, I.A.: Using program schemes to logically capture polynomial-time on certain classes of structures, University of Leicester Technical Report 1998/13 (1998).
22. Tiuryn, J., Urzyczyn, P.: Some relationships between logics of programs and complexity theory, Theoret. Comput. Sci. **60** (1988) 83–108.

Open Least Element Principle and Bounded Query Computation

L.D. Beklemishev *

Steklov Mathematical Institute
Gubkina 8, 117966 Moscow, Russia
e-mail: bekl@mi.ras.ru, beklemi@math.uni-muenster.de

Abstract. We show that elementary arithmetic formulated in the language with a free function symbol f and the least element principle for open formulas (where we assume that the symbols for all elementary functions are included in the language) does not prove the least element principle for bounded formulas in the same language. A related result is that composition and any number of unnested applications of bounded minimum operator are, in general, insufficient to generate the elementary closure of a function, even if all elementary functions are available. Thus, unnested bounded minimum operator is weaker than unnested bounded recursion.

1 Introduction and Motivation

This paper arose out of the problem of separating the schemes of Δ_1-induction and Σ_1-collection in arithmetic [4,6]. It turns out that this question is closely related to the comparison of different operators generating the elementary closure of a class of functions and to the problems of separating the corresponding systems of subrecursive arithmetic. These questions are also natural from a purely computational point of view.

In this paper we compare the relative strength of bounded μ-operator and bounded recursion. We show that unnested bounded μ-operator is, in general, weaker than unnested bounded recursion. In contrast, it is well known that, when nestings are allowed, each of the two operators together with composition is sufficient to generate the elementary closure of a class of functions.

We compare the strength of the two operators against the problem of computing the maximum of a function on a finite interval. In order to compute $\max_{i\leq x} f(i)$ on a Turing machine with a function oracle for f one needs of order x different oracle queries (see [2]). In particular, any Turing machine that may only ask a bounded number of queries cannot, in general, compute this function. This implies that the class of functions generated from all elementary functions and f by composition does not, in general, coincide with the elementary closure of a function f. This idea was used in [2] to show the independence of Σ_1-collection rule in arithmetic, which improved the result of Parsons on the independence of

* Supported by Alexander von Humboldt Foundation and RFBR grant 98-01-00249.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 389–404, 1999.

Σ_1-collection schema from the set of all true arithmetical Π_2-sentences (see also [8] for a related work on subrecursive degrees).

If unnested applications of bounded μ-operator are allowed on a par with composition, then the available power of computation increases compared to the bounded query oracle Turing machine. It is worth explaining here informally, why such a computation mechanism is still too weak to compute the maximum of f.

A μ-operator of the form $\mu i \leq x.\ R(i)$, where $R(i)$ is a bounded query predicate, can be interpreted in terms of a parallel bounded query machine. To evaluate this operator the machine generates $x+1$ independent subprocesses, i-th process P_i evaluates the predicate $R(i)$ and returns **true** or **false**. Then the machine runs through their outputs to find the least i such that $R(i)$ evaluates to **true**.

Since $R(x)$ is a bounded query predicate, all the subprocesses have a uniform bound on the number of queries each of them may ask. More important still, each subprocess P_i only returns **true** or **false**, that is, exactly one bit. This means that the processes cannot exchange too much information. This is crucial for the fact that $\max_{i\leq x} f(i)$ cannot be computed by such a machine: each subprocess P_i can only learn the values of f on a boundedly small subset of the large interval $[0, x]$, but it lacks the ability to communicate, say, the maximum of these values to other processes. (Compare with the usual algorithm of computing the maximum of f by querying successively $f(0), f(1), \ldots, f(x)$. Here, one has to always store the intermediate maximum value of f, which may potentially exceed any bounded number of bits.)

Of course, this rough idea will be made more precise in the proof of our main result. This proof is based on a recursion-theoretic diagonal construction that involves a combinatorial argument using infinite Ramsey theorem.

2 Statement of the Results

As usual, for a given predicate $R(x, \boldsymbol{v})$ the expression $\mu x \leq a.\ R(x, \boldsymbol{v})$ denotes the function

$$m(a, \boldsymbol{v}) = \begin{cases} \text{the minimal } x \leq a \text{ such that } R(x, \boldsymbol{v}) \text{ holds,} & \text{if } \exists x \leq a R(x, \boldsymbol{v}) \\ a+1, & \text{otherwise.} \end{cases}$$

For a set of functions K, let $\mathbf{C}(K)$ denote the closure of K and the class of elementary functions $\mathcal{E}$ under composition. Further, let $[K, \mathbf{M}]$ denote the closure of the class $K \cup \mathcal{E}$ under composition and unnested applications of bounded minimum operator, that is, the closure under composition of $K \cup \mathcal{E}$ and all functions of the form $\mu x \leq a.\ R(x, \boldsymbol{v})$, where $R(x, \boldsymbol{v}) \in \mathbf{C}(K)$. Similarly, $[K, \mathbf{BR}]$ denotes the closure of $K \cup \mathcal{E}$ under composition and unnested applications of bounded recursion schema, that is, primitive recursion bounded by some function from $\mathbf{C}(K)$. The closure of $K \cup \mathcal{E}$ under composition and (nested) bounded recursion is called the *elementary closure* of K and is denoted $\mathbf{E}(K)$. By a result of Parsons [9], $\mathbf{E}(f) = \mathbf{C}(\bar{f})$, where $\bar{f}$ denotes the function $\bar{f}(x) = \langle f(0), \ldots, f(x) \rangle$.

It is known [10] that $\mathbf{E}(K)$ coincides with the closure of $K \cup \mathcal{E}$ under composition and (nested) bounded minimum operator. It is also easy to see from [10] that $\mathbf{E}(K) = [K, \mathbf{BR}]$. On the other hand, we prove the following theorem.

Theorem 1. *There is a function $f : \mathbb{N} \to \mathbb{N}$ such that*

$$[\mathbf{C}(f), \mathbf{M}] \neq [\mathbf{C}(f), \mathbf{BR}] = \mathbf{E}(f).$$

This means that unnested bounded μ-operator is insufficient to generate the elementary closure of a function. In contrast, we also show that for any f, $[[\mathbf{C}(f), \mathbf{M}], \mathbf{M}] = \mathbf{E}(f)$.

Our main concern will be the counterpart of these results in formal arithmetic. We consider the language of first order Kalmar elementary arithmetic EA (with function symbols for all elementary functions and the only relation symbols $\leq$ and $=$) enriched by a new unary function symbol f. The set of open formulas in this language will be denoted $\mathbf{C}(f)$. The set of bounded formulas, that is, the formulas obtained from atomic ones using boolean connectives and bounded quantifiers (f may occur in bounding terms) is denoted $\Delta_0(f)$. Relativized elementary arithmetic $\mathsf{EA}(f)$ is the extension of EA in the enriched language without any additional mathematical axioms for f.

We study the comparative strength of the *least element principle* for open and for bounded formulas. For a formula $R(x)$, let L_R denote the following formula:

$$R(a) \to \exists x \leq a\, (R(x) \wedge \forall y < x\, \neg R(y)).$$

Of course, we assume that $R(x)$ may involve other parameters apart from x. $L\mathbf{C}(f)$ will denote the set of formulas L_R for all open R; $L\Delta_0(f)$ is similarly defined. By abusing the terminology, $L\mathbf{C}(f)$ will also denote the theory axiomatized over $\mathsf{EA}(f)$ by this schema. The related *induction schema* is similarly defined: I_R denotes the formula

$$R(0) \wedge \forall x \leq a\, (R(x) \to R(x+1)) \to R(a),$$

$I\mathbf{C}(f)$ denotes the set of formulas I_R for all open R, $I\Delta_0(f)$ denotes the set of formulas I_R for all bounded R.

Obviously, for any formula R, the formula $L_{\neg R}$ implies I_R, and $I_{\forall u \leq x\, \neg R(u)}$ implies L_R. This shows that the schemata $I\Delta_0(f)$ and $L\Delta_0(f)$ are equivalent over $\mathsf{EA}(f)$. Notice, however, that the formula $\forall u \leq x\, \neg R(u)$ involves bounded quantifiers. In fact, over $\mathsf{EA}(f)$ the induction schema $I\mathbf{C}(f)$ turns out to be strictly weaker than the corresponding least element principle $L\mathbf{C}(f)$. This fact is related to the question of separating the schema of arithmetical Δ_1-induction from the Σ_1-collection schema [4,6]. We give a proof of this fact in a subsequent paper.

Our goal here is to prove the following result.

Theorem 2. $L\mathbf{C}(f) \nvdash L\Delta_0(f)$.

Before starting with the proof, let us notice that the theorem above is sensitive to the presence of additional axioms for f. For example, if the additional axioms expressing that f is any of the usual fast growing functions of the Grzegorczyk hierarchy are added to $L\mathbf{C}(f)$, then the schema $L\Delta_0(f)$ will be provable. In fact, an axiom asserting the monotonicity of f and an elementary definition of the graph of f suffice [1]. As a trivial example consider the axiom $\forall x\, f(x) = 0$. In this case both theories are equivalent to the elementary arithmetic EA.

The proof of Theorem 2 goes in two relatively independent steps. The first step is a proof-theoretic reduction of the question of separation of the two axiom systems to a purely computation-theoretic question. This will be a more or less straightforward application of Herbrand's theorem for $\exists\forall$ formulas. The second step is a recursion-theoretic construction, which allows to separate the classes of computable functions resulting from the previous Herbrand analysis. This result strengthens Theorem 1 and is obtained by a slight modification of its proof.

Applications of Herbrand's theorem for $\exists\forall$ formulas have proved to be very useful in the questions of separating the systems of bounded arithmetic. Connections between the classes of functions representable in systems of bounded arithmetic and bounded query computation have also been established (see [7]). This paper shows that a similar methodology is useful in the context of subrecursion and subrecursive arithmetic. However, there are some notable differences. An essential feature of our techniques is that we deal with function oracles, whereas bounded query computation has mostly been considered for the set oracle Turing machines. This difference seems to be essential, e.g., the problem of computing the maximum of a function has little meaning for 0–1-valued functions.

3 Herbrand Analysis

Recall that $\bar{f}$ denotes the function $\bar{f}(x) = \langle f(0), \dots, f(x)\rangle$, assuming some standard elementary coding of finite sequences of numbers. Notice that the graph of $\bar{f}$ can be defined in the language of $\mathsf{EA}(f)$ by the following bounded formula:

$$\bar{f}(x) \simeq y \equiv [y \in Seq \wedge lh(y) = x + 1 \wedge \forall i \leq x\ (y)_i = f(i)]. \tag{1}$$

The following lemma is straightforward, but we shall give a detailed argument in order to see how much induction is actually used. For $n \geq 1$ let $\Pi^b_n(f)$ and $\Sigma^b_n(f)$ denote the classes of formulas obtained from $\mathbf{C}(f)$ by n alternating blocks of bounded quantifiers, starting from $\forall$ and $\exists$, respectively. For technical reasons (that will only be essential in Section 5) we assume that all the bounding terms are elementary, that is, do not involve the function symbol f.

Lemma 1. $L\Delta_0(f) \vdash \forall x \exists y \bar{f}(x) \simeq y$.

Proof. By induction on x we first prove the following statement:

$$\forall x\, [\exists u \leq x\, \forall i \leq x\ f(i) \leq f(u)].$$

The formula in square brackets (let us denote it $\varphi(x)$) is bounded and, in fact, belongs to the class $\Sigma^b_2(f)$. Moreover, the induction step $\forall x\ (\varphi(x) \to \varphi(x+1))$

is obviously provable in $\mathsf{EA}(f)$. Hence, this argument can be done in $I\Sigma_2^b(f)$. Further, taking $y = f(u)$ we can conclude within that theory and within $L\Delta_0(f)$ that $y = \max_{i \le x} f(i)$ exists.

As a separate argument we now prove the following statement:

$$\forall i \le x\, f(i) \le y \to \exists v \le b(x,y)\, \bar{f}(x) \simeq v,$$

for a suitable elementary term $b(x,y)$ that bounds the code of a sequence given its length and a bound to its elements. This statement follows by the least element principle, for $\bar{f}(x)$ is the least element $v \le b(x,y)$ such that $\forall i \le x\, f(i) \le (v)_i$. (This holds under the usual conventions on the coding of sequences.) Thus, this part of the argument is formalizable in $L\Pi_1^b(f)$.

In Section 5 we improve on the first part of the above argument and show that the statement $\forall x \exists y \bar{f}(x) \simeq y$ is provable in $L\Pi_1^b(f)$. In fact, $L\Pi_1^b(f)$ is shown to be equivalent to $L\Delta_0(f)$. Our main aim in this paper is to show that the totality of $\bar{f}$ is *not* provable in $L\mathbf{C}(f)$.

The following easy lemma states that symbols for all functions of the class $[\mathbf{C}(f), \mathbf{M}]$ can be introduced in a definitional extension of the theory $L\mathbf{C}(f)$. First, for any formula $R(x) \in \mathbf{C}(f)$ we introduce a symbol m_R for the function $\mu x \le a.\ R(x)$ (here and below we ignore possible additional parameters in R and m_R). Second, let M_R denote the following formula:

$$R(a) \to (R(m_R(a)) \wedge \forall y < m_R(a)\, \neg R(y)).$$

Let $L\mathbf{C}(f)^\circ$ denote the extension of $\mathsf{EA}(f)$ by symbols $m_R(x)$ for all $R \in \mathbf{C}(f)$ and the corresponding axioms M_R.

Lemma 2. *$L\mathbf{C}(f)^\circ$ is a conservative definitional extension of $L\mathbf{C}(f)$.*

Proof. The two things to notice is that axioms M_R logically imply the corresponding axioms L_R (because M_R implies $R(a) \to m_R(a) \le a$), and that L_R also implies that the minimum is unique.

Notice that $L\mathbf{C}(f)^\circ$ has a purely universal axiomatization. A version of Herbrand's theorem for $\exists\forall$ formulas suitable for our present purposes reads as follows [3].

Lemma 3. *Let T be a theory axiomatized by a set of purely universal formulas, and let $\varphi(x,u,y)$ be an open formula such that $T \vdash \exists y \forall u \varphi(x,u,y)$. Then there are terms $t_0(x)$, $t_1(x,u_0)$, $\ldots$, $t_n(x,u_0,\ldots,u_{n-1})$ such that*

$$T \vdash \varphi(x,t_0(x),u_0) \vee \varphi(x,t_1(x,u_0),u_1) \vee \ldots \vee \varphi(x,t_n(x,u_0,\ldots,u_{n-1}),u_n).$$

We now apply this theorem to the formula $\exists y \bar{f}(x) \simeq y$. In other words, we take T to be $L\mathbf{C}(f)^\circ$ and $\varphi(x,u,y)$ to be $(u \le x \to f(u) = (y)_u)$ and obtain the following corollary.

Corollary 1. *If* $L\mathbf{C}(f) \vdash \forall x \exists y\, \bar{f}(x) \simeq y$, *then there are terms* $t_0(x)$, $t_1(x, u_0)$, $\dots$, $t_n(x, u_0, \dots, u_{n-1})$ *in* $L\mathbf{C}(f)^\circ$ *such that the following disjunction holds for any* f *and all* $x, u_0, \dots, u_n \in \mathbb{N}$ *(and is, in fact, provable in* $L\mathbf{C}(f)^\circ$*):*

$$\begin{aligned} &(u_0 \le x \to f(u_0) = (t_0(x))_{u_0}) \quad \vee \\ &(u_1 \le x \to f(u_1) = (t_1(x, u_0))_{u_1}) \quad \vee \quad \cdots \\ &(u_n \le x \to f(u_n) = (t_n(x, u_0, \dots, u_{n-1}))_{u_n}). \end{aligned} \tag{2}$$

In order to compute $\bar{f}(a)$ we do not need to know any values of f outside the interval $[0, a]$. This idea is captured in the following definition of *a-reduced* $[\mathbf{C}(f), \mathbf{M}]$-term t, where a is a distinguished free variable of t.

A term t is *a-reduced*, if every term s such that $f(s)$ is a subterm of t graphically has the form $s = \min(s_0, a)$, for some term s_0. (We also consider the subterms $f(s)$ occurring inside the μ-operators $\mu i \le t_0.\, R(i)$, that is, in t_0 and R.)

The following corollary allows us to restrict our attention below to the reduced terms.

Corollary 2. *In the formulation of Corollary 1 we can assume all the terms* $t_0(x)$, $t_1(x, u_0)$, $\dots$, $t_n(x, u_0, \dots, u_{n-1})$ *to be x-reduced.*

Proof. Let the terms t_i for $i < n$ satisfy the conclusion of Corollary 1. Replace all subterms of the form $f(s)$ occurring in one of the terms t_i by the terms $f(\min(s, x))$. (We also make the substitution inside the subterms of the form $\mu z \le t.\, R(z)$, that is, in t and R.) Each term of the resulting sequence $t'_0(x)$, $t'_1(x, u_0)$, $\dots$, $t'_n(x, u_0, \dots, u_{n-1})$ is x-reduced. We claim that this sequence also satisfies the conclusion of Corollary 1.

Indeed, the sequence of terms $t_0(x)$, $t_1(x, u_0)$, $\dots$, $t_n(x, u_0, \dots, u_{n-1})$ is supposed to satisfy the disjunction (2) for any function f. Let the function f' coincide with f on the interval $[0, x]$ and let $f'(y) = f(x)$ for $y > x$. Then, obviously, $\bar{f}'(x) = \bar{f}(x)$. Moreover, the value of $f'(s)$ for any s coincides with $f(\min(s, x))$. Hence the value of any term t'_i for f coincides with the value of t_i for f' (easy induction on the build-up of t_i). It follows that the sequence t'_i satisfies the conclusion of the corollary.

The next section will be devoted to the proof of the fact that for a suitably chosen f a sequence of (x-reduced) terms satisfying (2) cannot exist. This will be preceded by a proof of Theorem 1.

4 Bounded Query μ-Programs

Every term $t \in \mathbf{C}(f)$ can be considered as a program with an oracle for a function f, which may only ask a bounded number of oracle queries (this number is bounded by the number of occurrences of the symbol f in t and does not depend on the input of the program). The running time is then bounded by an elementary function of the size of the input together with all the oracle answers.

For technical reasons it will be convenient for us to deal with terms from $\mathbf{C}(f)$ in the format of programs (this fixes a specific order of subterms of a given term).

A *simple program* R is a sequence of assignments of the following kind:

$$\begin{aligned} y_0 &:= f(t_0(\boldsymbol{x})); \\ y_1 &:= f(t_1(\boldsymbol{x}, y_0)); \\ &\dots \\ y_n &:= f(t_n(\boldsymbol{x}, y_0, \dots, y_{n-1})) \\ R &:= R(\boldsymbol{x}, y_0, \dots, y_n). \end{aligned}$$

Here $t_0, \dots, t_n$ are elementary terms, $\boldsymbol{x}$ are the input variables, and R is an elementary output predicate. (We shall only consider simple programs that compute predicates, that is, return **true** or **false**.) The number n is called the *length* of the program R, and y_i are the *inner variables* of R.

Execution of a program for a given oracle function $f : \mathbb{N} \to \mathbb{N}$ is defined in the obvious way. If f is only a partial function, then the result of the execution of a program may not be defined (this happens, if the value of some term t_i computed during the execution of the program does not belong to $dom(f)$). A program R is *a-reduced*, if all the terms t_i of R have the form $\min(t'_i, a)$, for some elementary term t'_i, where a is a fixed input variable.

Clearly, any predicate $R \in \mathbf{C}(f)$ can be rewritten as a simple program. Moreover, an a-reduced predicate is represented as an a-reduced program.

Suppose R is a simple program of length n, and let the values of all input variables be fixed. Then the output of R can be considered as a boolean-valued function of the oracle answers obtained during its execution. However, there may be some dependencies between these answers. We will show that, under certain conditions on the values of f, these dependencies can be simplified. This is formally expressed by Lemma 4 below. The idea behind it is that by a suitable restriction of the (infinite) range of f, and hence of the range of possible oracle answers, one can insure that the output of any given simple program does not actually depend on the those answers. This idea already has a Ramsey-type flavour and, indeed, Ramsey theorem plays a central role in the argument. Technical details follow.

Let $f : D \to \mathbb{N}$ be a partial function with a finite domain D, and $Y \subseteq \mathbb{N}$ be an infinite set. Let $\prec$ be a linear ordering on $[0, n]$, it will be interpreted as a certain preference ordering of the inner variables $y_0, \dots, y_n$ of R. A sequence of numbers $\boldsymbol{c} = (c_0, \dots, c_n)$ from the interval $[0, a]$ and a sequence $(y_0, \dots, y_n)$ are *coherent* if they satisfy the following conditions:

f-constraints: $\forall i \le n (c_i \in dom(f) \Rightarrow y_i = f(c_i))$;
$\boldsymbol{c}$-constraints: $\forall i, j \le n (c_i = c_j \Rightarrow y_i = y_j)$.

If some of the following additional assumptions are satisfied, we speak about Y-, $\prec$- or $(Y, \prec)$-coherence.

Y-constraints: $\forall i \le n (c_i \notin dom(f) \Rightarrow y_i \in Y)$;

$\prec$-constraints: If $i, j \leq n$ are such that $c_i, c_j \notin dom(f)$ and $(\forall k < i\ c_k \neq c_i)\ \&\ (\forall k < j\ c_k \neq c_j)$, then $i \prec j$ implies $y_i < y_j$.

Clearly, any coherent pair of sequences represents an extension f' of the given function f to the domain $D \cup \{c_0, \ldots, c_n\}$. Y- and $\prec$-constraints tell us that the values of this function are chosen in a pre-specified order from a given set Y. Given a sequence $\boldsymbol{c}$, one can define the subsequence of *essential* elements of $\boldsymbol{c}$ as follows. c_0 is essential if $c_0 \notin dom(f)$. c_{k+1} is essential if $c_{k+1} \notin dom(f)$ and for no $i \leq k$ do we have $c_i = c_{k+1}$. Note that the sequence of essential elements of $\boldsymbol{c}$ only depends on f. $\prec$-constraints state that the order of the values of f' on essential arguments agrees with the preference order $\prec$.

Notice the following obvious monotonicity property: if a pair of sequences is Y-coherent, then it is also Y'-coherent for any set $Y' \supseteq Y$.

Let a reduced program R and the values of the input variables of R be fixed. Let a be the value of the distinguished variable of R.

Lemma 4. *For any f, Y and $\prec$ as above, one can find an infinite subset $Y' \subseteq Y$ and a sequence $\boldsymbol{c} = (c_0, \ldots, c_n)$ such that any sequence $\boldsymbol{y} = (y_0, \ldots, y_n)$ $(Y', \prec)$-coherent with $\boldsymbol{c}$ yields an extension f' of f for which the program R terminates. Moreover, the values of the terms t_i computed during the execution of R on f' equal the constants c_i, and the output value of R is constant (either R is true for all such sequences $\boldsymbol{y}$, or false for all of them).*

Proof. We successively construct infinite sets $Y_0 \supseteq Y_1 \supseteq \ldots \supseteq Y_n$ and the corresponding numbers $(c_0, \ldots, c_n)$, so that the coherent values of f' taken from set Y_i on arguments $(c_0, \ldots, c_i)$ allow to execute the program R up to line i.

Since the values of the input variables are fixed, we simply define c_0 to be the value of t_0 and let Y_0 equal Y. Assume Y_k and $\boldsymbol{c} = (c_0, \ldots, c_k)$ are already constructed. Consider the term $t_{k+1}(y_0, \ldots, y_k)$. First, substitute in t_{k+1} the constants $f(c_i)$ for all variables y_i such that $c_i \in dom(f)$. Second, substitute for y_j the variable y_i, if c_i is essential, $c_j = c_i$ and $i < j$. Call the resulting term t'.

Obviously, the value of t_{k+1} on any sequence $\boldsymbol{y}$ coherent with $\boldsymbol{c}$ coincides with the value of t' on the subsequence of $\boldsymbol{y}$ corresponding to essential elements of $\boldsymbol{c}$.

The ordering $\prec$ of the essential elements of $\boldsymbol{c}$ allows us to naturally associate with t' a function

$$\varphi_{t'} : [Y_k]^p \to [0, a],$$

where p is the number of arguments of t' (and coincides with the number of essential elements of $\boldsymbol{c}$), and $[X]^p$ denotes the set of all p-element subsets of X.

The function $\varphi_{t'}$ is defined as follows. Let $e : [1, p] \to [0, k]$ enumerate the essential elements $(c_{e1}, \ldots, c_{ep})$ of $\boldsymbol{c}$. Let π be the unique permutation of $[1, p]$ such that

$$\pi i < \pi j \iff ei \prec ej,$$

that is, $e \circ \pi^{-1}$ enumerates the essential elements in the order induced by $\prec$. We define

$$t''(v_1, \ldots, v_p) = t'(v_{\pi 1}, \ldots, v_{\pi p}).$$

Any p-element subset $V \subseteq Y_k$ can be uniquely ordered into an increasing sequence $\boldsymbol{v}$ of elements of Y_k. We define $\varphi_{t'}(V) = t''(\boldsymbol{v})$.

By the infinite Ramsey theorem there is an infinite subset $Y_{k+1} \subseteq Y_k$ such that $\varphi_{t'}$ is constant on $[Y_{k+1}]^p$. Let c_{k+1} be the constant value of $\varphi_{t'}$ on $[Y_{k+1}]^p$. We show that $t_{k+1}(y_0, \dots, y_k) = c_{k+1}$ for all sequences $(y_0, \dots, y_k)$ which are $(Y_{k+1}, \prec)$-coherent with $\boldsymbol{c}$.

Indeed, let $(y_0, \dots, y_k)$ be $(Y_{k+1}, \prec)$-coherent with $\boldsymbol{c}$. Then

$$\begin{aligned} t_{k+1}(y_0, \dots, y_k) &= t'(y_{e0}, \dots, y_{ep}) \\ &= t''(y_{e\pi^{-1}0}, \dots, y_{e\pi^{-1}p}). \end{aligned}$$

By $\prec$-coherence we have

$$y_{e\pi^{-1}i} < y_{e\pi^{-1}j} \iff e\pi^{-1}i \prec e\pi^{-1}j \iff i < j.$$

This means that $(y_{e\pi^{-1}0}, \dots, y_{e\pi^{-1}p})$ is an increasing sequence, hence by the choice of Y_{k+1}

$$t''(y_{e\pi^{-1}0}, \dots, y_{e\pi^{-1}p}) = c_{k+1}.$$

Thus, we have constructed the required sequence $(c_0, \dots, c_n)$ and a set Y_n, which allow to execute R up to line n.

To ensure that the output value of R is constant we apply Ramsey theorem in the same fashion to the predicate $R(y_0, \dots, y_n)$ considered as a boolean valued function of $\boldsymbol{y} = (y_0, \dots, y_n)$. Then we obtain a subset $Y' \subseteq Y_n$ such that R is constant on any sequence $\boldsymbol{y}$ which is $(Y', \prec)$-coherent with $(c_0, \dots, c_n)$.

Lemma 5. *The set Y' can be chosen uniformly in $\prec$, that is, given f, Y and the values of input variables of R, one can point out a subset $Y' \subseteq Y$ that satisfies the conclusion of Lemma 4 for any preference ordering $\prec$.*

Proof. This follows from the fact that there are only finitely many possible orderings of $[0, n]$. Thus, we can enumerate all such orderings $\prec_0, \dots, \prec_s$. Applying Lemma 4 $s + 1$ times we successively construct infinite sets $Y_0 \supseteq Y_1 \supseteq \dots \supseteq Y_s = Y'$. Then for any ordering $\prec_i$ one can point out a sequence $\boldsymbol{c}_i$ such that any $(Y', \prec_i)$-coherent sequence $\boldsymbol{y}$ is also $(Y_i, \prec_i)$-coherent with $\boldsymbol{c}_i$ (by the monotonicity property) and thus satisfies the conclusions of the previous lemma.

Next we define μ-programs. A *μ-operator* is an assignment of the following form:

$$y := \mu i \le z.\ R(i, \boldsymbol{x}), \qquad (*)$$

where R is a simple program with the input variables as shown. A μ-operator is a-reduced, if the simple program R is. If the variables $\boldsymbol{x}$ and z are already evaluated, then the output y of the μ-operator is defined, as usual, to be the minimal i for which $R(i, \boldsymbol{x})$ holds, if such an $i \le z$ exists, and y equals $z + 1$, otherwise. A *μ-program* P is a sequence of simple assignments of the form $y := t(\boldsymbol{u})$ ($t(\boldsymbol{u})$ is an elementary term), $y := f(x)$ and of μ-operators $(*)$, which satisfies the following natural variable restrictions:

1) any inner variable is assigned its value in P only once;

2) for any occurrence of any of the above kinds of operators, $\boldsymbol{u}$, $\boldsymbol{x}$, z and x are either input variables or inner variables introduced earlier than y.

We will only consider μ-programs that are reduced in the sense that all the μ-operators are a-reduced, for a fixed input variable a. Obviously, such programs suffice to represent any a-reduced $[\mathbf{C}(f), \mathbf{M}]$-term.

Notice that there can be various possibilities of implementing μ-operator on different machines. A simple deterministic strategy would be to evaluate successively $R(0), R(1), \ldots, R(z)$ until the minimal i for which $R(i)$ evaluates to **true** is found. However, there is a possibility of implementing the μ-operator on a parallel machine, where we independently evaluate $R(0), R(1), \ldots, R(z)$ on different processors and then run through their single-bit outputs to find the minimal i for which $R(i)$ holds. Notice that these independent processors can be bounded in that each one of them may only ask a uniformly bounded number of oracle queries, whereas the simple deterministic strategy requires a potentially unbounded number of oracle queries. Another important restriction is that each of the processors only returns a single bit, that is, it cannot exchange too much information with other processors. This nondeterministic picture is useful for the understanding of our formal construction below.

Suppose we are given a μ-operator and the values of its input variables. Further, assume that an infinite set Y and a finite function f are given. We shall describe a procedure that allows to evaluate the μ-operator under some conditions on the choice of additional values of f . Simultaneously we extend f (adding no more than $n+1$ new elements to its domain), and go from Y to an (infinite) subset $Y' \subseteq Y$.

First, consider the simple program $R(x)$ for the input value $x = 0$. By Lemma 5 we obtain an infinite subset $Y_0 \subseteq Y$ such that any ordering $\prec$ of the inner variables of R uniquely determines a sequence $\boldsymbol{c}$ together with an output value of $R(0)$. If for one of the finitely many orderings $\prec$ the corresponding value equals **true**, then we evaluate μ to 0, set $Y' = Y_0 \setminus [0, \max(f)]$ and extend f to the set $\{c_0, \ldots, c_n\}$ by choosing a suitable tuple $\boldsymbol{y}$ $(Y', \prec)$-coherent with $\boldsymbol{c}$. (Here $\max(f)$ is the maximal value of f, recall that $dom(f)$ is finite.)

Otherwise (if all preference orderings yield the output value **false**), then we consider the input value $x = 1$ for R and construct the corresponding set $Y_1 \subseteq Y_0$. Again, μ is evaluated to 1, if at least one of the orderings yields value **true**, and to **false**, otherwise.

Proceeding in this way, we construct subsets $Y_0 \supseteq Y_1 \supseteq \ldots \supseteq Y_k$ for $k \leq z$. At the end, μ is evaluated to k, if a k is found for which $R(k)$ evaluates to **true** under this procedure, or μ is evaluated to $z+1$, otherwise. In the latter case we define $Y' = Y_z$ and $f' = f$.

Notice that as the result of running this procedure we evaluate the μ-operator and simultaneously define an infinite subset $Y' \subseteq Y$ and a function f' extending f. Obviously, this extension procedure preserves the injectivity of f.

Notice the following important property of this construction.

Lemma 6. *Let the evaluation procedure yield Y' and f', and let g be any total injective function extending f' such that $g(\mathbb{N} \setminus dom(f')) \subseteq Y'$. Then the value of the μ-operator computed for the function g coincides with the result of the above evaluation procedure.*

Proof. Consider the evaluation procedure. If μ was evaluated to 0, this means that $R(0)$ evaluated to **true** under some preference ordering. Yet, the essential arguments and values in that case are fixed, that is, $R(0)$ evaluates to **true** under f', and hence under g.

If μ was evaluated to 1, then $R(1)$ evaluates to **true** under f' and g for similar reasons. Let us execute the simple program $R(0)$ for g. Notice that by the construction of f', $f'(\mathbb{N} \setminus dom(f)) \subseteq Y'$. This means that all values assigned to the inner variables of R during the execution of $R(0)$ on g are elements of Y'. Moreover, the sequence of these values $\boldsymbol{y}$ and the corresponding arguments $\boldsymbol{c}$ of g satisfy f-, $\boldsymbol{c}$- and Y'-constraints.

Further, by our assumption g has different values on different arguments. Hence, the values y_i (for essential arguments c_i) are strictly linearly ordered. Let $\prec$ be any extension of the induced linear ordering to $[0, n]$ (it does not matter, how the inessential arguments are ordered). Then $\boldsymbol{y}$ and $\boldsymbol{c}$ will be $(Y', \prec)$-coherent. By the construction of Y', the value **false** of $R(0)$ is then uniquely determined for any $(Y', \prec)$-coherent sequences, that is, this value coincides with the computed value of $R(0)$ on g.

The argument showing that the values of $R(j)$ for $j \leq i$ are preserved, for the case that the μ-operator was evaluated to $i > 1$, is similar.

Now we show that the function $\bar{f}$ cannot be computed by a μ-program.

Lemma 7. *There is a total function $f : \mathbb{N} \to \mathbb{N}$ such that for any μ-program $P(x)$ there is a number a such that $P(a) \neq \bar{f}(a)$.*

Proof. First of all, by Corollary 2 it is sufficient to construct an f that falsifies any x-reduced μ-program (actually, we apply a particular case of this corollary for a sequence consisting of a single term t_0).

We enumerate all such programs $P_0, \ldots, P_\alpha, \ldots$ and construct f in such a way that by the stage α the function f is defined on a finite domain D_α and falsifies the programs $P_0, \ldots, P_{\alpha-1}$. We set $D_0 = \emptyset$.

Consider the program $P_\alpha(x)$. Let q be the total number of occurrences of the symbol f in P_α, where we also count the occurrences of f inside the μ-operators. Choose a number a so large that there are more than q elements in the set $[0, a] \setminus D_\alpha$. Evaluate $P_\alpha(a)$ along with constructing an infinite set $Y \subseteq \mathbb{N}$ and an extension of f according to the following rules.

Initialize $Y = \{x \in \mathbb{N} \mid x > \max(f)\}$ ($dom(f) = D_\alpha$ is finite).

Simple assignments of the form $y := t(\boldsymbol{z})$, where t is an elementary term, are evaluated in the usual way, f and Y are not changed.

An assignment of the form $y := f(z)$ is treated as follows. If $z \in dom(f)$, then return the corresponding value of f, do not change f or Y. If $z \notin dom(f)$, then set $f(z) = \min(Y)$, add z to D_α and delete $\min(Y)$ from Y.

An assignment of the form $y := \mu i \leq z.\ R(i)$ is treated in accordance with the evaluation procedure described above (f and Y are changed as specified in that procedure).

Clearly, the program $P_\alpha(a)$ will be eventually evaluated, and f will be defined at no more than q new points of the interval $[0, a]$. Hence, there will be a point of that interval, where f is not yet defined. Define the value of f at such points to be any sequence of different elements of Y, each of which is greater than the output of the program $P_\alpha(a)$. (This is sufficient for our purposes, for $\max_{i\leq a} f(i) < \bar{f}(a)$.) End of stage α of the construction of f.

Now we have to show that f as constructed really falsifies the program P_α on $[0, a]$. It suffices to show that the computation of P_α with the oracle function f yields the same result as our evaluation procedure. We prove this by induction on the length of P_α. Simple assignments obviously yield the same results.

Consider the crucial case of a μ-operator. Let f_1 and Y_1 be constructed by the evaluation procedure for this μ-operator. Obviously, f restricted to $[0, a]$ is an injective function extending f_1. Besides, by the construction only the values from Y_1 are added to f_1 at later stages of the evaluation procedure for the given μ-program. Therefore, Lemma 6 can be applied, which completes the induction step and the proof of the lemma.

Obviously, the function $\bar{f}$ belongs to $[\mathbf{C}(f), \mathbf{BR}]$ (see [10]). (It follows that the class $[\mathbf{C}(f), \mathbf{BR}]$ coincides with the elementary closure of f.) On the other hand, by Lemma 7 there is an f such that $\bar{f}$ is not in $[\mathbf{C}(f), \mathbf{M}]$. This completes the proof of Theorem 1.

Theorem 1 contrasts with the fact that doubly nested μ-operator suffices to generate the elementary closure.

Lemma 8. *For any function f,*

$$[[\mathbf{C}(f), \mathbf{M}], \mathbf{M}] = [\mathbf{C}(f), \mathbf{BR}] = \mathbf{E}(f).$$

Proof. If $R \in \mathbf{C}(f)$, then the predicate $\forall i \leq x\ R(i)$ belongs to $[\mathbf{C}(f), \mathbf{M}]$, for

$$\forall i \leq x\ R(i) \leftrightarrow x + 1 = \mu i \leq x.\neg R(i).$$

Defining $t(x) = \mu z \leq x.\ \forall u \leq x\ f(u) \leq f(z)$ yields

$$\max_{i\leq x} f(i) = f(t(x)) \in [[\mathbf{C}(f), \mathbf{M}], \mathbf{M}].$$

On the other hand, the relation $\bar{f}(x) \simeq y$ (see (1)) belongs to $[\mathbf{C}(f), \mathbf{M}]$, and there is an elementary function $g(x, y)$ such that for all x,

$$\bar{f}(x) = \mu u \leq g(x, \max_{i\leq x} f(i)).\bar{f}(x) \simeq u.$$

Hence, $\bar{f} \in [[\mathbf{C}(f), \mathbf{M}], \mathbf{M}]$ as a composition of two functions from this class. But $\mathbf{C}(\bar{f}) = \mathbf{E}(f)$.

In order to prove Theorem 2 we need a somewhat sharper separation result. The sequence of terms $t_0(x)$, $t_1(x, u_0)$, ... , $t_n(x, u_0, \dots, u_{n-1})$ that appears in Lemma 3 and Corollary 1 can be considered as a kind of teacher-student game [7]: the student first tries to compute $\bar{f}(x)$ with a program $t_0(x)$. If the answer is correct, the student wins. If the answer is incorrect, the teacher has to show him/her a counterexample u_0 demonstrating that t_0 fails. The student then has the right to come up with a better solution $t_1(x, u_0)$ that may depend on the teacher's example u_0, and so on. The game has boundedly many rounds.

We have to show that the student who is only able to compute bounded query μ-programs cannot, in general, interactively compute $\bar{f}$ in the above sense. We will essentially use the fact that the teacher can only help the student boundedly many times. A sequence P of (reduced) μ-programs $P_0(x)$, $P_1(x, u_0)$, ... , $P_n(x, u_0, \dots, u_{n-1})$ will be called an *interactive μ-program*.

Lemma 9. *There is a function $f : \mathbb{N} \to \mathbb{N}$ such that for any interactive μ-program P there is a number a and a sequence $u_0, \dots, u_n$ of elements of $[0, a]$ such that the student looses the game $P(a)$ with the teacher's counterexamples $u_0, \dots, u_n$:*

$$\begin{aligned} &f(u_0) \neq (P_0(a))_{u_0} \quad \wedge \\ &f(u_1) \neq (P_1(a, u_0))_{u_1} \quad \wedge \quad \cdots \\ &f(u_n) \neq (P_n(a, u_0, \dots, u_{n-1}))_{u_n}. \end{aligned}$$

Proof. The argument is very similar to the one for Lemma 7. We only have to take care that the interval on which the game is being played has a place for all the teacher's answers.

Enumerate all possible interactive μ-programs. Let P_α be the program to be falsified at stage α, and let f be already defined on a finite domain D_α. Let q be greater than the total number of occurrences of the symbol f in P plus n. Fix an a such that $[0, a]$ has more than q elements outside D_α.

Evaluate P_0 as described in Lemma 7. Find the minimal u_0 such that $u_0 \notin dom(f)$, define $f(u_0)$ to be the minimal element of Y bigger than the output of P_0 and throw out all elements of Y smaller than $f(u_0)$. Proceed in the same way with the evaluation of $P_1(a, u_0)$.

Obviously, after P_α is evaluated, less than q many new elements are added to the domain of f. Hence, all $u_0, \dots, u_n$ are successively defined and falsify P_α.

This completes the proof of Theorem 2.

5 Further Remarks

In this section we consider the strength of the induction and the least element principles for the classes of formulas between $\mathbf{C}(f)$ and $\Delta_0(f)$. First, we state some equivalences. Deductive equivalence of theories will be denoted $\equiv$.

Lemma 10. $I\Sigma_n^b(f) \equiv I\Pi_n^b(f)$.

Proof. This is proved by the usual trick of Parsons: induction for a formula $\varphi(x)$ can be reduced to the induction for $\neg\varphi(a \dot{-} x)$, where a is a free parameter. The reduction can be carried out in $\mathsf{EA}(f)$.

Lemma 11. $I\Pi^b_n(f) \equiv L\Sigma^b_n(f)$.

Proof. This follows by the usual proof of the least element principle by induction: $I_{\forall u \le x\, \neg R(u)}$ implies L_R, and the class of $\Pi^b_n(f)$-formulas is closed under bounded universal quantifiers.

Notice that the dual form of this lemma for $n = 1$ is wrong. This can be seen from the next lemma.

Lemma 12. *The following theories are equivalent:*

1. $L\Sigma^b_1(f) \equiv L\mathbf{C}(f)$;
2. $L\Sigma^b_{n+1}(f) \equiv L\Pi^b_n(f)$, *for* $n \ge 1$.

Proof. We only prove Part 1. The proof of Part 2 is similar.

We are to derive inside $L\mathbf{C}(f)$ the formula L_φ, where $\varphi(a)$ has the form $\exists u \le t(a)\, \psi(u, a)$ (possibly with some additional parameters). Here $t(a)$ is an elementary term.

Consider the set of pairs $S = \{\langle x, u\rangle \mid u \le t(x)\}$ and order it as follows:

$$\langle x_1, u_1\rangle \prec \langle x_2, u_2\rangle \iff (x_1 < x_2 \vee (x_1 = x_2 \wedge u_1 < u_2)).$$

Obviously, there is an elementary isomorphism between $(S, \prec)$ and $(\mathbb{N}, <)$, so within $L\mathbf{C}(f)$ one can prove the least element principle for open formulas for $(S, \prec)$.

In order to prove the formula L_φ we formalize the following argument within $L\mathbf{C}(f)$. Assume $\exists u \le t(a)\, \psi(u, a)$, then for some u satisfying $\psi(u, a)$ the pair $\langle a, u\rangle$ is in S. Applying the least element principle for $(S, \prec)$ to the $\mathbf{C}(f)$-formula $\psi_0(z) = \psi((z)_1, (z)_0)$ we obtain a pair $\langle x, v\rangle \preceq \langle a, u\rangle$ such that

$$\psi(v, x) \wedge \forall y, w\, (\langle y, w\rangle \prec \langle x, v\rangle \to \neg\psi(w, y)).$$

We claim that x is as required. Obviously, $x \le a$ and $\exists v \le x\, \psi(v, x)$. On the other hand, for all $y < x,\ w \le t(y)$ there holds $\neg\psi(w, y)$, because all such pairs $\langle y, w\rangle$ belong to S and preceed $\langle x, v\rangle$ in the sense of $\prec$.

Corollary 3. *The following theories are equivalent:*

1. $L\mathbf{C}(f) \equiv L\Sigma^b_1(f) \equiv I\Sigma^b_1(f) \equiv I\Pi^b_1(f)$,
2. $L\Pi^b_1(f) \equiv L\Sigma^b_2(f) \equiv I\Sigma^b_2(f) \equiv I\Pi^b_2(f)$.

Let $\mathsf{EA}(\bar{f})$ denote the theory formulated in the language of EA with an extra function symbol $\bar{f}$. In addition to the axioms of EA it has the following two axioms:

1. $\forall x\,(\bar{f}(x) \in Seq \wedge lh(\bar{f}(x)) = x + 1)$,
2. $\forall x, y\,(x \leq y \rightarrow \bar{f}(x) = (\bar{f}(y) \upharpoonright x))$.

Here $(z \upharpoonright x)$ denotes the natural elementary function selecting the initial segment of length $x+1$ of a sequence coded by z. Further, let $I\mathbf{C}(\bar{f})$ denote the extension of $\mathsf{EA}(\bar{f})$ by the schema of induction for open formulas.

The following lemma is established in [1]. Its proof can be obtained by formalization of the equivalence $\mathbf{C}(\bar{f}) = \mathbf{E}(f)$.

Lemma 13. *$I\mathbf{C}(\bar{f})$ contains $L\Delta_0(\bar{f})$ and $L\Delta_0(f)$, under the interpretation of $f(x)$ as the term $(\bar{f}(x))_x$.*

Notice that bounding terms occurring in the instances of $L\Delta_0(\bar{f})$ may actually involve $\bar{f}$.

As it is shown in [1], $L\Delta_0(f)$ also naturally interprets $I\mathbf{C}(\bar{f})$, see equation (1). The following lemma is an improvement of this.

Lemma 14. *Under the natural interpretation of $\bar{f}$, $L\Pi^b_1(f)$ contains $L\mathbf{C}(\bar{f})$.*

Proof. We first recall that by the proof of Lemma 1

$$I\Sigma^b_2(f) \vdash \forall x \exists y \bar{f}(x) \simeq y.$$

Hence, the same formula is provable in $L\Pi^b_1(f)$.

Second, we use an argument similar to the proof of Proposition 5.11 from [1] (which, in turn, derives from [5]). $L\mathbf{C}(\bar{f})$ can be reduced to the least element principle for positive Σ^b_1-formulas in the graph of $\bar{f}$, more precisely, the formulas built up from elementary ones and $\bar{f}(x) = y$ using $\wedge$ and bounded existential quantifiers. This relies on the monotonicity of $\bar{f}$. The natural translation of the formula $\bar{f}(x) = y$ belongs to $\Pi^b_1(f)$, hence $L\mathbf{C}(\bar{f})$ is interpretable in $L\Sigma^b_2(f)$, which by Corollary 3 is equivalent to $L\Pi^b_1(f)$.

Thus, the hierarchy of least element schemata in our setting collapses above the level of $\Pi^b_1(f)$.

Corollary 4. $L\Pi^b_1(f) \equiv L\Delta_0(f)$.

Conclusion: The theories considered so far fall into four distinct classes:

$$\mathsf{EA}(f) \subset I\mathbf{C}(f) \subset L\mathbf{C}(f) \subset L\Pi^b_1(f) \equiv L\Delta_0(f).$$

Main theorem of this paper shows that $L\mathbf{C}(f)$ and $L\Delta_0(f)$ are really distinct. It is also possible to separate $I\mathbf{C}(f)$ from $L\mathbf{C}(f)$ and from $\mathsf{EA}(f)$. These separation results are tightly related to the question of positioning the schema of arithmetical Δ_1-induction in the hierarchy of subsystems of Peano arithmetic. The proofs of the latter two results will be given in a subsequent paper.

References

1. L.D. Beklemishev. Induction rules, reflection principles, and provably recursive functions. *Annals of Pure and Applied Logic*, 85:193–242, 1997.
2. L.D. Beklemishev. A proof-theoretic analysis of collection. *Archive for Mathematical Logic*, 37:275–296, 1998.
3. S.R. Buss. Introduction to Proof Theory. In S.R. Buss, editor, *Handbook of Proof Theory*, pages 1–78. Elsevier, North-Holland, Amsterdam, 1998.
4. P. Clote and J. Krajíček. Open problems. In P. Clote and J. Krajíček, editors, *Arithmetic, Proof Theory, and Computational Complexity*, pages 1–19. Oxford University Press, Oxford, 1993.
5. H. Gaifman and C. Dimitracopoulos. Fragments of Peano's arithmetic and the MDRP theorem. In *Logic and algorithmic (Zurich, 1980), (Monograph. Enseign. Math., 30)*, pages 187–206. Genève, University of Genève, 1982.
6. P. Hájek and P. Pudlák. *Metamathematics of First Order Arithmetic*. Springer-Verlag, Berlin, Heidelberg, New-York, 1993.
7. J. Krajíček. *Bounded arithmetic, Propositional logic, and Complexity theory*. Cambridge University Press, Cambridge, 1995.
8. L. Kristiansen. Fragments of Peano arithmetic and subrecursive degrees. Manuscript, 1998.
9. C. Parsons. Hierarchies of primitive recursive functions. *Zeitschrift f. math. Logik und Grundlagen d. Math.*, 14(4):357–376, 1968.
10. H.E. Rose. *Subrecursion: Functions and Hierarchies*. Clarendon Press, Oxford, 1984.

A Universal Innocent Game Model for the Böhm Tree Lambda Theory

Andrew D. Ker, Hanno Nickau, and C.-H. Luke Ong

Computing Laboratory, Parks Road, Oxford OX1 3QD, UK,
{Andrew.Ker, Hanno.Nickau, Luke.Ong}@comlab.ox.ac.uk.

Abstract. We present a game model of the untyped λ-calculus, with equational theory equal to the Böhm tree λ-theory $\mathcal{B}$, which is universal (i.e. every element of the model is definable by some term). This answers a question of Di Gianantonio, Franco and Honsell. We build on our earlier work, which uses the methods of innocent game semantics to develop a universal model inducing the maximal consistent sensible theory $\mathcal{H}^*$. To our knowledge these are the first syntax-independent universal models of the untyped λ-calculus.

1 Introduction

We aim to construct a *universal* model (i.e. every element of the model is the denotation of some term) of the pure untyped λ-calculus which induces the Böhm tree λ-theory $\mathcal{B}$, by building on the game models presented in [4]. Although the general approach is innocent in the sense of [3] and [7], the two-player games we use are simpler and can be considered a special case where moves are neither questions nor answers but simply "declarations". A notable feature of game semantics is that the λ-definable strategies are effective methods for copying moves uniformly from one "component" of the game to another. For example, the identity strategy on an arena $A \Rightarrow A$ is *everywhere copycat* i.e. P always plays back every O-move (but in the opposite component of A). The key idea is that the innocent strategies definable by untyped λ-terms are, what we call, *effectively almost-everywhere copycat* (EAC). Informally this means that at every position, except in response to finitely many possible O-moves, the strategy is constrained to behave, from that point onwards, uniformly in an everywhere-copycat fashion, just like the identity strategy. Effectively here means that (not only is the strategy itself recursive but also) at every position, the boundary of that finite part of the game tree in which the strategy is *not* forced to play copycat must be computable.

We find it convenient to introduce innocent strategies in a concrete setting whereby (tree) arenas are defined as subsets of $\mathbb{N}^*$ of a certain kind, and this we do in Sect. 2. Section 3 introduces the EAC strategies which give rise to a universal λ-model $\mathcal{D}_{\mathrm{EAC}}$ whose theory is the maximal consistent sensible λ-theory $\mathcal{H}^*$. The definition of such strategies uses an efficient encoding of innocent strategies, as partial functions from $\mathbb{N}^*$ to tuples of numbers, which we call

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 405–419, 1999.

economical form. Sections 2 and 3 should be regarded as a survey of [4], and this paper is a sequel to that work. The notion of EAC strategies has a natural extension to *explicitly and effectively almost-everywhere copycat*. However finding an ambient cartesian closed category for these strategies to inhabit proved to be a painful process as we briefly show in Sect. 4 – the natural analogues fail to work quite as intended. Once this has been overcome we use a reflexive object to describe a λ-algebra which we call $\mathcal{M}$. We formulate a new version of the powerful *Exact Correspondence Theorem* of the earlier work, with which we can show that $\mathcal{M}$ is a universal λ-model which induces the intended equational theory. To our knowledge, $\mathcal{D}_{\mathrm{EAC}}$ and $\mathcal{M}$ are the first syntax-independent universal λ-models.

In [2], Di Gianantonio et al. have obtained game models of the untyped λ-calculus using history-free strategies. They show that all their models induce the same λ-theory $\mathcal{H}^*$ and have asked for "new techniques for overcoming this apparent rigidity of game λ-models". This paper answers that question by constructing a universal game model for the Böhm tree *lambda*-theory.

2 Arenas and Innocent Strategies

This section and the next give a quick introduction to the basic ideas underpinning the main result of the paper. We refer the reader to [4] for further details and to [3] and [5] for proofs of all results quoted. We define an *arena* to be a finite tuple of nonempty trees of *moves*. The root of each tree is called an *initial move*. Our trees are considered "upside-down" with the root at the top, rather like family trees. We can also refer to the child of a node, and say that one node inherits from another, in the same vein. We say that moves at an even depth of the trees (including the roots at depth 0) are *O-moves*, and moves at an odd depth are *P-moves*. O-moves are often denoted by $\bullet$ and P-moves by $\circ$.

We will only be interested in countably branching, countably deep trees. Thus we can encode each tree of the arena as a subset of $\mathbb{N}^*$[1] by inductively labelling the root as ε and the n^{th} child of the move $\boldsymbol{s}$ as $\boldsymbol{s} \cdot n$ (we use the notation $\boldsymbol{s}, \boldsymbol{t}$ etc. to denote sequences). Hence each move of each tree is associated uniquely with a sequence of natural numbers. Conversely, given any subset $A \subseteq \mathbb{N}^*$ which is prefix-closed and has the property that whenever $\boldsymbol{s} \cdot n \in A$ we have $\boldsymbol{s} \cdot m \in A$ for each $m \leq n$, we can form an arena of one tree where the moves are the elements of A, ordered by prefix. *Henceforth by arenas, we shall always mean arenas in sequence-subset (of $\mathbb{N}^*$) form.* For example, the empty sequence $\langle\rangle$ is an arena, which we call the *empty arena*; $\langle\{\,\varepsilon\,\}\rangle$ is the minimal one-tree arena consisting of a root node; the maximal one-tree arena, consisting of an infinitely deep, infinitely branching tree, is $\langle\mathbb{N}^*\rangle$. As the empty arena, the minimal and maximal one-tree arenas are important, we shall name them E, M and U respectively.

There are two major constructions for forming arenas. Suppose $A = \langle A_1, \ldots, A_m\rangle$ and $B = \langle B_1, \ldots, B_n\rangle$ are arenas.

- The *product arena* $A \times B$ is the "disjoint union" of the trees of A and B, the concatenation of their tuples. Formally $A \times B = \langle A_1, \ldots, A_m, B_1, \ldots, B_n\rangle$.

[1] We *do not* include 0 in the set $\mathbb{N}$, and write $\mathbb{N}_0$ for $\mathbb{N} \cup \{\,0\,\}$.

– The *function space arena* $A \Rightarrow B$ is constructed as follows: the initial moves of $A \Rightarrow B$ are those of B; and to the tree below each such initial move, we graft onto it a copy of A. More precisely $A \Rightarrow B = \langle C_1, \ldots, C_n \rangle$ where

$$C_i = \{\, \varepsilon \,\} \cup \{\, a \cdot \boldsymbol{s} \mid 1 \leq a \leq m \wedge \boldsymbol{s} \in A_a \,\} \cup \{\, (a+m) \cdot \boldsymbol{s} \mid a \cdot \boldsymbol{s} \in B_i \,\}.$$

The reader may wish to check that $M \Rightarrow M = \langle \{\, \varepsilon, 1 \,\} \rangle$, and that $U \Rightarrow U$ and U are equal as arenas.

A *justified sequence* of an arena A is a sequence of moves of which each element except the first, which must be an initial move, is equipped with a pointer to some previous move. We call the pointer a *justification pointer* and if the move m^- is pointed to by m we say that m^- *justifies* m. We say that a move m^- in a justified sequence *hereditarily justifies* m if one can reach m^- from m by repeatedly following justification pointers. A justified sequence $\boldsymbol{s}$ is said to be *well-formed* if elements of $\boldsymbol{s}$ alternate between P-moves and O-moves and if $m \in \boldsymbol{s}$ is justified by m^- then the move m is directly beneath m^- in the tree of the arena[2]. Henceforth all justified sequences are assumed to be well-formed.

The *P-view* of a justified sequence $\boldsymbol{s}$, written $\ulcorner \boldsymbol{s} \urcorner$, is given recursively by:

$$\begin{aligned}
\ulcorner \varepsilon \urcorner &= \varepsilon && \text{for initial moves } \varepsilon \\
\ulcorner \boldsymbol{s} \cdot m \urcorner &= \ulcorner \boldsymbol{s} \urcorner \cdot m && \text{for } m \text{ a P-move} \\
\ulcorner \boldsymbol{s} \cdot m^- \cdot \boldsymbol{t} \cdot m \urcorner &= \ulcorner \boldsymbol{s} \urcorner \cdot m^- \cdot m && \text{for } m \text{ an O-move justified by } m^-
\end{aligned}$$

The definition of *O-view*, $\llcorner \boldsymbol{s} \lrcorner$, is given analagously.

A *legal position* of an arena A is a well-formed justified sequence $\boldsymbol{s}$ satisfying the *visibility condition*: for each non-initial P-move m justified by m^-, say $\boldsymbol{s} = \boldsymbol{t_1} \cdot m^- \cdot \boldsymbol{t_2} \cdot m \cdot \boldsymbol{t_3}$, we have that $m^- \in \ulcorner \boldsymbol{t_1} \cdot m^- \cdot \boldsymbol{t_2} \cdot m \urcorner$. Similarly all O-moves are justified by P-moves appearing in the O-view up to that point. Then if $\boldsymbol{s}$ is a legal position then so are $\ulcorner \boldsymbol{s} \urcorner$ and $\llcorner \boldsymbol{s} \lrcorner$. By *a P-view* of an arena A, we shall mean a justified sequence which is the P-view of some legal position of A.

Lemma 1 (View Characterisation). *A justified sequence of an arena A is a P-view if and only if it is well-formed and every non-initial O-move is justified by the immediately preceding P-move.*

Within arenas there are games played out between P and O. A *P-strategy* σ for a single-tree arena A consists of a prefix-closed subset of legal positions of A which is deterministic (if $\boldsymbol{s} \cdot m \in \sigma$ and $\boldsymbol{s} \cdot m' \in \sigma$ for P-moves m and m' then $m = m'$) and such that if $\boldsymbol{s} \cdot m \in \sigma$ for a P-move m and $\boldsymbol{s} \cdot m \cdot m'$ is a legal position of A then $\boldsymbol{s} \cdot m \cdot m' \in \sigma$. An *O-strategy* is defined analogously. However we are more often interested in P-strategies which we will usually just refer to as strategies. For a general arena $A = \langle A_1, \ldots, A_n \rangle$ a P-strategy is an n-tuple of P-strategies, one for each tree. In contrast, an O-strategy is just a single O-strategy on one of the trees, together with information which selects that tree.

[2] The no-dangling-question-mark condition in [3,7] (equivalently the well-bracketing condition) is redundant for our arenas.

If we have strategies σ and τ on arenas $A \Rightarrow B$ and $B \Rightarrow C$ respectively then we can form their composite strategy $\sigma;\tau$ on $A \Rightarrow C$. Informally we do this by identifying O/P-moves of the B component of $A \Rightarrow B$ with P/O-moves of the B component of $B \Rightarrow C$, and then hiding all the moves in B. This is reminiscent of CSP's "parallel composition" and "hiding" operators; see [4] for a formal definition. Similar ideas extend to arenas of multiple trees. An essentially straightforward result, although tedious in proof, is that composition is well-defined and associative. We will only be interested in strategies with a property called *innocence*.

A P-strategy σ is *innocent* if for odd-length legal positions $\boldsymbol{s}$ and $\boldsymbol{t}$ and P-moves m, $\boldsymbol{s} \cdot m \in \sigma \wedge \boldsymbol{t} \in \sigma \wedge \ulcorner \boldsymbol{s} \urcorner = \ulcorner \boldsymbol{t} \urcorner \rightarrow \boldsymbol{t} \cdot m \in \sigma$ and the moves m are justified by moves which are identical in the P-view $\ulcorner \boldsymbol{s} \urcorner = \ulcorner \boldsymbol{t} \urcorner$. i.e. P's next move, and its justification, at each stage depends only on the P-view up to that point. An important fact is that composition of innocent strategies is well-defined (for a proof see [3, §5.3]): if σ is an innocent strategy on $A \Rightarrow B$ and τ an innocent strategy on $B \Rightarrow C$ then $\sigma;\tau$ is an innocent strategy on $A \Rightarrow C$.

The property of innocence means that such a strategy is determined by a partial function from odd-length P-views to *justified P-moves* i.e. P-moves equipped with a justification pointer back into the P-view. In fact, given an innocent strategy σ, we can define a canonical such function, which we write f_σ, that defines it. A function constructed in this way is called *innocent* and we can formalise such functions. We say that f is an *innocent function* if f is a partial function from odd-length P-views in A to justified P-moves of A such that $\mathrm{dom}(f)$ is closed under odd-length prefix, and if $\boldsymbol{s} \cdot m \cdot m' \in \mathrm{dom}(f)$ then $f(\boldsymbol{s}) = m$. We note that such a function can only encode an innocent strategy. The conditions given are required to make the function "strategic", i.e. the set of legal positions it describes are prefix-closed, deterministic and made up of properly justified sequences of moves. These conditions are sufficient to allow us to define the reverse construction of a unique strategy σ_f from an innocent function f such that the construction is invertible (i.e. $f_{\sigma_f} = f$ and $\sigma_{f_\sigma} = \sigma$) and it preserves and reflects inclusion (i.e. $f \subseteq f' \iff \sigma_f \subseteq \sigma_{f'}$). Thus we can identify the representation by innocent function and that by subset of legal positions.

An innocent strategy is said to be *compact* if the graph of its innocent function is finite (i.e. is defined on finitely many P-views). It is said to be *recursive* if the innocent function representing it is recursive. It is easy to see that the composition of two recursive innocent strategies is itself recursive.

Definition 1. *Objects of the* Category of Arenas and Innocent Strategies, $\mathbb{A}$, *are arenas (in sequence-subset form); morphisms $f : A \rightarrow B$ are innocent strategies on the function space arena $A \Rightarrow B$. Composition of morphisms is composition as strategies. The* Category of Arenas and Recursive Innocent Strategies, $\mathbb{A}_{\mathrm{REC}}$, *has recursive arenas as objects and recursive innocent strategies as morphisms.*

Theorem 1. $\mathbb{A}$ *and* $\mathbb{A}_{\mathrm{REC}}$ *are both cartesian closed.*

The terminal object $\mathbf{1}$ of both $\mathbb{A}$ and $\mathbb{A}_{\mathrm{REC}}$ is the empty arena E, and the categorical constructions of product and function space are exactly the respective

arena constructs. The category $\mathbb{A}$ is enriched over dI-domains. One cannot say the same of $\mathbb{A}_{\mathrm{REC}}$, because the computable partial functions do not form a cpo. For example, one can "approximate" the Halting Problem by computable functions.

Scott has observed that every λ-algebra arises from a reflexive object R in some cartesian closed category $\mathbb{C}$; that is, there exist morphisms $Fun : R \to [R \Rightarrow R]$ and $Gr : [R \Rightarrow R] \to R$ such that $Gr\,;\, Fun = \mathrm{id}_{[R \Rightarrow R]}$. Thus we may specify a λ-*algebra* by a 4-tuple $\langle \mathbb{C}, R, Fun, Gr \rangle$ (it is in fact a $\lambda\eta$-algebra if $Fun\,;\, Gr = \mathrm{id}_R$); the underlying set of the λ-algebra is the set $\mathbb{C}(\mathbf{1}, R)$ of global sections. If the reflexive object R *has enough points* (i.e. $\forall\, f, g : R \to R.[\forall\, r : \mathbf{1} \to R.r\,;\, f = r\,;\, g] \to f = g$) then $\langle \mathbb{C}, R, Fun, Gr \rangle$ is a λ-*model* (i.e. a weakly extensional λ-algebra). We refer the reader to [1] for a comprehensive treatment of the model theory of the untyped λ-calculus.

Recall that the arena U has the key property that $U = U \Rightarrow U$ so that in this case the morphisms Fun and Gr are both the identity on U. We can now define the first two of our game λ-algebras (which are both $\lambda\eta$-algebras): $\langle \mathbb{A}, U, \mathrm{id}_U, \mathrm{id}_U \rangle$ which we shall write simply as $\mathcal{D}$, and $\langle \mathbb{A}_{\mathrm{REC}}, U, \mathrm{id}_U, \mathrm{id}_U \rangle$ which we shall write as $\mathcal{D}_{\mathrm{REC}}$. By abuse of notation, we shall use $\mathcal{D}$ and $\mathcal{D}_{\mathrm{REC}}$ to denote the respective underlying sets. Clearly $\mathcal{D}_{\mathrm{REC}} \subset \mathcal{D}$. By a method of approximation we can show that both the $\lambda\eta$-algebras are *sensible* i.e. all unsolvable λ-terms have the same denotation which in this case is given by the everywhere undefined innocent function.

3 Effectively Almost-Everywhere Copycat Strategies

There are three properties that allow for a more compact representation of an innocent strategy:

(i) Each non-initial O-move in any P-view must be a child of the previous move, and the initial move must be ε.

(ii) Given only the O-moves of a P-view and the value of the innocent function on strictly shorter P-views we can reconstruct the original P-view entirely.

(iii) The P-move to which this P-view is mapped must be a child of the move justifying it.

In view of these redundancies, we encode innocent strategies σ, over any single-tree arena, as (partial) maps from $\mathbb{N}^*$ to $\mathbb{N} \times \mathbb{N}_0$ (where $\mathbb{N}_0 = \{0, 1, 2, \cdots\}$). We call this encoding the *economical form* of σ and sometimes write it e_σ (quite often we abuse notation and write it f_σ too). It is defined as follows:

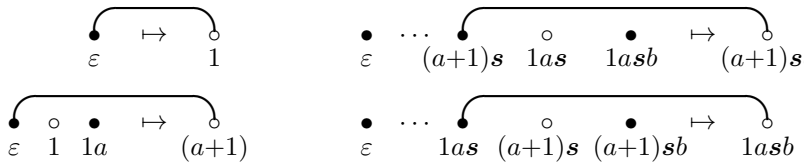
$e_\sigma : \langle v_1, \ldots, v_n \rangle \mapsto (i, p)$ if and only if

$f_\sigma :$ $\bullet_{\varepsilon}$ $\circ_{\boldsymbol{s}_1}$ $\bullet_{\boldsymbol{s}_1 v_1}$ $\circ_{\boldsymbol{s}_2}$ $\bullet_{\boldsymbol{s}_2 v_2}$ $\cdots$ $\circ_{\boldsymbol{s}_{n-p}}$ $\bullet_{\boldsymbol{s}_{n-p} v_{n-p}}$ $\cdots$ $\circ_{\boldsymbol{s}_n}$ $\bullet_{\boldsymbol{s}_n v_n}$ $\mapsto$ $\circ_{\boldsymbol{s}_{n-p}(v_{n-p} i)}$

Justification pointers in the P-view can be deduced from the behaviour of f_σ on shorter P-views, and so have been omitted. Note that each $\boldsymbol{s}_i$ is a sequence of natural numbers.

Furthermore, we can expand any partial function $f : \mathbb{N}^* \rightharpoonup \mathbb{N} \times \mathbb{N}_0$ which has prefix-closed domain and satisfies $f(\boldsymbol{v}) = (i, p) \to 0 \leq p \leq |\boldsymbol{v}|$ into an innocent strategy on U. Depending on the function, we might not need the whole of U to contain the strategy. We could extend this idea for multiple-tree arenas, but since we will not use it except on the arena U there is no need to do so.

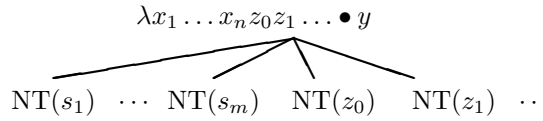
Example 1. The following is the innocent function of the "copycat" strategy id_U:



Here $\boldsymbol{s}$ range over sequences of appropriate parity, a and b over positive natural numbers. The reader is invited to check that the economical form of this strategy is given by: $\varepsilon \mapsto (1, 0), i \mapsto (i + 1, 1)$ and for nonempty sequences $\boldsymbol{v}$, $\boldsymbol{v}i \mapsto (i, 1)$.

A principle of the λ-calculus is that a term can be applied successively to any other term. So the term $\lambda x.x$ (say) is really more like "$\lambda x z_0 z_1 z_2 \ldots \bullet x z_0 z_1 z_2 \ldots$" (we use a large dot $\bullet$ to make the "end" of the infinite chain of abstractions really clear). Thus there is some notion of infinite η-expansion. If we think about the denotation of $\lambda x.x$ in the game models, it is similarly expanded — it copies the whole of the first subtree to the rest of the arena, as if copying not only the x variable but also all of its arguments. This correspondence turns out to be general, and can be made precise by relating innocent strategies in economical form to a kind of (infinitely) η-expanded Böhm trees first studied by Nakajima in [6]. We call a formal connexion of this form an *Exact Correspondence Theorem.*

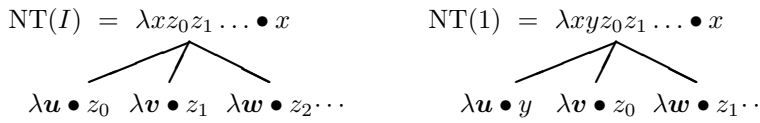
For a λ-term s the *Nakajima tree* of s, written $\mathrm{NT}(s)$, is (informally) the countably branching, countably deep tree labelled as follows. If s is unsolvable then $\mathrm{NT}(s) = \bot$, the empty tree. If s has HNF $\lambda x_1 \ldots x_n \bullet y s_1 \ldots s_m$ then



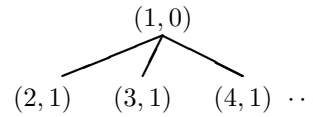
where $z_0, z_1, \ldots$ are countably many fresh variables. The process of finding such fresh variables given in [6] is quite complicated. In [4] we propose a *variable-free* representation of Nakajima trees so that for a closed term s, $\mathrm{NT}(s)$ is represented as $\mathrm{VFF}(s)$, a partial function from $\mathbb{N}^*$ to $\mathbb{N} \times \mathbb{N}_0$. Note that the "infinitely nested" λ-abstractions of the form $\lambda z_1 z_2 \ldots \bullet y$, which label the nodes of a Nakajima tree (of a closed term), can be coded as a pair (i, r) whereby the head variable y is the i^{th} in the infinite list of variables bound by the λ-abstraction situated r levels up in the tree. The map $\mathrm{VFF}(s)$ is just a function that maps occurrences (of nodes) to such labels encoded as pairs of numbers.

The theorem of key importance in [4] is the *Exact Correspondence Thoerem*, which states that for every closed λ-term s, the innocent strategy denoting s (in both $\mathcal{D}$ and $\mathcal{D}_{\mathrm{REC}}$) given in economical form is exactly $\mathrm{VFF}(s)$, the Nakajima tree of s in variable free form.

Example 2. We now introduce example terms and strategies which we will use repeatedly to illustrate many of the concepts in the rest of the paper. Consider the terms $I = \lambda x.x$ and $1 = \lambda xy.xy$. The reader may wish to verify that the following represents the first two levels of the Nakajima trees of those terms:



After renaming of bound variables, these are the same. Since I and 1 differ only by η-conversion, this should be no surprise. Thus we can calculate their common variable-free form, the first two levels of which is:



For example, the node labelled $(2, 1)$ means that the head variable of the corresponding node in the Nakajima tree is found as the second in the list of variables abstracted at the node one level above. The Exact Correspondence Theorem tells us that $[\![I]\!] = [\![1]\!]$ has the economical form which is given (in part) by $\varepsilon \mapsto (1, 0), \langle 1\rangle \mapsto (2, 1), \langle 2\rangle \mapsto (3, 1)$ and so on.

We say that a λ-algebra is *universal* if every element is the denotation of some λ-term. By the Exact Correspondence Theorem, it is easy to see that neither $\mathcal{D}$ nor $\mathcal{D}_{\mathrm{REC}}$ is universal, since no *non-trivial* compact innocent strategy can be the denotation of any λ-term (note that the only finite Nakajima tree is the single-node tree $\bot$). Our aim in the rest of this section is to characterise the definable parts of $\mathcal{D}_{\mathrm{REC}}$, and we shall do so by capturing the right ambient CCC.

Notation For tree-like $A \subseteq \mathbb{N}^*$ (i.e. those subsets which are prefix-closed and satisfy $\boldsymbol{s} \cdot n \in A \to \boldsymbol{s} \cdot m \in A$ for all $m < n$) and for any $\boldsymbol{s} \in A$ we define

$$A \mathbin{@} \boldsymbol{s} = \text{the subtree of } A \text{ rooted at } \boldsymbol{s}$$
$$A^{>m} = \text{the tree obtained from } A \text{ by deleting the first } m \text{ branches.}$$

For example, for the maximal single-tree arena U, we have $U \mathbin{@} \boldsymbol{s} = U = U^{>n}$ for all sequences $\boldsymbol{s}$ and numbers n. Next fix an innocent strategy in economical form f and let $\boldsymbol{v} \in \mathrm{dom}(f)$. We shall use the following shorthand:

$$\mathbf{m_o}^f(\boldsymbol{v}) = \text{the last move of the P-view encoded by } \boldsymbol{v}$$
$$\mathbf{m_p}^f(\boldsymbol{v}) = \text{the response of } \sigma_f \text{ at the P-view.}$$

Note that the former is by definition an O-move and the latter a P-move. We omit the superscript f wherever it is clear which strategy is intended. For example, for any innocent strategy f the O-move $\mathbf{m_o}(\varepsilon)$ is the initial move ε and $\mathbf{m_p}(\varepsilon)$ is the first P-move made by σ_f in response. Now we can define a new property of strategies:

Definition 2. *Consider an innocent strategy in economical form* $f : \mathbb{N}^* \to \mathbb{N} \times \mathbb{N}_0$*, over some single-tree arena* A*. We say that* f *is* everywhere copycat *(EC) at* $\boldsymbol{v} \in \mathbb{N}^*$ *if* f *is undefined at* $\boldsymbol{v}$ *or the following hold:*

(i) The arenas $A \mathbin{@} \mathbf{m_o}(\boldsymbol{v})$ *and* $A \mathbin{@} \mathbf{m_p}(\boldsymbol{v})$ *are order-isomorphic (with respect to the prefix ordering).*

(ii) Whenever $\boldsymbol{w} \geq \boldsymbol{v}$ *we have that for all* $i \in \mathbb{N}$ $f(\boldsymbol{w} \cdot i) = (i, 1)$*.*

(iii) If $f(\boldsymbol{v}) = (i, p)$ *then* $p > 0$*.*

We say that f *is* almost-everywhere copycat *(AC) at* $\boldsymbol{v}$ *if* f *is undefined at* $\boldsymbol{v}$ *or there exist numbers* $t_v \in \mathbb{N}_0$ *and* $o_v \in \mathbb{Z}$ *with* $o_v \leq t_v$ *called the* copycat threshold *and* offset *respectively, such that*

(i) The arenas $(A \mathbin{@} \mathbf{m_o}(\boldsymbol{v}))^{>(t_v - o_v)}$ *and* $(A \mathbin{@} \mathbf{m_p}(\boldsymbol{v}))^{>t_v}$ *are isomorphic.*

(ii) For all $i > t_v$*,* $f(\boldsymbol{v} \cdot i) = (i - o_v, 1)$ *and* f *is everywhere copycat at* $\boldsymbol{v} \cdot i$*.*

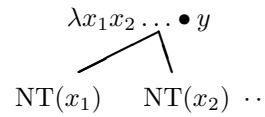
(iii) For all $\boldsymbol{w} \geq (\boldsymbol{v} \cdot k)$ *with* $k \leq t_v$*, if* $f(\boldsymbol{w}) = (i, |\boldsymbol{w}| - |\boldsymbol{v}|)$ *then* $i \leq t_v - o_v$*.*

(iv) If $f(\boldsymbol{v}) = (i, 0)$ *then* $i \leq t_v - o_v$*.*

(Note that f *is EC at* $\boldsymbol{v}$ *if and only if* f *is AC at* $\boldsymbol{v}$ *with* $t_v = o_v = 0$*.)*

Finally, we say that f *is* effectively almost-everywhere copycat *(EAC) if* f *is computable, almost-everywhere copycat at every sequence on which it is defined and the functions* $\boldsymbol{v} \mapsto t_v$ *and* $\boldsymbol{v} \mapsto o_v$ *are computable. A strategy* σ *over an arena* A *is EAC if its innocent function is EAC, and we can generalise to multiple-tree arenas in the usual way.*

To illustrate the definition of everywhere copycat strategies, suppose f is defined at $\boldsymbol{v}$. Intuitively we say that f is everywhere copycat at $\boldsymbol{v}$ if, from $\mathbf{m_p}(\boldsymbol{v})$ onwards, f's behaviour is simply to play copycat for as long as the arena will allow it. So if O's move is mi, the i^{th} child of the justifying move m, then P responds with the i^{th} child of the move immediately preceding m in the P-view. Condition (i) in the definition guarantees that P's copycat move will always be available. As before we will primarily be interested in strategies on U. Since $U \mathbin{@} \boldsymbol{s} = U = U^{>n}$ for all sequences $\boldsymbol{s}$ and numbers n, Condition (i) will always hold. Condition (ii) is best understood with reference to the Exact Correspondence Theorem which relates innocent strategies to Nakajima trees. It specifies that the subtree of the Nakajima tree corresponding to f, rooted at $\boldsymbol{v}$, has the following shape:



Condition (iii) of the definition is a technicality, which ensures that the variable y is not one of the x_i.

Definition 3. *The* category of arenas and EAC strategies, $\mathbb{A}_{\mathrm{EAC}}$*, has recursive arenas as objects and EAC strategies on* $A \Rightarrow B$ *as morphisms from* A *to* B*.*

A main result in [4] is that the category $\mathbb{A}_{\mathrm{EAC}}$ is well-defined; the proof that EAC strategies compose is highly technical. In fact,

Theorem 2. $\mathbb{A}_{\mathrm{EAC}}$ *is cartesian closed.*

The arena U is still an object of $\mathbb{A}_{\mathrm{EAC}}$ and still equal to its function space. Thus we can define a $\lambda\eta$-algebra $\langle \mathbb{A}_{\mathrm{EAC}}, U, \mathrm{id}_U, \mathrm{id}_U \rangle$ which we shall denote by $\mathcal{D}_{\mathrm{EAC}}$. Properties of $\mathcal{D}_{\mathrm{EAC}}$ will be presented later.

4 Effectively and Explicitly Almost-Everywhere Copycat Strategies

We wish to find a new game model which invalidates the rule of η-conversion. To do so, we would require the terms I and 1 to be denoted differently. They have the same variable-free form of Nakajima tree, so it is not apparent how this might be achieved. The key is to make use of the fact that the copycat thresholds are not unique — any number greater than a given valid copycat threshold is also a valid copycat threshold. Different thresholds (at some P-view) may be used to distinguish I and 1.

This idea is prompted by the observation that when one compares a term with its denotation, the part of the EAC strategy which is specified by the rules of copycat corresponds precisely to the part of the Nakajima tree which has been generated by η-expansion (i.e. the part of the tree with the fresh variables as the head variables). Recall the Nakajima trees of I and 1 — the former has fresh variables appearing at every node except the root, whereas the latter is similar except that there is not a fresh variable at the first child of the root. Therefore we aim to find a model where I and 1 are represented by the strategy with the same moves, but the copycat threshold of $[\![I]\!]$ at the first P-view is 0, whereas that of $[\![1]\!]$ is 1.

However, the definition of an EAC strategy is stated in terms of the existence of some computable function which associates a pair of numbers to each P-view of the strategy and this function is *not* specified along with the strategy. (A consequence of this is that there is no computable procedure for finding valid thresholds for an EAC strategy.) It is really the thresholds (rather than the offsets) that are important because, for a certain P-view $\boldsymbol{v}$ of an EAC strategy σ, the copycat threshold t gives enough information to compute the offset o directly. This motivates the following definition:

Definition 4. *An* effectively and explicitly almost-everywhere copycat strategy *(EXAC strategy) is given by a pair $\langle \sigma, t_\sigma \rangle$, where σ is an EAC strategy and t_σ is an effective function mapping the P-views where σ is defined to valid copycat thresholds. We sometimes write the EXAC strategy $\langle \sigma, t_\sigma \rangle$ just as σ.*

We will usually refer to the first and second part of an EXAC strategy as the "(underlying) EAC strategy (part)" and the "threshold function (part)", respectively. In view of our comments above, however, we will sometimes speak of the offsets as if they too are specified by the threshold function.

This definition allows us to make the intended finer distinction between strategies: two strategies with the same moves must be equal as EAC strategies, but may have different copycat thresholds and so can be distinguished as EXAC strategies. There is an obvious forgetful map from EXAC strategies to EAC

strategies, which takes only the strategy part (i.e. erasing the threshold information).

In a similar vein to the economical form of innocent strategies, using the same encoding of a P-view as a sequence of natural numbers, we can give an economical form of EXAC strategies over single-tree arenas. We can also take advantage of the fact that parts of the strategy are completely dictated by its copycat nature. Let us say that a P-view is *entirely explicit* if none of the O-moves in it exceed the copycat threshold of the P-view at which they are made. Thus if a P-view is not entirely explicit the ensuing move can be deduced from the threshold and offset of the P-view preceding the first O-move in it which did exceed the copycat threshold.

Definition 5. *The* economical form *of an EXAC strategy is a map from* $\mathbb{N}^*$ *to* $\mathbb{N} \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{Z}$. *The domain is the encoding of P-views in the usual way. The map is defined at a sequence* $\boldsymbol{v}$ *only if the P-view encoded by* $\boldsymbol{v}$ *is entirely explicit, in which case*

$$\boldsymbol{v} \mapsto (i, r, t, o)$$

where the resulting P-move is encoded as before — it is the i^{th} *child of the move* $2r$ *from last of the P-view — and the copycat threshold and offset at this P-view are* t *and* o *respectively.*

Example 3. We take the EXAC strategies η_0 and η_1 to be $\langle [\![I]\!], t_0 \rangle$ and $\langle [\![1]\!], t_1 \rangle$, where t_0 maps every P-view to the threshold 0 and t_1 does likewise except that the minimal P-view is mapped to the threshold 1. Since $[\![I]\!] = [\![1]\!]$, they have the same EAC strategy part, but different threshold functions. These are the suggestions we made for the denotations of I and 1 in a model not supporting η-conversion. Nearly every P-view of either is not entirely explicit, and the respective economical forms are given by:

$$\varepsilon \mapsto (1,0,0,-1) \quad \text{and} \quad \begin{array}{l} \varepsilon \mapsto (1,0,1,-1) \\ \langle 1 \rangle \mapsto (2,1,0,0) \end{array}$$

We now need a method to compose EXAC strategies. Of course the EAC strategy part will just be the standard composition of innocent strategies, and we give below an algorithm for computing the composition of the threshold functions.

Algorithm (The Composition Algorithm). Let $\langle \sigma, t_\sigma \rangle$ be an EXAC strategy over $A \Rightarrow B$, and $\langle \tau, t_\tau \rangle$ be an EXAC strategy over $B \Rightarrow C$. Take a P-view $\boldsymbol{v}$ on which the strategy $\sigma; \tau$ (which is given by the usual composition of innocent strategies) is defined and suppose that the last move of the P-view is $\underline{m}$ and the resulting move is m.

We write $\boldsymbol{u} = \mathbf{u}(\boldsymbol{v}, \sigma, \tau)$ for the *uncovering* of the composition up to the move m. A formal definition can be found in [3] or [4], but we may describe it as the sequence of moves of the composition which result after the P-view $\boldsymbol{v}$, including any relevant B-moves which would be hidden by the composition. It will be of the form $\langle \varepsilon, \ldots, \underline{m}, m_1, m_2, \ldots, m_{p-1}, m_p, m \rangle$.

The moves m_i are the intermediate interactions which might have taken place between σ and τ before the move m became the visible outcome, and are all in the arena B. Possibly there are no such intermediate moves, in which case $p = 0$. We do not care about justification pointers, and for tidiness set $m_0 = \underline{m}$ and $m_{p+1} = m$.

For $1 \leq i \leq p+1$ we consider the P-view $\boldsymbol{u_i}$ that the strategies σ or τ are faced with when the move m_i was made. (For details on how one may define such a P-view precisely see [4] or [5]). Define t_i and o_i to be the copycat threshold and offset of σ, or τ as appropriate, at the P-view $\boldsymbol{u_i}$. These are specified by t_σ or t_τ. Then set:

$$\begin{array}{rl} t'_i = t_i + |A|, & \text{if } m_i \text{ is a root of the arena } B \\ \quad t_i, & \text{otherwise} \\ o'_i = o_i + |A|, & \text{if } m_i \text{ is a root of the arena } B \\ \quad o_i, & \text{otherwise} \end{array} \qquad \begin{array}{r} T_1 = t'_1 \\ O_1 = o'_1 \\ T_{i+1} = \max(T_i + o'_{i+1}, t'_{i+1}) \\ O_{i+1} = O_i + o'_{i+1} \end{array}$$

$$\begin{array}{l} \mathbf{t} = T_{p+1} \\ \mathbf{o} = O_{p+1} - |A| + |B|, \text{ if } \underline{\mathrm{m}} \text{ is a root of the arena } C \\ \quad O_{p+1}, \qquad\qquad \text{otherwise} \end{array}$$

(By $|A|$ we mean the number of trees in the arena A). Then $\mathbf{t}$ and $\mathbf{o}$ are the copycat threshold and offset of the composition $\langle \sigma, t_\sigma \rangle; \langle \tau, t_\tau \rangle$ at the P-view $\boldsymbol{v}$.

Now we must show that this method does indeed produce an EXAC strategy, i.e. that the composite threshold function specifies valid thresholds and offsets for the composite strategy. In fact it does so only under some restrictions, for which we need an additional definition.

Definition 6. *Let σ be an EAC strategy over a single-tree arena. If σ has a first move, then it has a copycat threshold and offset, say t and o, at the P-view consisting only of the root O-move (we call this P-view the minimal P-view). The* l-number *of σ is the value $t - o$, and we write it $\mathbf{l}(\sigma)$.*

If $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ is an EAC strategy over an arena with n trees, and defined on at least one of the minimal P-views, then we say $\mathbf{l}(\sigma) = \min^n_{i=1, \sigma_i \neq \perp} \{\mathbf{l}(\sigma_i)\}$.

This is termed the l-number of σ because, as will eventually be shown, it corresponds to the number of λ-abstractions at the root of the Böhm tree of the term whose denotation is σ. For example $\mathbf{l}(\eta_0) = 1$ and $\mathbf{l}(\eta_1) = 2$, and we will be able to show that η_0 is the denotation of $\lambda x.x$ and η_1 the denotation of $\lambda xy.xy$.

Theorem 3. *If $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ are EAC strategies satisfying $\mathbf{l}(\sigma) \geq |A|$ (or σ is everywhere undefined) and $\mathbf{l}(\tau) \geq |B|$ (or τ is everywhere undefined) then Algorithm 4 produces valid copycat thresholds and offsets for $\sigma; \tau$.*

There is an "obvious" category, which derives directly from the conditions required for the composition algorithm to work correctly.

Definition 7. *The* category of arenas and EXAC strategies*, written* $\mathbb{A}_{\mathrm{EXAC}}$*, has recursive arenas as objects, and the morphisms from A to B are the EXAC strategies on the arena $A \Rightarrow B$ which have l-number greater than or equal to $|A|$, or are everywhere undefined. The identity morphism on A is the EXAC strategy $\langle id_A, 0\rangle$, i.e. the copycat threshold is zero everywhere.*

One can show that this does indeed specify a category. Also, $\mathbb{A}_{\mathrm{EXAC}}$ has the obvious terminal object — the empty arena E — and products given in the usual way. However, $\mathbb{A}_{\mathrm{EXAC}}$ does not form a CCC with the usual constructions, as the following example shows:

Example 4. Suppose that $\sigma : A \times B \to C$. Then we know that $\mathbf{l}(\sigma) \geq |A| + |B|$. We need a morphism $\Lambda(\sigma) : A \to B \Rightarrow C$, which must have l-number at least $|A|$, so we could take $\Lambda(\sigma)$ to be the same EXAC strategy as σ. However this choice may not be unique. For example, consider η_0 and η_1 as defined earlier in this section. One can verify that both η_0 and η_1 can be considered as morphisms $U \to U \Rightarrow U$ and that in this case $\eta_0 \times \mathrm{id}_U ; \mathrm{eval}_{U,U} = \eta_1 \times \mathrm{id}_U ; \mathrm{eval}_{U,U} : U \times U \to U$, and that this is the same as the morphism $U \times U \to U$ described by η_1. Hence there are two candidates for $\Lambda(\eta_1)$.

It is not clear that $\mathbb{A}_{\mathrm{EXAC}}$ forms a CCC with any unusual function space constructions either.

If we try to fix the definition of $\mathbb{A}_{\mathrm{EXAC}}$, by cutting down the homsets some more, it becomes clear that one must also specify minimum copycat thresholds at the minimal P-views, along with minimum l-numbers. The obvious solution still does not work, and we can repeat the fixing-up process to obtain a sequence of failures — each is either not a category at all because identities fail to work properly, or has a non-uniqueness of curried morphisms as above.

We now present a new category based on EXAC strategies, which does form a CCC. Although it does appear to be much more complicated than the "almost-CCC" $\mathbb{A}_{\mathrm{EXAC}}$, it seems to be the natural limit of the fixing-up process.

Firstly let us write $\mathrm{br}(A)$ for the number of branches of a tree at the root (assuming that this is finite). Then we can write $\mathrm{br}(A \mathbin{@} m)$, for any move m of a finitely-branching forest A, to mean the number of direct children of m in A. Then we make the following definition:

Definition 8. *Let A be an arena and X a finitely-branching subarena*[3] *of A. We say that an EXAC strategy σ over A is* X-explicit *if the following holds:*

Let $\sigma : \boldsymbol{v} \mapsto (i, r, t, o)$ be the economical form of any clause of the innocent function. Suppose that the sequence $\boldsymbol{v}$ codes a P-view ending in the O-move $\underline{m}$, and that the consequent P-move encoded by this clause is m. Then:

(i) if $\underline{m}$ is in the subarena X then $t - o \geq br(X \mathbin{@} \underline{m})$,

(ii) if m is in the subarena X then $t \geq br(X \mathbin{@} m)$.

[3] We say that the arena $A = \langle A_1, \cdots, A_m\rangle$ is a *subarena* of $B = \langle B_1, \cdots, B_n\rangle$ if $m = n$, and for each i, A_i is a subset of B_i. We say that an arena is *finitely-branching* if every tree in it is finitely-branching

An intuitive description of this definition is the following: The subarena X determines a part of the arena A where the strategy is known to be explicitly defined, i.e. moves in X are neither in the "domain" nor the "range" of automatic copycat forced by the threshold information of σ. This means that given a strategy σ over A which is X-explicit, any P-view of σ with moves only in X is entirely explicit.

Definition 9. *The category $\mathbb{XA}_{\mathrm{EXAC}}$, or simply $\mathbb{XA}$, is given by the following: objects are pairs (A, X) consisting of a recursive arena A and a finitely-branching recursive subarena X; a morphism $\sigma : (A, X) \to (B, Y)$ is an EXAC strategy on $A \Rightarrow B$ which is $(X \Rightarrow Y)$-explicit. Composition of morphisms is composition of EXAC strategies, and the identity strategy on (A, X), $id_{(A,X)}$, is the EXAC strategy $\langle id_A, t\rangle$, where id_A is the EAC identity strategy on A, and t is the function that takes the least value on every P-view which still leaves the EXAC strategy $\langle id_A, t\rangle$ as $(X \Rightarrow X)$-explicit.*

The fact that the composition algorithm gives valid thresholds and offsets for EXAC strategies satisfying these conditions comes from Theorem 3 — a morphism $\sigma : (A, X) \to (B, Y)$ is $(X \Rightarrow Y)$-explicit, and X has the same number of trees as A, so in particular $\mathbf{l}(\sigma) \geq |A|$.

Theorem 4. *$\mathbb{XA}$ forms a CCC with the following constructions: the terminal object $\mathbf{1}$ is (E, E), where E is the empty arena; the product $(A, X)\times(B, Y)$ is $(A\times B, X\times Y)$, with the threshold functions for projections specified in the same style as identities; the exponential object $(A, X) \Rightarrow (B, Y)$ is $(A \Rightarrow B, X \Rightarrow Y)$, and the evaluation map $eval_{(B,Y),(C,Z)}$ is the same EXAC strategy as $id_{(B,Y)\Rightarrow(C,Z)}$.*

Now that we have found an ambient CCC for the EXAC strategies, we can construct another λ-algebra based on it. In this category, however, the reflexive object is not isomorphic to its function space — exactly as we would hope for a model invalidating η-conversion.

Let us write U^0 for the object $\langle U, M\rangle$ of $\mathbb{XA}$, and U^1 for the object $U^0 \Rightarrow U^0$. Here U and M are the maximal and minimal single-tree arenas described in Sect. 2. We define morphisms $F : U^0 \to U^1$ and $G : U^1 \to U^0$ to both be given by the EXAC strategy η_1 (the definition of which can be found in Sect. 4). It is straightforward to check that this does give proper morphisms, and that they satisfy $G; F = \mathrm{id}_{U^1}$ and $F; G \neq \mathrm{id}_{U^0}$.

Hence we can identify a new λ-algebra $\langle \mathbb{XA}, U^0, F, G\rangle$ which invalidates the rule of η-conversion; we denote this λ-algebra $\mathcal{M}$. By erasing all threshold information, we can reduce $\mathcal{M}$ to (a subset of) $\mathcal{D}$ and deduce that $\mathcal{M}$ is also sensible.

In the same way that the denotation of a term in the model $\mathcal{D}_{\mathrm{EAC}}$ had a strong connection with its Nakajima tree, the denotation in $\mathcal{M}$ corresponds closely to (a variable-free version of) the Böhm tree. The variable-free form of the Böhm tree of a term is similar to the construction VFF mentioned earlier and defined in [4], but it includes extra information describing how many abstractions there are at each node, and how many children.

Definition 10. *For a* $(\mathbb{N} \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{Z})$*-labelled tree* p *the tree* p^* *is the same tree labelled identically, except that nodes at depth* d *labelled* $(i, d+1, t, o)$ *are relabelled* $(i, d+2, t, o)$*.*

Similarly the tree $\{\, p \,\}^n$*, for* $n \in \mathbb{N}_0$*, is labelled identically except that firstly the node at the root* (i, r, t, o) *is first relabelled to* $(i, r, t, o-n)$*, and then nodes of depth* d *are relabelled as follows:*

(i) those labelled (i, d, t, o) *are relabelled* $(i+n, d, t, o)$*;*

(ii) those labelled $(i, d+1, t, o)$ *for* $i \leq n$ *are relabelled* $(n-i+1, d, t, o)$*;*

(iii) those labelled $(i, d+1, t, o)$ *for* $i > n$ *are relabelled* $(i-n, d+1, t, o)$*.*

For a term s *with free variables within* Δ *the* variable-free form *of the Böhm tree of* s*,* $VFBT_\Delta(s)$*, is the following* $(\mathbb{N} \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{Z})$*-labelled tree:*

$$VFBT_\Delta(s) = \bot, \qquad \textit{the empty tree, for unsolvable } s.$$

$$VFBT_\Delta(\lambda x_1 \ldots x_n \bullet s) = \{\, VFBT_{\Delta \cdot \langle x_1, \ldots, x_n \rangle}(s) \,\}^n, \quad \textit{if } s \textit{ is of the form } v_j s_1 \ldots s_m.$$

$$VFBT_\Delta(v_j s_1 \ldots s_m) = \begin{array}{ccc} & (j,1,m,m) & \\ \diagup & & \diagdown \\ VFBT_\Delta(s_1)^* & \cdots & VFBT_\Delta(s_m)^* \end{array}$$

where $\Delta = \langle v_k, \ldots, v_1 \rangle$ *(note the reverse order).*

Exactly as before we can show that, at each node of the Böhm tree of s, the first two elements of the tuple at the corresponding node of VFBT(s) encode the head variable by counting how many levels one goes up the tree, and how many abstractions along, to find where the variable is abstracted. The third component just counts the number of children at the node, and the fourth is the number of children minus the number of abstractions. We choose to encode the number of abstractions in this rather elliptic fashion in order to make the following theorem easier to state:

Theorem 5 (Exact Correspondence for $\mathcal{M}$). *If* $s \in \Lambda$ *with free variables in* $\Delta = \langle v_k, \ldots, v_1 \rangle$ *then* $[\![s]\!]^{\mathcal{M}}_\Delta = \{\, VFBT_\Delta(s) \,\}^k$ *when the former is considered as an EXAC strategy in economical form and the latter as a labelling function.*

In particular for closed terms s*,* $[\![s]\!]_\varepsilon = VFBT_\varepsilon(s)$

Example 5. Although it is hard to see directly, the given definition of VFBT does work as intended. One may check that VFBT(I) and VFBT(1) are:

$$(1,0,0,-1) \qquad \text{and} \qquad \begin{array}{c} (1,0,1,-1) \\ | \\ (2,1,0,0) \end{array}$$

The node $\lambda xy.x$, in the Böhm tree of 1, corresponds to the node of VFBT(1) labelled $(1,0,1,-1)$, which is so labelled because the head variable is the first abstracted variable zero levels up the tree (namely x), the node has one child, and the number of abstractions at this level is $1-(-1)=2$. The Exact Correspondence Theorem gives us the economical forms of $[\![I]\!]$ and $[\![1]\!]$ which, as we hoped, are the EXAC strategies η_0 and η_1 described earlier in this section.

The Exact Correspondence results are of key importance in examining the local structure of the game models. In [4] we use them, and a powerful result which we call the Separation Lemma, to obtain proofs that $\mathcal{D}_{\mathrm{EAC}}$ is a *universal* and *order-extensional* λ-model, with equational theory given by $\mathcal{H}^*$ (the maximal consistent sensible theory). The models $\mathcal{D}$ and $\mathcal{D}_{\mathrm{REC}}$ are neither universal nor extensional, but do generate the same equational theory on terms.

In a similar way, we can use the Exact Correspondence Theorem for $\mathcal{M}$ to prove the following:

Theorem 6. *(i) $\mathcal{M}$ is universal i.e. every element is definable as the denotation of some term of the λ-calculus. (ii) $\mathcal{M}$ is weakly extensional, so it is a λ-model (for a discussion of weak extensionality see [1, §5]. (iii) $\mathcal{M}$ equates two terms of the λ-calculus precisely when they have the same Böhm tree. Thus the local structure of the model is the Böhm tree theory $\mathcal{B}$.*

5 Further Work

Although we succeeded in our aim to find a universal game model of $\mathcal{B}$, there are other questions which the work prompts. Firstly, one might ask if there is a less syntactic way to arrive at $\mathcal{D}_{\mathrm{EAC}}$ from $\mathcal{D}_{\mathrm{REC}}$, perhaps by some sort of *extensional collapse*. In fact extensional collapse appears to be insufficient, and further investigation would be of interest. In another direction, we can use $\mathcal{D}$, $\mathcal{D}_{\mathrm{REC}}$, $\mathcal{D}_{\mathrm{EAC}}$ and $\mathcal{M}$ as very natural combinatory algebras which are in some sense *sequential*. Therefore one might wish to study realizability models over them. Finally, and more practically, we could examine the Böhm tree composition algorithm given by the game model: it is quite different from the usual method in that it is "demand-driven" – for every node of the answer only the relevant nodes of the composed trees are examined.

References

[1] Barendregt, H.: *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[2] Di Gianantonio, P., Franco, G., Honsell, F.: Games Semantics for Untyped λ-Calculus, Preprint, announced on the Types Mailing List. April 1998.

[3] Hyland, J. M. E, Ong, C.-H. L.: On Full Abstraction for PCF: I, II and III. To appear in *Information and Computation*, 133 pages, ftp-able. 1994.

[4] Ker, A., Nickau, H., Ong, C.-H. L.: Game Models of Untyped λ-Calculus. Submitted for publication, 49 pages, ftp-able. 1998.

[5] McCusker, G.: *Games and Full Abstraction for a Functional Metalanguage with Recursive Types*. Cambridge University Press, 1998.

[6] Nakajima, R.: Infinite Normal Forms for the λ-Calculus. In Goos, G. and Hartmanis, J., editors, *Symposium on λ-Calculus and Computer Science*, *LNCS* 37, pages 62–82. Springer-Verlag, 1975.

[7] Nickau, H.: *Hereditarily Sequential Functionals: A Game-Theoretic Approach to Sequentiality*. Dissertation, Universität GH Siegen. Shaker-Verlag, 1996.

Anti-Symmetry of Higher-Order Subtyping

Adriana Compagnoni and Healfdene Goguen

[1] Stevens Institute of Technology, Castle Point on Hudson, Hoboken, NJ 07030, USA. abc@cs.stevens-tech.edu. Tel:201-216-5328. Fax:201-216-8249.
[2] AT&T Labs, 180 Park Ave., Florham Park, NJ 07932, USA. hhg@att.com.*

Abstract. This paper shows that the subtyping relation of a higher-order lambda calculus, $\mathcal{F}^{\omega}_{\leq}$, is anti-symmetric. It exhibits the first such proof, establishing in the process that the subtyping relation is a partial order—reflexive, transitive, and anti-symmetric up to β-equality. While a subtyping relation is reflexive and transitive by definition, anti-symmetry is a derived property. The result, which may seem obvious to the non-expert, is technically challenging, and had been an open problem for almost a decade. In this context, typed operational semantics for subtyping offers a powerful new technology to solve the problem: of particular importance is our extended rule for the well-formedness of types with head variables. The paper also gives a presentation of $\mathcal{F}^{\omega}_{\leq}$ without a relation for β-equality, apparently the first such, and shows its equivalence with the traditional presentation.

1 Introduction

Object-oriented programming languages such as Smalltalk, C++, Modula 3, and Java have become popular because they encourage and facilitate software reuse and abstract design. One attempt to give a theoretical understanding of these object-oriented programming languages has been to introduce type systems with features to model constructs from object-oriented programming languages [8,10], for example bounded quantification [20] and recursive types [2].

Metatheoretic properties of the type systems are important to justify the programming languages being modeled. One important property of a type system is subject reduction or type preservation, which states that evaluation of programs preserves their type. This is one of the central results of an earlier paper [15] about $\mathcal{F}^{\omega}_{\leq}$, which also showed the correctness of the algorithms for type-formation and subtyping. Another important property for type systems is the decidability of type-checking and subtyping: a compiler should be able to process basic type information reliably without help from the programmer in order to prevent basic programming errors. Decidability of type-checking ensures that this will always be possible, and decidability of subtyping is a crucial step to proving this. This result was proved for our calculus in [14].

* Most of this author's work was carried out at LFCS, University of Edinburgh, JCMB, The King's Buildings, Edinburgh, EH9 3JZ, UK.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 420–438, 1999.

The subtyping relation has been extensively researched because of its importance in applications to programming languages [1,6,21,28], proof assistants [3,25,29], and metatheoretical studies [4,5,7,10,14,15,16,18,19,24,30], to name a few. However, none of these studies has established the anti-symmetry of the subtyping relation for a higher-order calculus. In some cases it has been conjectured, as in [35]. In other cases, the problem is avoided by taking an equality that satisfies anti-symmetry by definition: $A = B$ is defined as $A \leq B$ and $B \leq A$. Steffen [33] has showed the simpler property, appropriate to his setting of polarized subtyping, that $A \leq B$ and $B \leq A$, where both derivations are *without uses of promotion*[1], if and only if $A =_\beta B$. However, in most higher-order subtyping calculi, including ours, the problem in showing anti-symmetry is exactly to show that the derivations of $A \leq B$ and $B \leq A$ contain no uses of promotion.

Anti-symmetry has been demonstrated for $F_{\leq}$ [20]. For Mitchell's second order λ-calculus a la Curry with subtyping, a completely different style of subtyping from the one we consider here, anti-symmetry has been studied under the name of equational axiomatization of bicoercibility [34]. There, A and B are called bicoercible if $A \leq B$ and $B \leq A$, and the paper proves that if A and B are bicoercible then $A \equiv B$, for an appropriate equivalence relation $\equiv$. However, the problem of bicoercibility of $F_{\leq}$ is considerably easier than anti-symmetry of higher-order subtyping, because there is no notion of computation on types, and in particular no β-reduction on types.

The rest of the paper is structured as follows. In the remaining sections of the introduction we discuss technical points relating to typed operational semantics and anti-symmetry of subtyping. In Section 2 we introduce the basic language of $\mathcal{F}^{\omega}_{\leq}$. In Section 3 we introduce the typed operational semantics, which as explained above plays a central role in our proof of anti-symmetry by providing a powerful induction principle. Section 4 outlines the proof of anti-symmetry using the typed operational semantics and sketches an approach to implementing subtyping and equality simultaneously. In Section 5 we give a sketch of the metatheory of a new presentation of $\mathcal{F}^{\omega}_{\leq}$ without judgemental equality or conversion. Finally, we draw conclusions in Section 6. The appendices contain an outline of the results that we use from our earlier development of typed operational semantics for $\mathcal{F}^{\omega}_{\leq}$, rules for the traditional presentation of $\mathcal{F}^{\omega}_{\leq}$, and rules for the typed operational semantics.

Typed Operational Semantics Our proof of anti-symmetry of higher-order subtyping relies on our understanding of subtyping built up using typed operational semantics. Typed operational semantics [22] gives an alternative induction principle for type theories, by presenting type theory operationally rather than declaratively. The typed operational semantics for $\mathcal{F}^{\omega}_{\leq}$ has judgements for reduction to weak-head and normal form for types, and for subtyping comparison between types in weak-head normal form and arbitrary types. Because the system is presented from the perspective of computation, many properties about

[1] Promotion is a step of transitivity along the bound of the head variable, and consists of replacing the head variable by its bound in a given context.

the relationship between reduction and typing are particularly easy to show by induction on derivations of the typed operational semantics. This topic is dealt with extensively in an earlier article [14].

A typed approach like typed operational semantics is essential to studying anti-symmetry. Similar to Church–Rosser for $\beta\eta$-reduction in type theories, which is only true for typed terms, anti-symmetry is only true for well-formed judgements. For example, in $Y{\leq}(\Lambda X{\leq}A{:}K.(X\ X)){:}K', Z{\leq}Y{:}K''$, an invalid context, we can show that $Z\ Z \leq Y\ Z$ and $Y\ Z \leq Z\ Z$, but the two types are clearly not β-equal.

A key property in our proof of anti-symmetry, that $\Gamma(X)(A_1, \ldots, A_n) \leq X(A_1, \ldots, A_n)$ is underivable in the context Γ (where $\Gamma(X)$ denotes the bound of X in Γ), also relies on the well-formedness of the judgement. Intuitively, if $\Gamma(X)(A_1, \ldots, A_n)$ and $X(A_1, \ldots, A_n)$ are well-formed then they can never be β-equal. While the base case, that $\Gamma(X) \leq X$ is impossible, is straightforward, the complication in the general case is that X may appear in some A_i, so it is not obvious that $\Gamma(X)(A_1, \ldots, A_m)$ may not β-reduce to $X(A_1, \ldots, A_m)$, for example. The type system rules out cases like $\Gamma(X)$ being $\Lambda Y{\leq}A{:}K.(Y\ Y)$ and A_1 being X.

Essential to our approach is the extended rule ST-TApp from our most recent papers on decidability of subtyping [14,15]. This rule contains as a premise the well-formedness of the bound $\Gamma(X)$ applied to a sequence of types, in order to conclude the well-formedness of the variable X applied to that sequence of terms. This rule is justified by an extension of the logical relation proof of Soundness (Theorem 3) using saturated sets. It is the powerful induction principle arising out of this rule, already crucial to our proof of decidability, that allows us to show the underivability of $\Gamma(X)(A_1, \ldots, A_m) \leq X(A_1, \ldots, A_m)$.

Induced Equivalence Relations The equivalence relation induced by $A \leq B$ and $B \leq A$ may be stronger than the usual intensional equality associated with type theory, syntactic equivalence on normal forms. One such case occurs in [32], where the types $\forall(X <: Bot)X{\to}X$ and $\forall(X <: Bot)Bot{\to}Bot$ are "equivalent in the subtype relation[2], even though they are not syntactically identical." A similar situation appears in intersection types disciplines, where $\top = A{\to}\top$ and also $A{\to}\top \leq \top$ and $\top \leq A{\to}\top$. A final example is extensible records [11], where the extension operator is associative and commutative in the subtyping relation. These equivalence relations have models in existing frameworks, for example game semantics [13] or PER models [17].

Such equivalences also arise in the context of programming languages. For example, consider object types with a private section and a public interface, where two object types O_1 and O_2 may satisfy $O_1 \leq O_2$ and $O_2 \leq O_1$ but differ in their private section. The equivalence relation that only considers the public interface will be more useful to the programmer than intensional equality, because it is more permissive.

[2] A and B are equivalent in the subtyping relation if $A \leq B$ and $B \leq A$.

In our current work on $\mathcal{F}^{\omega}_{\leq}$, the equivalence relation is the usual notion of β-equality at the type level. However, Martin-Löf [26] has demonstrated how to capture more sophisticated equivalence relations in intensional type theory. He allows the equality on elements of a type to include an arbitrary decidable equivalence relation on the normal forms of a type, rather than taking simple syntactic equivalence of normal forms. To extend our work to the example above of public and private sections of an object type, we can take an equivalence on object types in normal form that simply compares the fields of the public interfaces.

Metatheoretic Consequences of Anti-Symmetry In addition to providing the answer to a long-standing open problem, our work has several consequences in the development of the metatheory of type systems with subtyping. First, in the presence of anti-symmetry for subtyping, we can show the equivalence between traditional presentations, either with judgemental equality or untyped β-conversion, and a system without the notion of equality. This means that the proof of soundness for model constructions, such as that for proofs of strong normalization or PER models, can be more concise. In Section 5 we give a sketch of a new presentation of $\mathcal{F}^{\omega}_{\leq}$ without judgemental equality or conversion, apparently not found elsewhere in the literature, and discuss how the development of the metatheory proceeds for this system. The lack of a notion of equality may also have consequences for the implementation of type theories with subtyping.

Another consequence of this result is that we can now prove the Minimum Types Property, as opposed to the Minimal Types Property normally proved. A type inference algorithm can be shown to find one of many minimal types for a term. We can now clearly state the relationship between all of these minimal types: whereas before we knew that any two minimal types were subtypes of each other, we now know that they are β-equal.

2 Syntax

We now introduce the basic language of $\mathcal{F}^{\omega}_{\leq}$. A complete development of its meta-theory can be found in [15].

The kinds of $\mathcal{F}^{\omega}_{\leq}$ are the kind $\star$ of proper types and the kinds $\Pi X{\leq}A{:}K_1.K_2$ of functions on types, or type operators. The types of $\mathcal{F}^{\omega}_{\leq}$ are a straightforward higher-order extension of $F_{\leq}$, where we allow bounds on the abstraction $\Lambda X{\leq}A{:}K_1.B$. There is a top type $\mathrm{T}_{\star}$, and we define top type operators T_K at every kind K by $\mathrm{T}_{\Pi X \leq A_1:K_1.K_2} = \Lambda X{\leq}A_1{:}K_1.\mathrm{T}_{K_2}$. The language of terms is the same as that for $F_{\leq}$, with bounded type abstraction $\Lambda X{\leq}A{:}K.M$. As in $F_{\leq}$, each type variable is given an upper bound at the point where it is introduced.

The operational semantics of $\mathcal{F}^{\omega}_{\leq}$ is given by the usual β-reduction rules on terms and types, and is extended to a compatible relation with respect to term or type formation. We write $\twoheadrightarrow_{\beta}$ for the transitive and reflexive closure of $\rightarrow_{\beta}$, and $=_{\beta}$ for the least equivalence relation containing $\rightarrow_{\beta}$ and closed under α-

equivalence. We write A^{nf} to indicate the β-normal form of A, and $A \equiv B$ when A and B are α-equivalent.

Weak head reduction, or leftmost outermost reduction, is a less familiar notion from lambda calculus that appears in our presentation.

For technical reasons relating to the model construction, we need to consider a slightly stronger notion of weak-head normal form.

Definition 1 (Weak-Head Normal). *The types* $\mathrm{T}_\star$, $A_1 \to A_2$, $\forall X \leq A{:}K.B$, *and* $\Lambda X \leq A{:}K.B$ *are weak-head normal.* $X(A_1, \ldots, A_m)$ *is weak-head normal if* $A_1, \ldots, A_m$ *are in normal form.*

Contexts are defined as usual, where the empty context is written $\emptyset$, term variable bindings have the form $x{:}A$, and type variable bindings have the form $X \leq A{:}K$. We write $\mathrm{dom}(\Gamma)$ for the set of term and type variables defined in a context Γ. The sets of free term and type variables occurring in terms, types, kinds, contexts or judgements are written $\mathrm{FV}(\text{—})$ and $\mathrm{FTV}(\text{—})$. Since we are careful to ensure that no variable is bound more than once, we sometimes consider contexts as finite functions: $\Gamma(X)$ yields the bound of X in Γ, where $X \in \mathrm{dom}(\Gamma)$ is implicitly asserted.

We write $A(B_1, \ldots, B_n)$ for $((A\,B_1)\ldots B_n)$. If A is of the form $X(B_1, \ldots, B_n)$ then A has head variable X. We write $\mathrm{HV}(\text{—})$ for the partial function returning the head variable of a term. We write $B[X \leftarrow A]$ for the capture-avoiding substitution of A for X in B. We identify types that differ only in the names of bound variables.

The system $\mathcal{F}^{\omega}_{\leq}$ is presented as simultaneously defined inductive relations with the following judgement forms:

$\Gamma \vdash \mathrm{ok}$	well-formed context	$\Gamma \vdash K$	well-formed kind
$\Gamma \vdash K =_\beta K'$	kind equality	$\Gamma \vdash A : K$	well-kinded type
$\Gamma \vdash A =_\beta B : K$	type equality	$\Gamma \vdash A \leq B : K$	subtype
$\Gamma \vdash M : A$	well-typed term.		

We sometimes use the metavariable J to range over statements (right-hand sides of judgements) of any of these judgement forms.

We now give an overview of the rules of inference for $\mathcal{F}^{\omega}_{\leq}$. The context formation rules are as usual in $F^{\omega}_{<:}$. Kind formation differs by incorporating information about the bounds in Π:

$$\frac{\Gamma,\, X \leq A{:}K_1 \vdash K_2}{\Gamma \vdash \Pi X \leq A{:}K_1.K_2} \qquad (\text{K-}\Pi)$$

The rules of inference for kind equality simply define a typed equivalence relation compatible with respect to the kind formers.

The rules for type formation similarly need to be adjusted for bounded operator abstraction.

$$\frac{\Gamma,\, X \leq A_1{:}K_1 \vdash A_2 : K_2}{\Gamma \vdash \Lambda X \leq A_1{:}K_1.A_2 : \Pi X \leq A_1{:}K_1.K_2} \qquad (\text{T-TA}_{\text{BS}})$$

$$\frac{\Gamma \vdash A : \Pi X \leq B{:}K_1.K_2 \quad \Gamma \vdash C \leq B : K_1}{\Gamma \vdash A\,C : K_2[X \leftarrow C]} \qquad (\text{T-TA}_{\text{PP}})$$

$$\frac{\Gamma \vdash A : K \qquad \Gamma \vdash K =_\beta K'}{\Gamma \vdash A : K'} \tag{T-Conv}$$

Type equality is defined as the typed equivalence relation compatible with respect to the type formers and closed under β-reduction for types.

$$\frac{\Gamma,\, X{\leq}A_1{:}K_1 \vdash A_2 : K_2 \qquad \Gamma \vdash C \leq A_1 : K_1}{\Gamma \vdash (\Lambda X{\leq}A_1{:}K_1.A_2)\, C =_\beta A_2[X{\leftarrow}C] : K_2[X{\leftarrow}C]} \tag{T-Eq-Beta}$$

The type equality rules appear in Appendix B.

The subtyping rules are again those of $F^\omega_{<:}$ [9,10,27], except for those dealing with bounded type abstraction and type application and the rule for subtyping the quantifier. We chose Cardelli and Wegner's kernel Fun rule for quantifiers with equal bounds [12], because the contravariant rule for quantifiers renders the system undecidable [31]. Furthermore, transitivity elimination in the presence of such a rule in the higher-order case remains an open problem. Type equality is included in subtyping.

$$\frac{\Gamma \vdash A =_\beta B : K}{\Gamma \vdash A \leq B : K} \tag{S-Conv}$$

The subtyping rules also appear in Appendix B. Our goal is to prove that this relation is anti-symmetric up to β-equality.

The term formation rules are standard. We remind the reader that the subtyping relation on $\star$ induces an inclusion over types.

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \leq B : \star}{\Gamma \vdash M : B} \tag{T-Sub}$$

3 The Typed Operational Semantics

The typed operational semantics for $\mathcal{F}^\omega_{\leq}$ is organized in five judgement forms:

$\Gamma \vdash_S$ ok	valid context	$\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} B \;{}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} C : K$	type reduction
$\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$	kind reduction	$\Gamma \vdash_S A \leq B : K$	subtyping
$\Gamma \vdash_S A \leq_W B : K$	weak-head subtyping.		

This system can be understood informally as a particularly informative algorithm for kinding and subtyping in $\mathcal{F}^\omega_{\leq}$. The first judgement simply represents well-formed contexts. The second and third judgements represent normalization of kinds and types, where the formulation of the rules of inference ensures strong normalization, Church–Rosser and other good metatheoretic properties. The last two judgements represent an algorithm for subtyping that first reduces the left and right-hand sides to weak-head normal forms and then compares these.

We use various notations that omit unnecessary components of the judgement. For example, we write $\Gamma \vdash_S K$ for $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$ for some K', and similarly for types, $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} B : K$ for $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} B \;{}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} C : K$ for some C, and $\Gamma \vdash_S A \;{}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} B : K$ for $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} A \twoheadrightarrow_{\mathrm{n}} B : K$. We also write $\Gamma \vdash_S K, K' \twoheadrightarrow_{\mathrm{n}} K''$ when $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K''$ and $\Gamma \vdash_S K' \twoheadrightarrow_{\mathrm{n}} K''$, and similarly for types.

We now present the rules of inference. The context formation and kind normalization rules are modifications of the corresponding rules for $\mathcal{F}^{\omega}_{\leq}$. The rules for type reduction combine kinding information and computational behavior in the form of weak-head and β-normal forms. For example, the rule for arrow types says how to obtain the weak-head and β-normal forms of $A_1 \to A_2$ in $\star$ from those for A_1 and A_2 in $\star$.

Kind reduction depends on reduction on types. This is not the case in Cardelli's $F^{\omega}_{<:}$ or in $Ob_{\omega\leq:}$[1], where kinds are syntactically simpler.

$$\frac{\Gamma \vdash_S K_1 \twoheadrightarrow_{\mathrm{n}} K_1' \qquad \Gamma \vdash_S A \twoheadrightarrow_{\mathrm{n}} B : K_1' \qquad \Gamma,\, X{\leq}A : K_1 \vdash_S K_2 \twoheadrightarrow_{\mathrm{n}} K_2'}{\Gamma \vdash_S \Pi X{\leq}A{:}K_1.K_2 \twoheadrightarrow_{\mathrm{n}} \Pi X{\leq}B{:}K_1'.K_2'} \quad (\text{SK-}\Pi)$$

The rules for type reduction are in Appendix C.

The beta rule, besides uncovering the outermost redex of the application $B\,C$ and contracting it, finds the weak-head normal form E and the normal form F. The premise $\Gamma \vdash_S K_2[X{\leftarrow}C] \twoheadrightarrow_{\mathrm{w}} K$ ensures that E and F have β-equal kinds, and the subtyping premise $\Gamma \vdash_S C \leq A : K_1{}'$ enforces the well-formation of $B\,C$.

$$\frac{\begin{array}{l}\Gamma \vdash_S B \twoheadrightarrow_{\mathrm{w}} \Lambda X{\leq}A{:}K_1.D : \Pi X{\leq}A'{:}K_1'.K_2 \qquad \Gamma \vdash_S K_2[X{\leftarrow}C] \twoheadrightarrow_{\mathrm{n}} K \\ \Gamma \vdash_S D[X{\leftarrow}C] \twoheadrightarrow_{\mathrm{w}} E \;{}_{\mathrm{w}}{\twoheadrightarrow_{\mathrm{n}}}\; F : K \qquad \Gamma \vdash_S C \leq A : K_1'\end{array}}{\Gamma \vdash_S B\,C \twoheadrightarrow_{\mathrm{w}} E \;{}_{\mathrm{w}}{\twoheadrightarrow_{\mathrm{n}}}\; F : K} \quad (\text{ST-Beta})$$

The weak-head subtyping rules are motivated by the algorithmic rules in [16]. The rules SWS-Arrow, SWS-All, and SWS-TAbs are structural. The rule SWS-TApp implicitly uses transitivity, reducing the problem of a variable being less than another type to the problem of the bound of the variable being less than that type. The side condition ensures determinism. The rules for weak-head subtyping are in Appendix C.

Finally, full subtyping is defined by reference to the weak-head subtyping relation.

$$\frac{\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} C : K \quad \Gamma \vdash_S B \twoheadrightarrow_{\mathrm{w}} D : K \quad \Gamma \vdash_S C \leq_W D : K}{\Gamma \vdash_S A \leq B : K} \quad (\text{SS-Inc})$$

We have developed extensive results for this system in [15]. Those relevant to our development here, including the equivalence of the original system and the typed operational semantics, are summarized in Appendix A.

4 Anti-Symmetry

Our goal is to prove that if $\Gamma \vdash A \leq B : K$ and $\Gamma \vdash B \leq A : K$ then $\Gamma \vdash A =_\beta B : K$.

We obtain this (Proposition 7) as a consequence of the corresponding property in the typed operational semantics using Soundness (Theorem 3) and Completeness (Proposition 2).[3] In the semantics, the way to say that two types are equal is that they have the same normal form.

[3] Our choice of terminology for Soundness and Completeness comes from proofs of strong normalization using saturated sets, where the Soundness theorem says that everything derivable in the syntax is satisfied in the saturated sets model.

The main difficulty appears when A is of the form $X(A_1, \ldots, A_m)$. Intuitively, in this case B can only be $X(A_1, \ldots, A_m)$. To discard the possibility that B could be other than A, we need to prove that $\Gamma \vdash_S \Gamma(X)(A_1, \ldots, A_m) \leq X(A_1, \ldots, A_m) : K$ is impossible.

4.1 Basic Properties

In this section we establish some preliminary lemmas.

The typed operational semantics is deterministic (Lemma 13), so there is only one derivation for each judgement, and we can invert the rules deriving premises from conclusions. We sometimes use this fact without mentioning it.

Lemma 1. *Suppose* $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{n}} B : \Pi X{\leq}E{:}K_1.K_2$, $\Gamma \vdash_S C \leq E : K_1$ *and* $\Gamma \vdash_S K_2[X{\leftarrow}C] \twoheadrightarrow_{\mathrm{n}} K_2'$. *Then there is an* F *such that* $\Gamma \vdash_S A\,C \twoheadrightarrow_{\mathrm{n}} F : K_2'$ *and* $\Gamma \vdash_S B\,C \twoheadrightarrow_{\mathrm{n}} F : K_2'$.

Proof. By Completeness, equational reasoning and Soundness. □

As shown in [15], if $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} B \;{}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} C : K$ then B is the weak-head normal form of A and C the normal form of A. Therefore, the following structural property of the typed operational semantics can be read as follows: if B is the weak-head normal form of A, then B is its own weak-head normal form, and if C is the normal form of A, then C has itself as weak-head and normal forms.

Lemma 2.

1. *If* $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} B \;{}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} C : K$ *then* $\Gamma \vdash_S B \twoheadrightarrow_{\mathrm{w}} B \;{}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} C : K$ *and* $\Gamma \vdash_S C \twoheadrightarrow_{\mathrm{w}} C \;{}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} C : K$.
2. *If* $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$ *then* $\Gamma \vdash_S K' \twoheadrightarrow_{\mathrm{n}} K'$.

Proof. By simultaneous induction on derivations, where most cases are immediate or follow by the induction hypothesis. The interesting case, ST-TApp, follows by the induction hypothesis and Subtyping Conversion.

Lemma 3 (Upper Bound).

1. *If* $\Gamma \vdash_S X(A_1, \ldots, A_m) : K$ *then* $\Gamma \vdash_S \Gamma(X)(A_1, \ldots, A_m) : K$.
2. *If* $\Gamma \vdash_S X(A_1, \ldots, A_m) \leq B : K$ *and* $B \not\equiv X(A_1, \ldots, A_m)$ *then* $\Gamma \vdash_S \Gamma(X)(A_1, \ldots, A_m) \leq B : K$.
3. *If* $\Gamma \vdash_S X(A_1, \ldots, A_m) \leq_W B : K$ *and* $B \not\equiv X(A_1, \ldots, A_m)$ *then* $\Gamma \vdash_S \Gamma(X)(A_1, \ldots, A_m) \leq B : K$.

Proof. Case 1 follows by Soundness and Completeness from Lemma 16, because $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K$. Cases 2 and 3 follow by simultaneous induction on derivations. □

4.2 An Impossible Judgement

In this section we show that $\Gamma \vdash_S \Gamma(X)(A_1 \dots A_m) \leq X(A_1 \dots A_m) : K$ is an underivable judgement.

The particular property that drives our proof of this result is that the typing rule ST-TApp in the typed operational semantics, for variables applied to a sequence of types, has a subderivation stating the well-formedness of the type resulting from the replacement of the variable with its bound. This rule of inference is justified by our earlier proofs of Soundness and Completeness of the usual rules of inference for the typed operational semantics, published elsewhere [15].

The development of the full metatheory of typed operational semantics is quite complex, and in particular the proof of Soundness is similar to proofs of strong normalization and relies on logical relations. However, once we have established the equivalence of the original presentation and the typed operational semantics, we can define a measure, from derivations to natural numbers, that counts the number of uses of promotion (replacing a variable by its bound in the operational semantics) before reaching Top. Clearly, the number of uses of the rule SWS-TApp in a derivation of $\Gamma \vdash_S X(A_1, \dots, A_n) \leq \mathrm{T} : K$ must be greater than the number of uses in a derivation of $\Gamma \vdash_S \Gamma(X)(A_1, \dots, A_n) \leq \mathrm{T} : K$, because the latter is a subderivation of the former. On the other hand, we can also show that if $\Gamma \vdash_S A \leq B : K$ then the number of uses of the rule SWS-TApp in $\Gamma \vdash_S A \leq \mathrm{T} : K$ is greater than or equal to the number in $\Gamma \vdash_S B \leq \mathrm{T} : K$. Hence, if we have a derivation of $\Gamma \vdash_S \Gamma(X)(A_1, \dots, A_n) \leq X(A_1, \dots, A_n) : K$ then we can derive a contradiction.

We introduce a function from derivations in the typed operational semantics to numbers, informally capturing the number of uses of the rule of inference that a variable is less than its bound. We do not have a good notation for defining a function on derivations because such definitions do not occur commonly in the literature. We therefore abbreviate in the following definition $\Gamma(X)$ for the subderivation of the bound of X in Γ for the case ST-TVar; $\Gamma(X)^{\mathrm{nf}}(A_1, \dots, A_m)$ for the subderivation of the normal form of the bound applied to the arguments of X for the case ST-TApp; and β for a derivation of $B\,C$ using ST-Beta and *reduct* for the subderivation of the β-reduct of $B\,C$.

Definition 2. *We define* $\sharp(-)$*, from derivations of* $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} B \;{}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} C : K$ *to numbers, by induction on derivations:*

ST-Top	$\sharp(\mathrm{T}_\star) = 0$	ST-TApp	$\sharp(X(A_1, \dots, A_m)) =$
ST-Arrow	$\sharp(A_1 \to A_2) = 0$		$\sharp(\Gamma(X)^{\mathrm{nf}}(A_1, \dots, A_m)) + 1$
ST-All	$\sharp(\forall X \leq A_1 : K.A_2) = 0$	ST-TVar	$\sharp(X) = \sharp(\Gamma(X)) + 1$
ST-TAbs	$\sharp(\Lambda X \leq A_1 : K.A_2) = 0$	ST-Beta	$\sharp(\beta) = \sharp(\mathit{reduct})$

Notice that if $\Gamma \vdash_S A \leq B : K$ then $\Gamma \vdash_S A : K$ and $\Gamma \vdash_S B : K$ by Lemma 17, so $\sharp(A)$ and $\sharp(B)$ are defined.

We now show that this length function is invariant with respect to well-formed types or type-operators that have the same normal form. This lemma

justifies our later informality in writing types or type-operators in place of their derivations of well-formedness.

Lemma 4. *If $\mathcal{D}$ is a derivation of $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} C \ {}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} E : K$ and $\mathcal{D}'$ is a derivation of $\Gamma \vdash_S B \twoheadrightarrow_{\mathrm{w}} D \ {}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} E : K$ then $\sharp(\mathcal{D}) = \sharp(\mathcal{D}')$.*

Proof. By induction on derivations $\mathcal{D}$.

In each case where E is a type constructor or a variable, we perform a nested induction on $\mathcal{D}'$, and there are two interesting cases. The first is when the nested case is the same rule as for the outermost induction. In this case the lengths are equal by assumption for rules ST-Top, ST-Arrow, ST-All and ST-TAbs. For ST-TVar and ST-TApp, we know $\Gamma(X)$ is a function, so the result follows by Determinacy and the induction hypothesis. In the case ST-TApp we have that $\Gamma \vdash_S A'_1\, A'_2 \twoheadrightarrow_{\mathrm{w}} X(A_1, \ldots, A_m, F) \ {}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} X(A_1, \ldots, A_m, F) : K$ and $\Gamma \vdash_S B_1\, B_2 \twoheadrightarrow_{\mathrm{w}} X(A_1, \ldots, A_m, F) \ {}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} X(A_1, \ldots, A_m, F) : K$. Then $\mathcal{D}$ and $\mathcal{D}'$ have the same subderivation $\Gamma \vdash_S D(A_1, \ldots, A_m, F) : K$. Hence, by the definition of $\sharp(-)$, $\sharp(\mathcal{D}) = \sharp(D(A_1, \ldots, A_m)) + 1 = \sharp(\mathcal{D}')$.

The second interesting case in the nested inductions is ST-Beta, where the result follows by the induction hypothesis. All of the other nested cases contradict the assumption that the normal forms are the same.

The final outermost case, ST-Beta, follows by the induction hypothesis. □

Then, if $\Gamma \vdash_S X(A_1, \ldots, A_m) : K$ we have

$$\begin{aligned}\sharp(X(A_1, \ldots, A_m)) &= \sharp(\Gamma(X)^{\mathrm{nf}}(A_1, \ldots, A_m)) + 1 \\ &= \sharp(\Gamma(X)(A_1, \ldots, A_m)) + 1 > \sharp(\Gamma(X)(A_1, \ldots, A_m)),\end{aligned}$$

where by Lemma 1 there is a B such that $\Gamma \vdash_S \Gamma(X)^{\mathrm{nf}}(A_1, \ldots, A_m) \twoheadrightarrow_{\mathrm{n}} B : K$ and $\Gamma \vdash_S \Gamma(X)(A_1, \ldots, A_m) \twoheadrightarrow_{\mathrm{n}} B : K$, and by Lemma 4 they have the same length.

Now, we come to the main lemma about $\sharp(A)$:

Lemma 5.

1. *If $\Gamma \vdash_S A \leq B : K$ then $\sharp(A) \geq \sharp(B)$.*
2. *If $\Gamma \vdash_S A \leq_W B : K$ then $\sharp(A) \geq \sharp(B)$.*

Proof. By simultaneous induction on derivations. In Case 1, the only possible rule for $\Gamma \vdash_S A \leq B : K$ is SS-Inc, which follows by Determinacy, Lemma 4, and the induction hypothesis. In Case 2 the only interesting case is SWS-TApp, where by the induction hypothesis $\sharp(E) \geq \sharp(A)$, so

$\sharp(X((A_1, \ldots, A_m)) = \sharp(\Gamma(X)^{\mathrm{nf}}(A_1, \ldots, A_m)) + 1 = \sharp(E) + 1 > \sharp(E) \geq \sharp(A)$, where we know that $\sharp(\Gamma(X)^{\mathrm{nf}}(A_1, \ldots, A_m)) = \sharp(E)$ by Determinacy. □

Lemma 6. *There can be no derivation of the judgement*
$\Gamma \vdash_S \Gamma(X)(A_1, \ldots, A_m) \leq X(A_1, \ldots, A_m) : K$.

Proof. Suppose that there were a derivation of $\Gamma \vdash_S \Gamma(X)(A_1, \ldots, A_m) \leq X(A_1, \ldots, A_m) : K$. By Lemma 5 we have
$\sharp(\Gamma(X)(A_1, \ldots, A_m)) \geq \sharp(X(A_1, \ldots, A_m))$, contradicting the fact that
$\sharp(X(A_1, \ldots, A_m)) > \sharp(\Gamma(X)(A_1, \ldots, A_m))$. □

Based on our understanding of the behavior of bounds in Lemma 3, the negative result Lemma 6 seems intuitive and believable. However, the results in Lemma 3 have been known for systems of higher-order subtyping, while Lemma 6 has remained a conjecture, so this was the major challenge of our development.

4.3 Main Result

We can now prove our main result. Observe that in the case SWS-TApp we use our key lemma (Lemma 6).

Lemma 7 (Anti-Symmetry of TOS).

1. *If* $\Gamma \vdash_S A \leq B : K$ *and* $\Gamma \vdash_S B \leq A : K$ *then* $\Gamma \vdash_S A, B \twoheadrightarrow_{\mathrm{n}} C : K$.
2. *If* $\Gamma \vdash_S A \leq_W B : K$ *and* $\Gamma \vdash_S B \leq_W A : K$ *then* $\Gamma \vdash_S A, B \twoheadrightarrow_{\mathrm{n}} C : K$.

Proof. The argument is by simultaneous induction on derivations.

1. The only rule to derive both assumptions is SS-Inc. By Determinacy, the premises are $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} C \ {}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} E : K$, $\Gamma \vdash_S B \twoheadrightarrow_{\mathrm{w}} D \ {}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} F : K$, $\Gamma \vdash_S C \leq_W D : K$, and $\Gamma \vdash_S D \leq_W C : K$. By Lemma 2 $\Gamma \vdash_S C \twoheadrightarrow_{\mathrm{w}} C \ {}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} E : K$ and $\Gamma \vdash_S D \twoheadrightarrow_{\mathrm{w}} D \ {}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} F : K$. By the induction hypothesis $\Gamma \vdash_S C \ {}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} G : K$ and $\Gamma \vdash_S D \ {}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} G : K$. Finally, $E \equiv F \equiv G$ by Determinacy.
2. The proof is by induction on the derivation of $\Gamma \vdash_S A \leq_W B : K$.
   - SWS-Top Then $B \equiv \mathrm{T}_\star$, and $\mathrm{HV}(A)$ undefined, which means that A is not a type application or a type variable. Therefore the only rule to derive $\Gamma \vdash_S \mathrm{T}_\star \leq_W A : K$ is SWS-Top, which means that $A \equiv \mathrm{T}_\star$. By Lemma 17 and Determinacy, $\Gamma \vdash_S \mathrm{T}_\star \twoheadrightarrow_{\mathrm{w}} \mathrm{T}_\star \ {}_{\mathrm{w}}\!\!\twoheadrightarrow_{\mathrm{n}} \mathrm{T}_\star : \star$.
   - SWS-TApp We are going to show that this case is not possible. Assume that it is. We have that $A \equiv X(A_1, \ldots, A_m)$ and $B \not\equiv X(A_1, \ldots, A_m)$. By Upper Bound (Lemma 3), $\Gamma \vdash_S \Gamma(X)(A_1, \ldots, A_m) \leq B : K$, and by transitivity $\Gamma \vdash_S \Gamma(X)(A_1, \ldots, A_m) \leq X(A_1, \ldots, A_m) : K$, which is a contradiction, by Lemma 6.
   - SWS-Refl By the premise.
   - SWS-Arrow The only rule to derive $\Gamma \vdash_S B \leq_W A : K$ is also SWS-Arrow, so the result follows by the induction hypothesis and ST-Arrow.
   - SWS-All The only rule to derive $\Gamma \vdash_S B \leq_W A : K$ is also SWS-All. The premises are, $\Gamma, X{\leq}A_1{:}K \vdash_S A_2 \leq B_2 : \star$, $\Gamma, X{\leq}B_1{:}K' \vdash_S B_2 \leq A_2 : \star$, $\Gamma \vdash_S A_1, B_1 \twoheadrightarrow_{\mathrm{n}} C_1 : K''$, and $\Gamma \vdash_S K, K' \twoheadrightarrow_{\mathrm{n}} K''$. By Context Conversion (Lemma 12), $\Gamma, X{\leq}A_1{:}K \vdash_S B_2 \leq A_2 : \star$. We can now apply the induction hypothesis to obtain $\Gamma, X{\leq}A_1{:}K \vdash_S A_2, B_2 \twoheadrightarrow_{\mathrm{n}} C_2 : \star$. Finally, $\Gamma \vdash_S (\forall X{\leq}A_1{:}K.A_2), (\forall X{\leq}B_1{:}K'.B_2) \twoheadrightarrow_{\mathrm{n}} \forall X{\leq}C_1{:}K''.C_2 : \star$, by ST-All.

SWS-TABS Similar to case SWS-ALL. □

Theorem 1 (Anti-Symmetry). *If* $\Gamma \vdash A \leq B : K$ *and* $\Gamma \vdash B \leq A : K$ *then* $\Gamma \vdash A =_\beta B : K$.

Proof. By Soundness (Theorem 3), $\Gamma \vdash_S A \leq B : K'$, $\Gamma \vdash_S B \leq A : K'$, and $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$. By Proposition 7, $\Gamma \vdash_S A, B \twoheadrightarrow_{\mathrm{n}} C : K'$. By Completeness and symmetry of kind equality $\Gamma \vdash K' =_\beta K$. By Completeness $\Gamma \vdash A =_\beta C : K'$ and $\Gamma \vdash B =_\beta C : K'$. Finally, by T-EQ-SYM, T-EQ-TRANS, and S-K-CONV, $\Gamma \vdash A =_\beta B : K$. □

4.4 Equality by Subtyping

A consequence of the proof of Anti-Symmetry of the typed operational semantics is that if $\Gamma \vdash_S A \leq B : K$ and $\Gamma \vdash_S B \leq A : K$ then the derivations do not contain any uses of the promotion rule SWS-TAPP. This fact is used to show that:

Lemma 8. *If* $\Gamma \vdash A =_\beta B : K$ *then the derivation of* $\Gamma \vdash_S A \leq B : K'$ *does not contain any uses of* SWS-TAPP*, where* $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$.

Proof. By Completeness and by the proof of Anti-Symmetry for the typed operational semantics. □

Furthermore, if one of the subtyping derivations does not use SWS-TAPP then the other derivation does not either.

Lemma 9.

1. *If it is not the case that* $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} C : K'$, $C \not\equiv \mathrm{T}_\star$ *and* $\Gamma \vdash_S B \twoheadrightarrow_{\mathrm{w}} \mathrm{T}_\star : K'$ *then if the derivation of* $\Gamma \vdash_S A \leq B : K'$ *contains no uses of* SWS-TAPP*, then* $\Gamma \vdash_S B \leq A : K'$ *and the derivation contains no uses of* SWS-TAPP.
2. *If it is not the case that* $A \not\equiv \mathrm{T}_\star$ *and* $B \equiv \mathrm{T}_\star$ *then if the derivation of* $\Gamma \vdash_S A \leq_W B : K'$ *contains no uses of* SWS-TAPP*, then* $\Gamma \vdash_S B \leq_W A : K'$ *and the derivation contains no uses of* SWS-TAPP.

Proof. By simultaneous induction on the derivations of $\Gamma \vdash_S A \leq B : K'$ and $\Gamma \vdash_S A \leq_W B : K'$, using that Context Conversion (Lemma 12) creates no new uses of SWS-TAPP in the cases SWS-ALL and SWS-TABS.

In the light of Lemmas 8 and 9, an algorithm may implement equality and subtyping simultaneously: to check if $A =_\beta B$ it is not necessary to check both $A \leq B$ and $B \leq A$, but is enough to check whether the algorithm uses a step corresponding to promotion (the application rule) in showing $A \leq B$. If it does not then $A =_\beta B$.

The only exception is the case in which the derivation does not contain promotion and is of the form:

$$\frac{A \twoheadrightarrow_{\mathrm{w}} C \qquad C \not\equiv \mathrm{T}_\star \qquad B \twoheadrightarrow_{\mathrm{w}} \mathrm{T}_\star \qquad C \leq_W \mathrm{T}_\star}{A \leq B}$$

where $C \leq_W \mathrm{T}_\star$ is obtained by the rule SWS-TOP. This is the only case in which a subtyping derivation not containing promotion relates two types which are not β-equal. To exclude this case Lemma 9 Case 1 has the added restrictions on the normal forms of A and B.

5 Replacing Equality with Subtyping

In this section we sketch a presentation of $\mathcal{F}^{\omega}_{\leqslant}$ without judgemental equality or conversion: we simply add β-expansion to the left- and right-hand sides of the subtyping judgement, and allow the rules for transitivity and compatible closure of subtyping to handle the extension to full β-equality. To our knowledge, this is the first such presentation for a higher-order subtyping calculus. This leads to simplifications in the proof of soundness for models of the system, because we remove equality and all of the rules of inference that rely on it.

We write $\Gamma \vdash_{\leqslant} J$ for judgements derived in the system without an equality judgement, and $\Gamma \vdash_{\leqslant} A \leqslant B : K$ if $\Gamma \vdash_{\leqslant} A \leq B : K$ and $\Gamma \vdash_{\leqslant} B \leq A : K$.

5.1 Modifications to Inference Rules

The system requires several changes in order to accommodate the removal of equality. First, the inclusion of the equality relation in subtyping, and in particular reflexivity and the rule for β-equality, needs to be recovered in the rules for subtyping itself:

$$\frac{\Gamma \vdash_{\leqslant} A : K}{\Gamma \vdash_{\leqslant} A \leq A : K} \qquad (\textsc{se-Refl})$$

$$\frac{\Gamma,\, X{\leq}A_1{:}K_1 \vdash_{\leqslant} A_2 : K_2 \quad \Gamma \vdash_{\leqslant} C \leq A_1 : K_1 \quad \Gamma \vdash_{\leqslant} A_2[X{\leftarrow}C] \leq D : K_2[X{\leftarrow}C]}{\Gamma \vdash_{\leqslant} (\Lambda X{\leq}A_1{:}K_1.A_2)\, C \leq D : K_2[X{\leftarrow}C]} \qquad (\textsc{se-BetaL})$$

The rule SE-BetaR similarly introduces a β-expansion on the right-hand side.

Next, the rule for subtyping applications does not allow the argument to vary. This restriction is for a good reason: allowing subtyping in the argument is unsound. However, we need to recapture the behavior of equality in allowing equal types on the right-hand side of an application:

$$\frac{\Gamma \vdash_{\leqslant} A \leq B : \Pi X{\leq}E{:}K_1.K_2 \qquad \Gamma \vdash_{\leqslant} C \leqslant D : K_1 \quad \Gamma \vdash_{\leqslant} C \leq E : K_1}{\Gamma \vdash_{\leqslant} A\, C \leq B\, D : K_2[X{\leftarrow}C]} \qquad (\textsc{se-TApp})$$

Showing the soundness of this rule for the system with the usual rule S-TApp will need to use anti-symmetry somewhere, because S-TApp requires β-equality of the arguments of the type application.

Finally, we still need an equivalence relation on kinds in order to allow the kinded judgements to vary with respect to equal kinds. We simply lift the equivalence induced by subtyping to kinds. Otherwise, the rules for the new system arise by replacing equality by the new relation $\leqslant$.

5.2 Changes to Metatheory

We can now show several basic results relating the old presentation with judgemental equality and the new one without it.

First, we can show that equality can be captured by the equivalence relation induced by subtyping.

Definition 3. *Let $\Gamma \vdash_{\leqslant} A = B : K$ be defined using the same rules of inference as judgemental equality for $\Gamma \vdash A = B : K$, but replacing uses of $\Gamma \vdash A : K$, $\Gamma \vdash A \leq B : K$, and so on by their corresponding judgement $\Gamma \vdash_{\leqslant} A : K$, $\Gamma \vdash_{\leqslant} A \leq B : K$ and so on.*

Lemma 10. *If $\Gamma \vdash_{\leqslant} A = B : K$ then $\Gamma \vdash_{\leqslant} A \leqslant B : K$.*

This lemma, plus some minor modifications to the original proof, leads to a proof of Completeness of the new system $\Gamma \vdash_{\leqslant} J$ for the typed operational semantics.

Proposition 1 (Completeness).

1. *$\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} B \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} C : K$ implies $\Gamma \vdash_{\leqslant} A : K$, $\Gamma \vdash_{\leqslant} A =_\beta B : K$ and $\Gamma \vdash_{\leqslant} A =_\beta C : K$.*
2. *$\Gamma \vdash_S A \leq B : K$ implies $\Gamma \vdash_{\leqslant} A, B : K$ and $\Gamma \vdash_{\leqslant} A \leq B : K$.*

We now consider Soundness of $\Gamma \vdash_{\leqslant} J$ for the typed operational semantics.

Theorem 2 (Soundness).

1. *If $\Gamma \vdash_{\leqslant} A : K$ then there are K', B and C such that $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$ and $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} B \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} C : K'$.*
2. *If $\Gamma \vdash_{\leqslant} A =_\beta B : K$ then there are C and K' such that $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$, $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{n}} C : K'$ and $\Gamma \vdash_S B \twoheadrightarrow_{\mathrm{n}} C : K'$.*
3. *If $\Gamma \vdash_{\leqslant} A \leq B : K$ then there is a K' such that $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$ and $\Gamma \vdash_S A \leq B : K'$.*

It is probably true that the alternative presentation without equality could be developed by itself, with no reference to equality or anti-symmetry. In our setting, this would require a small change in the rule SWS-Refl, to allow type arguments that are subtypes of each other on the left- and right-hand sides rather than the same normal form. However, the proof of the equivalence of this system with the traditional presentation with equality relies on anti-symmetry.

6 Conclusions

We have solved a long-standing open problem, giving a general approach to showing the anti-symmetry of higher-order subtyping. To our knowledge, this is the first proof of anti-symmetry for higher-order subtyping: even for systems like $F^{\omega}_{<:}$, defined roughly ten years ago, the result was unknown. We have also showed this result for the subtyping relation of $\mathcal{F}^{\omega}_{\leq}$, a higher-order lambda calculus with bounded operator abstraction. Typed operational semantics was essential to our proof, especially the refined understanding of the behavior of types as embodied in the extended rule ST-TApp.

This result has several consequences for the metatheory of type systems with higher-order subtyping. First, it implies that we can now prove the Minimum Types Property, as opposed to the Minimal Types Property which is normally proved. For our system, we can now say what is the relation between all the minimal types of a given term; before we knew that any two minimal types were subtypes of each other, we can now say that they are β-equal. Secondly, we can simplify the basic judgements of type systems with subtyping by eliminating either judgemental equality or conversion.

A practical consequence of this work is that the implementation of higher-order type systems with subtyping can be simplified, because we no longer need to implement either judgemental equality or conversion.

Acknowledgments

We would like to thank Benjamin Pierce and Juliusz Chroboczek for interesting discussions to type equality and anti-symmetry, Giorgio Ghelli for the reference about anti-symmetry for $F_{\leq}$, and Jan Zwanenburg for comments on an earlier version. We also thank the anonymous referees for their suggestions.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.
3. D. Aspinall and A. Compagnoni. Subtyping dependent types. In *Eleventh Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA*, July 1996. Long version to appear in Theoretical Computer Science.
4. H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.
5. V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.
6. K. B. Bruce. Typing in object-oriented languages: Achieving expressiveness and safety. Unpublished, June 1996.
7. K. B. Bruce and G. Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990.
8. L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.
9. L. Cardelli. Notes about $F^{\omega}_{\leq:}$. Unpublished manuscript, Oct. 1990.
10. L. Cardelli and G. Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, Oct. 1991.
11. L. Cardelli and J. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in [23], and available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.
12. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), Dec. 1985.

13. J. Chroboczek. *Game Semantics and Subtyping.* PhD thesis, University of Edinburgh, 1999. Forthcoming Thesis. Supervisor: Prof. S. Abramsky.
14. A. Compagnoni and H. Goguen. Decidability of higher-order subtyping via logical relations, Dec. 1997.
15. A. Compagnoni and H. Goguen. Typed operational semantics for higher order subtyping. Technical Report ECS-LFCS-97-361, University of Edinburgh, July 1997. Submitted to *Information and Computation.*
16. A. B. Compagnoni. *Higher-Order Subtyping with Intersection Types.* PhD thesis, University of Nijmegen, The Netherlands, Jan. 1995. Supervisor: Prof. Barendregt. ISBN 90-9007860-6.
17. A. B. Compagnoni and B. C. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6:469–501, 1996.
18. M. Coppo and M. Dezani-Ciancaglini. A new type-assignment for λ-terms. *Archiv. Math. Logik*, 19:139–156, 1978.
19. P.-L. Curien and G. Ghelli. Subtyping + extensionality: Confluence of $\beta\eta$-reductions in $F_{\leq}$. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science, pages 731–749. Springer-Verlag, Sept. 1991.
20. P.-L. Curien and G. Ghelli. Coherence of subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.
21. D. Duggan and A. Compagnoni. Subtyping for object type constructors. In *Foundations of Object-Oriented Languages (FOOL 6).* http://www.cs.williams.edu/~kim/FOOL/sched6.html, 1999.
22. H. Goguen. Typed operational semantics. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 186–200. Springer–Verlag, Apr. 1995.
23. C. A. Gunter and J. C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* The MIT Press, 1994.
24. M. Hofmann and B. Pierce. Positive subtyping. *Information and Computation*, 126(1), 1996.
25. A. Jones, Z. Luo, and S. Soloviev. Some algorithmic and proof-theoretical aspects of coercive subtyping. In *Proceedings of TYPES'96*, Lecture Notes in Computer Science, 1996.
26. P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.
27. J. C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 109–124, Jan. 1990.
28. J. C. Mitchell, F. Honsell, and K. Fisher. A lambda calculus of objects and method specialization. In *1993 IEEE Symposium on Logic in Computer Science*, June 1993.
29. F. Pfenning. Refinement types for logical frameworks. In *Informal Proceedings of the 1993 Workshop on Types for Proofs and Programs*, pages 315–328, May 1993.
30. B. Pierce and M. Steffen. Higher-order subtyping. *Theoretical Computer Science*, 176(1–2):235–282, 1997. corrigendum in TCS vol. 184 (1997), p. 247.
31. B. C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 1993.
32. B. C. Pierce. Bounded quantification with bottom. Technical Report 492, Computer Science Department, Indiana University, 1997.
33. M. Steffen. *Polarized Higher-Order Subtyping.* PhD thesis, Lehrstuhl für Informatik VII, Friedrich Alexander Universität Erlangen-Nürnberg, 1998.
34. J. Tiuryn. Equational axiomatization of bicoercibility for polymorphic types. In *Proceedings of FST-TCS'95*, 1995.

35. J. Zwanenburg. *An Object Oriented Programming Logic Based on Type Theory.* PhD thesis, Eindhoven University of Technology, 1999. Supervisor: Prof. Hemerik.

A Results from [15]

Lemma 11 (Adequacy).

1. *If* $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$ *then* $K \twoheadrightarrow_\beta K'$.
2. *If* $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} B \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} C : K$ *then* $A \twoheadrightarrow_\beta B \twoheadrightarrow_\beta C$.

Lemma 12 (Context Conversion). *If* $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$ *and* $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{n}} A' : K'$ *and* $\Gamma,\ X{\leq}A{:}K \vdash_S J$ *then* $\Gamma,\ X{\leq}A'{:}K' \vdash_S J$.

Lemma 13 (Determinacy).

1. *If* $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} B \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} C : K$ *and* $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} D \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} E : K'$ *then* $B \equiv D$, $C \equiv E$ *and* $K \equiv K'$.
2. *If* $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$ *and* $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K''$ *then* $K' \equiv K''$.

Lemma 14 (Subject Reduction). $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} B \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} C : K$ *and* $A \to_\beta A'$ *imply there is a* B' *such that* $B \twoheadrightarrow_\beta B'$ *and* $\Gamma \vdash_S A' \twoheadrightarrow_{\mathrm{w}} B' \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} C : K$.

Lemma 15 (Subtyping Conversion). *Suppose that* $\Gamma \vdash_S A \leq_W B : K$. *Then:*

1. *If* $\Gamma \vdash_S A, A' \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} C : K$ *then* $\Gamma \vdash_S A' \leq_W B : K$.
2. *If* $\Gamma \vdash_S B, B' \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} C : K$ *then* $\Gamma \vdash_S A \leq_W B' : K$.

Similarly for $\Gamma \vdash_S A \leq B : K$.

Proposition 2 (Completeness).

1. $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$ *implies* $\Gamma \vdash K$ *and* $\Gamma \vdash K =_\beta K'$.
2. $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} B \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} C : K$ *implies* $\Gamma \vdash A : K$, $\Gamma \vdash A =_\beta B : K$ *and* $\Gamma \vdash A =_\beta C : K$.
3. $\Gamma \vdash_S A \leq_W B : K$ *implies* $\Gamma \vdash A \leq B : K$.
4. $\Gamma \vdash_S A \leq B : K$ *implies* $\Gamma \vdash A, B : K$ *and* $\Gamma \vdash A \leq B : K$.

Theorem 3 (Soundness).

1. *If* $\Gamma \vdash K$ *then there is a* K' *such that* $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$; *if* $\Gamma \vdash K =_\beta K'$ *then there is a* K'' *such that* $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K''$ *and* $\Gamma \vdash_S K' \twoheadrightarrow_{\mathrm{n}} K''$.
2. *If* $\Gamma \vdash A : K$ *then there are* K', B *and* C *such that* $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$ *and* $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} B \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} C : K'$; *if* $\Gamma \vdash A =_\beta B : K$ *then there are* C *and* K' *such that* $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$, $\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{n}} C : K'$ *and* $\Gamma \vdash_S B \twoheadrightarrow_{\mathrm{n}} C : K'$.
3. *If* $\Gamma \vdash A \leq B : K$ *then there is a* K' *such that* $\Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K'$ *and* $\Gamma \vdash_S A \leq B : K'$.

Lemma 16 (Upper Bound).
If $\Gamma \vdash X(A_1, \ldots, A_m) : K$ *then* $\Gamma \vdash \Gamma(X)(A_1, \ldots, A_m) : K$.

Lemma 17. *If* $\Gamma \vdash_S A \leq B : K$ *then* $\Gamma \vdash_S A : K$ *and* $\Gamma \vdash_S B : K$.

B Rules for $\mathcal{F}^{\omega}_{\leq}$

Equality Rules

The equality rules for $\mathcal{F}^{\omega}_{\leq}$ include typed rules of inference stating that $\Gamma \vdash A =_\beta B : K$ is an equivalence relation and a compatible closure for $A \to B$, $\forall X{\leq}A_1{:}K.A_2$, $\Lambda X{\leq}A_1{:}K.A_2$, and type application, in addition to the following two rules:

$$\frac{\Gamma,\ X{\leq}A_1{:}K_1 \vdash A_2 : K_2 \qquad \Gamma \vdash C \leq A_1 : K_1}{\Gamma \vdash (\Lambda X{\leq}A_1{:}K_1.A_2)\, C =_\beta A_2[X{\leftarrow}C] : K_2[X{\leftarrow}C]} \quad (\text{T-Eq-Beta})$$

$$\frac{\Gamma \vdash A =_\beta B : K \qquad \Gamma \vdash K =_\beta K'}{\Gamma \vdash A =_\beta B : K'} \quad (\text{T-Eq-Conv})$$

Subtyping Rules

The subtyping rules for $\mathcal{F}^{\omega}_{\leq}$ include typed rules stating that $\Gamma \vdash A \leq B : K$ is transitive, that it includes the relation $\Gamma \vdash A =_\beta B : K$, that T_K is greater than all A of kind K, and the following rules:

$$\frac{\Gamma_1,\ X{\leq}A{:}K,\ \Gamma_2 \vdash \mathrm{ok}}{\Gamma_1,\ X{\leq}A{:}K,\ \Gamma_2 \vdash X \leq A : K} \quad (\text{S-TVar})$$

$$\frac{\Gamma \vdash B_1 \leq A_1 : \star \qquad \Gamma \vdash A_2 \leq B_2 : \star}{\Gamma \vdash A_1{\to}A_2 \leq B_1{\to}B_2 : \star} \quad (\text{S-Arrow})$$

$$\frac{\Gamma,\ X{\leq}C{:}K \vdash A \leq B : \star}{\Gamma \vdash \forall X{\leq}C{:}K.A \leq \forall X{\leq}C{:}K.B : \star} \quad (\text{S-All})$$

$$\frac{\Gamma,\ X{\leq}C{:}K_1 \vdash A \leq B : K_2}{\Gamma \vdash \Lambda X{\leq}C{:}K_1.A \leq \Lambda X{\leq}C{:}K_1.B : \Pi X{\leq}C{:}K_1.K_2} \quad (\text{S-TAbs})$$

$$\frac{\Gamma \vdash A \leq B : \Pi X{\leq}D{:}K_1.K_2 \qquad \Gamma \vdash C \leq D : K_1}{\Gamma \vdash A\, C \leq B\, C : K_2[X{\leftarrow}C]} \quad (\text{S-TApp})$$

$$\frac{\Gamma \vdash A \leq B : K \qquad \Gamma \vdash K =_\beta K'}{\Gamma \vdash A \leq B : K'} \quad (\text{S-K-Conv})$$

C Rules for the Typed Operational Semantics

Type Reduction Rules

$$\frac{\Gamma \vdash_S \mathrm{ok}}{\Gamma \vdash_S \mathrm{T}_\star \twoheadrightarrow_{\mathrm{w}} \mathrm{T}_\star \ {}_{\mathrm{w}}{\twoheadrightarrow_{\mathrm{n}}}\ \mathrm{T}_\star : \star} \quad (\text{ST-Top})$$

$$\frac{\Gamma \vdash_S A : K' \qquad \Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K' \qquad (X{\leq}A : K) \in \Gamma}{\Gamma \vdash_S X \twoheadrightarrow_{\mathrm{w}} X \ {}_{\mathrm{w}}{\twoheadrightarrow_{\mathrm{n}}}\ X : K'} \quad (\text{ST-TVar})$$

$$\frac{\begin{array}{l}\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} X(A_1,\ldots,A_m) : \Pi Y{\leq}C{:}K_1.K_2 \qquad \Gamma \vdash_S \Gamma(X) \twoheadrightarrow_{\mathrm{n}} D : K' \\ \Gamma \vdash_S D(A_1,\ldots,A_m,F) : K \qquad \Gamma \vdash_S B \twoheadrightarrow_{\mathrm{w}} E \ {}_{\mathrm{w}}{\twoheadrightarrow_{\mathrm{n}}}\ F : K_1 \\ \Gamma \vdash_S E \leq_W C : K_1 \qquad \Gamma \vdash_S K_2[Y{\leftarrow}B] \twoheadrightarrow_{\mathrm{n}} K\end{array}}{\Gamma \vdash_S A\, B \twoheadrightarrow_{\mathrm{w}} X(A_1,\ldots,A_m,F) \ {}_{\mathrm{w}}{\twoheadrightarrow_{\mathrm{n}}}\ X(A_1,\ldots,A_m,F) : K} \quad (\text{ST-TApp})$$

$$\frac{\Gamma \vdash_S A_1 \twoheadrightarrow_{\mathrm{n}} B_1 : \star \qquad \Gamma \vdash_S A_2 \twoheadrightarrow_{\mathrm{n}} B_2 : \star}{\Gamma \vdash_S (A_1 \to A_2) \twoheadrightarrow_{\mathrm{w}} (A_1 \to A_2) \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} (B_1 \to B_2) : \star} \quad (\text{ST-Arrow})$$

$$\frac{\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{n}} C : K' \qquad \Gamma \vdash_S K \twoheadrightarrow_{\mathrm{n}} K' \qquad \Gamma,\, X{\leq}A{:}K \vdash_S B \twoheadrightarrow_{\mathrm{n}} D : \star}{\Gamma \vdash_S \forall X{\leq}A{:}K.B \twoheadrightarrow_{\mathrm{w}} \forall X{\leq}A{:}K.B \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} \forall X{\leq}C{:}K'.D : \star} \quad (\text{ST-All})$$

$$\frac{\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{n}} C : K_1' \qquad \Gamma \vdash_S K_1 \twoheadrightarrow_{\mathrm{n}} K_1' \qquad \Gamma,\, X{\leq}A{:}K_1 \vdash_S B \twoheadrightarrow_{\mathrm{n}} D : K_2}{\Gamma \vdash_S \Lambda X{\leq}A{:}K_1.B \twoheadrightarrow_{\mathrm{w}} \Lambda X{\leq}A{:}K_1.B \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} \Lambda X{\leq}C{:}K_1'.D : \Pi X{\leq}C{:}K_1'.K_2} \quad (\text{ST-TAbs})$$

$$\frac{\begin{array}{l}\Gamma \vdash_S B \twoheadrightarrow_{\mathrm{w}} \Lambda X{\leq}A{:}K_1.D : \Pi X{\leq}A'{:}K_1'.K_2 \qquad \Gamma \vdash_S K_2[X{\leftarrow}C] \twoheadrightarrow_{\mathrm{n}} K \\ \Gamma \vdash_S D[X{\leftarrow}C] \twoheadrightarrow_{\mathrm{w}} E \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} F : K \qquad \Gamma \vdash_S C \leq A : K_1'\end{array}}{\Gamma \vdash_S B\,C \twoheadrightarrow_{\mathrm{w}} E \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} F : K} \quad (\text{ST-Beta})$$

Weak-Head Subtyping and Subtyping

$$\frac{\Gamma \vdash_S A \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} B : \star \qquad \mathrm{HV}(A) \text{ undefined}}{\Gamma \vdash_S A \leq_W \mathrm{T}_\star : \star} \quad (\text{SWS-Top})$$

$$\frac{\begin{array}{l}\Gamma \vdash_S X(A_1, \ldots, A_m) \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} C : K \qquad \Gamma \vdash_S \Gamma(X) \twoheadrightarrow_{\mathrm{n}} B : K' \\ \Gamma \vdash_S B(A_1, \ldots, A_m) \twoheadrightarrow_{\mathrm{w}} E : K \qquad \Gamma \vdash_S E \leq_W A : K \\ A \not\equiv X(A_1, \ldots, A_m)\end{array}}{\Gamma \vdash_S X(A_1, \ldots, A_m) \leq_W A : K} \quad (\text{SWS-TApp})$$

$$\frac{\Gamma \vdash_S X(A_1, \ldots, A_m) \ {}_{\mathrm{w}}\!\twoheadrightarrow_{\mathrm{n}} B : K}{\Gamma \vdash_S X(A_1, \ldots, A_m) \leq_W X(A_1, \ldots, A_m) : K} \quad (\text{SWS-Refl})$$

$$\frac{\Gamma \vdash_S B_1 \leq A_1 : \star \qquad \Gamma \vdash_S A_2 \leq B_2 : \star}{\Gamma \vdash_S A_1 \to A_2 \leq_W B_1 \to B_2 : \star} \quad (\text{SWS-Arrow})$$

$$\frac{\begin{array}{l}\Gamma,\, X{\leq}A_1{:}K \vdash_S A_2 \leq B_2 : \star \qquad \Gamma \vdash_S K, K' \twoheadrightarrow_{\mathrm{n}} K'' \\ \Gamma \vdash_S A_1, B_1 \twoheadrightarrow_{\mathrm{n}} C : K''\end{array}}{\Gamma \vdash_S \forall X{\leq}A_1{:}K.A_2 \leq_W \forall X{\leq}B_1{:}K'.B_2 : \star} \quad (\text{SWS-All})$$

$$\frac{\begin{array}{l}\Gamma,\, X{\leq}A_1{:}K_1 \vdash_S A_2 \leq B_2 : K_2 \qquad \Gamma \vdash_S K_1, K_1' \twoheadrightarrow_{\mathrm{n}} K_1'' \\ \Gamma \vdash_S A_1, B_1 \twoheadrightarrow_{\mathrm{n}} C : K_1''\end{array}}{\Gamma \vdash_S \Lambda X{\leq}A_1{:}K_1.A_2 \leq_W \Lambda X{\leq}B_1{:}K_1'.B_2 : \Pi X{\leq}C{:}K_1''.K_2} \quad (\text{SWS-TAbs})$$

$$\frac{\Gamma \vdash_S A \twoheadrightarrow_{\mathrm{w}} C : K \quad \Gamma \vdash_S B \twoheadrightarrow_{\mathrm{w}} D : K \quad \Gamma \vdash_S C \leq_W D : K}{\Gamma \vdash_S A \leq B : K} \quad (\text{SS-Inc})$$

Safe Proof Checking in Type Theory with Y

Herman Geuvers[1], Erik Poll[2], and Jan Zwanenburg[2]

[1] Computer Science Department, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, herman@win.tue.nl
[2] Computer Science Department, University of Nijmegen, P.O. Box 9010, 6500 GL Nijmegen , The Netherlands, {erikpoll,janz}@cs.kun.nl

Abstract. We present an extension of type theory with a fixed point combinator Y. We are particularly interested in using this Y for doing unbounded proof search in the proof system. Therefore we treat in some detail a typed λ-calculus for higher order predicate logic with inductive types (a reasonable subsystem of the theory implemented in [Dowek e.a. 1991]) and show how bounded proof search can be done in this system, and how unbounded proof search can be done if we add Y. Of course, proof search can also be implemented (as a tactic) in the meta language. This may give faster results, but asks from the user to be able to program the implementation. In our approach the user works completely in the proof system itself. We also provide the meta theory of type theory with Y that allows to use the fixed point combinator in a *safe* way. Most importantly, we prove a kind of *conservativity result*, showing that, if we can generate a proof term M of formula φ in the extended system, and M does not contain Y, then M is already a proof of φ in the original system.

1 Introduction

In theorem provers based on type theory, we are always looking for an explicit *proof-object*, i.e. if we want to prove the formula φ, we are in fact looking for a term M such that $M : \varphi$. (M is of type φ.) Such a term M then corresponds to a derivation in standard natural deduction (and can be translated to a proof in natural language text). This has the advantage that, besides the proof engine telling us that the formula is provable, the engine also produces – interactively with the user – a proof term that can be checked independently. As a matter of fact, the program for *checking* a proof object is relatively simple: it is a type checking algorithm for a strongly dependent-typed language. This conforms with the basic idea that *finding* a proof is difficult – hence this is done interactively, whereas *verifying* an alleged proof is simple.

The interaction with the proof engine usually exists in a set of goal-directed tactics. So, we try to construct a proof-term by looking at the structure of the goal to be proved. Of course, one can define more powerful tactics, especially when we are dealing with a decidable fragment of the logic. An example is the 'Tauto' tactic in Coq, that automatically solves (i.e. constructs proof-terms) for first order propositional logic (and a bit beyond).

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 439–452, 1999.

Tactics like Tauto are built-in in the engine, but the user can also define his/her own tactics, by programming them in the meta-language of the proof system. To do so, the user has to know the meta-language and the way the proof system is implemented in it quite well. This makes it in general quite hard to program one's own tactics. In this paper we present a kind of 'tactic' that can be programmed in the proof system itself, which allows *searching* for proof-terms. So no knowledge of the implementation is required. We also present two examples of its use and the underlying explanation of the method in terms of the proof system (the typed λ-calculus that is implemented in the proof engine).

The method we present can also be implemented as a tactic in the meta-language, and then it can certainly be made much faster. We believe that it is nice that a 'search-tactic' can be safely implemented in the language of the proof system itself, which makes it much easier to apply for a user.

Due to the expressiveness of typed λ-calculus, a lot of 'proof search' can be defined already in the proof system itself. E.g. if we have a decidable predicate Q over $\mathtt{nat}$ (i.e. a proof term P of type $\forall n{:}\mathtt{nat}.Q(n) \vee \neg Q(n)$), then we can do a *bounded search* for an element $m \leq N$ such that $Q(m)$ holds. The idea is to iterate P up to N times until we find an $m : \mathtt{nat}$ for which $Pm = \mathtt{inl}\ t$; then $t : Q(m)$. Note that this m will also be the smallest n for which $Q(n)$ holds.

An *unbounded search* can also be defined if we add a *fixed point combinator* to the typed λ-calculus. In the example above: using the fixed point combinator, we can iterate P without bound, until we find an $m : \mathtt{nat}$ such that $Pm = \mathtt{inl}\ t$, and then $Q(m)$ holds. Adding a fixed point combinator is of course a real extension of the proof system: as the underlying typed λ-calculus is strongly normalizing, no fixed point combinator can be defined in it. In this paper we show that adding a fixed point combinator Y is *safe*. This is done by showing that the addition of Y yields a *conservative extension*. That is, if $\vdash_S$ denotes derivability in some typed λ-calculus and $\vdash_{S+Y}$ denotes derivability in S extended with Y, then

$$\Gamma \vdash_{S+Y} M : A \Rightarrow \Gamma \vdash_S M : A,$$

for Γ, M and A not containing Y. (Of course we do *not* have conservativity in the *logical* sense: $\exists_M(\Gamma \vdash_{S+Y} M : A) \not\Rightarrow \exists_M(\Gamma \vdash_S M : A)$, for Γ and A not containing Y.) Now, in order to show that adding Y is *safe*, let φ be a formula in a certain context Γ (both φ and Γ in the system S). Suppose we have constructed a proof-term $P : \varphi$ in $S + Y$, so P may possibly contain Y. Now we let P reduce until we find a term P' that does not contain Y. Then, due to the *subject reduction* property for $S + Y$ (which we will prove), we have $\Gamma \vdash_{S+Y} P' : \varphi$ and hence $\Gamma \vdash_S P' : \varphi$ by conservativity.

How exactly the fixed point combinator is used to perform proof search will be detailed in the paper by some examples. The proof search is in fact performed by the reduction of the fixed point combinator, so, in terms of the previous paragraph, the search is in the reduction from P to P'.

The conservativity of $S + Y$ over S will be proved for arbitrary functional Pure Type Systems S. Functional Pure Type Systems cover a large class of typed λ-calculi, among which we find the simple typed λ-calculus, dependent

typed λ-calculus, polymorphic λ-calculus (known as system F) and the Calculus of Constructions. The core of the proof system of Coq is also a functional Pure Type System. We believe that the conservativity of $S + Y$ over S will extend to the whole proof system of Coq, which is a functional Pure Type System extended with inductive types.

2 Theorem Proving in Typed λ-Calculus

In this section we briefly introduce the notion of Pure Type System and give some examples of how theorem proving is done in such a system. Our main focus will be on the system λPREDω. This is a typed λ-calculus that faithfully represents constructive higher order predicate logic. To motivate this we give some examples of derivable judgements in λPREDω. For more information on Pure Type Systems and typed λ-calculus in general, we refer to [Barendregt 1992] and [Geuvers 1993].

Pure Type Systems or PTSs were first introduced by Berardi [Berardi 1990] and Terlouw [Terlouw 1989a], with slightly different definitions. The advantage of the class of PTSs is that many known systems can be seen as PTSs. So, many specific results for specific systems are immediate instances of general properties of PTSs. In the following we will mention a number of these properties.

Definition 1. *For $\mathcal{S}$ a set, the so called* sorts*, $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ (the* axioms*) and $\mathcal{R} \subset \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ (the* rules*), the* Pure Type System *$\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is the typed λ-calculus with the following deduction rules.*

$$\text{(sort)} \quad \vdash s_1 : s_2 \qquad \text{if } (s_1, s_2) \in \mathcal{A}$$

$$\text{(var)} \quad \frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$$

$$\text{(weak)} \quad \frac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x{:}A \vdash M : C}$$

$$(\Pi) \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}$$

$$(\lambda) \quad \frac{\Gamma, x{:}A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$$

$$\text{(app)} \quad \frac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$$

$$\text{(conv}_\beta\text{)} \quad \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} \qquad A =_\beta B$$

If $s_2 \equiv s_3$ in a triple $(s_1, s_2, s_3) \in \mathcal{R}$, we write $(s_1, s_2) \in \mathcal{R}$. In these rules, the expressions are taken from the set of pseudoterms T*, defined by*

$$\mathsf{T} ::= \mathcal{S} \mid \mathsf{V} \mid (\Pi \mathsf{V}{:}\mathsf{T}.\mathsf{T}) \mid (\lambda \mathsf{V}{:}\mathsf{T}.\mathsf{T}) \mid \mathsf{TT}.$$

The pseudoterm A is typable *if there is a context Γ and a pseudoterm B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$ is derivable.*

In the following, we will mainly be dealing with the PTS λPREDω, which is defined as follows.

λPREDω
$\mathcal{S}$ $\mathsf{Set}, \mathsf{Type}^s, \mathsf{Prop}, \mathsf{Type}^p$
$\mathcal{A}$ $(\mathsf{Set} : \mathsf{Type}^s), (\mathsf{Prop} : \mathsf{Type}^p)$
$\mathcal{R}$ $(\mathsf{Set}, \mathsf{Set}), (\mathsf{Set}, \mathsf{Type}^p), (\mathsf{Type}^p, \mathsf{Type}^p),$ $(\mathsf{Prop}, \mathsf{Prop}), (\mathsf{Set}, \mathsf{Prop}), (\mathsf{Type}^p, \mathsf{Prop})$

The idea is that Set is the sort (universe) of 'small' sets, Prop is the sort of propositions Type^p is the sort of 'large' sets (so Prop is a large set) and Type^s is the sort containing just Set.

We briefly explain the rules. The rule $(\mathsf{Prop}, \mathsf{Prop})$ is for forming the implication: $\varphi \to \psi$ for $\varphi, \psi : \mathsf{Prop}$. With $(\mathsf{Set}, \mathsf{Type}^p)$ one can form $A \to \mathsf{Prop} : \mathsf{Type}^p$ and $A \to A \to \mathsf{Prop} : \mathsf{Type}^p$, the domains of unary predicates and binary relations over A. $(\mathsf{Type}^p, \mathsf{Type}^p)$ allows to extend this to higher order predicates and relations, like $(A \to \mathsf{Prop}) \to \mathsf{Prop} : \mathsf{Type}^p$, the domain of predicates over predicates over A, and $(A \to A \to \mathsf{Prop}) \to \mathsf{Prop} : \mathsf{Type}^p$. The rule $(\mathsf{Set}, \mathsf{Prop})$ allows the quantification over small sets (i.e. A with $A : \mathsf{Set}$): one can form $\Pi x{:}A.\varphi$ (for $A : \mathsf{Set}$ and $\varphi : \mathsf{Prop}$), which is to be read as a universal quantification. $(\mathsf{Type}^p, \mathsf{Prop})$ allows also higher order quantification, i.e. over large sets, e.g. $\Pi P{:}A \to \mathsf{Prop}.\varphi : \mathsf{Prop}$. Using $(\mathsf{Set}, \mathsf{Set})$ one can define function types like the type of binary functions: $A \to A \to A$, but also $(A \to A) \to A$, which is usually referred to as a 'higher order function type'.

We motivate the definition by giving some examples of mathematical notions that can be formalised in λPREDω.

Example 1.

1. $\mathtt{nat}{:}\mathsf{Set}, 0{:}\mathtt{nat}, {>}{:}\mathtt{nat} \to \mathtt{nat} \to \mathsf{Prop} \vdash \lambda x{:}\mathtt{nat}.x{>}0 : \mathtt{nat} \to \mathsf{Prop}$. Here we see the use of λ-abstraction to define a predicate.
2. $\mathtt{nat}{:}\mathsf{Set}, 0{:}\mathtt{nat}, S{:}\mathtt{nat} \to \mathtt{nat} \vdash$ $\Pi P{:}\mathtt{nat} \to \mathsf{Prop}.(P0) \to (\Pi x{:}\mathtt{nat}.(Px \to P(Sx))) \to \Pi x{:}\mathtt{nat}.Px : \mathsf{Prop}$. This is the induction formula written down in λPREDω as a term of type Prop.
3. $A{:}\mathsf{Set}, R{:}A \to A \to \mathsf{Prop} \vdash \Pi x, y, z{:}A.Rxy \to Ryz \to Rxz : \mathsf{Prop}$. (This formula expresses transitivity of R.)
4. $A{:}\mathsf{Set} \vdash \lambda R, Q{:}A \to A \to \mathsf{Prop}.\Pi x, y{:}A.Rxy \to Qxy :$ $(A \to A \to \mathsf{Prop}) \to (A \to A \to \mathsf{Prop}) \to \mathsf{Prop}$. (This relation between binary relations on A expresses inclusion of relations.)
5. $A{:}\mathsf{Set} \vdash \lambda x, y{:}A.\Pi P{:}A \to \mathsf{Prop}.(Px \to Py) : A \to A \to \mathsf{Prop}$. This binary relation on A is also called 'Leibniz equality' and is usually denoted by $=_A$, denoting the domain type explicitly.

6. $A{:}\mathsf{Set}, x, y{:}A \vdash \lambda r : x =_A y.\lambda P{:}A{\to}\mathsf{Prop}.r(\lambda z{:}A.Pz{\to}Px)(\lambda q{:}Px.q) : x =_A y{\to}y =_A x$. The proof of symmetry of Leibniz equality.

The rules of Pure Type Systems give the flexibility to define subsystems in a rather easy way by restricting the set $\mathcal{R}$. The Pure Type System λPRED2, representing second order predicate logic, is defined from λPREDω by removing $(\mathsf{Type}^p, \mathsf{Type}^p)$ from the $\mathcal{R}$. (Then we can no longer form higher order predicates and relations.) To obtain first order predicate logic, we remove $(\mathsf{Type}^p, \mathsf{Prop})$ from λPRED2, which forbids quantification over second order domains (predicates, relations). Other well-known typed λ-calculi that can be described as a PTS are simple typed λ-calculus, polymorphic typed λ-calculus (also known as system F) and the Calculus of Constructions.

2.1 Properties of Pure Type Systems

An important motivation for the definition of Pure Type Systems is that many important properties can be proved for all PTSs at once. Here we list the most important properties and discuss them briefly. Proofs can be found in [Geuvers and Nederhof 1991] and [Barendregt 1992]. Here we only mention the ones that are needed for the proof of conservativity of the extension of a PTS with a fixed point combinator.

In the following, unless explicitly stated otherwise, $\vdash$ refers to derivability in an arbitrary PTS. Furthermore, Γ is a *correct* context means that $\Gamma \vdash M : A$ for some M and A.

Proposition 1 (Church-Rosser (CR)).
The β-reduction is Church-Rosser on the set of pseudoterms T.

Proposition 2 (Correctness of Types (CT)).
If $\Gamma \vdash M : A$ then $\Gamma \vdash A : s$ or $A \equiv s$ for some some $s \in \mathcal{S}$.

Proposition 3 (Subject Reduction (SR)).
If $\Gamma \vdash M : A$ and $M \longrightarrow\!\!\!\!\!\rightarrow_\beta N$, then $\Gamma \vdash N : A$.

Proposition 4 (Predicate Reduction (PR)).
If $\Gamma \vdash M : A$ and $A \longrightarrow\!\!\!\!\!\rightarrow_\beta A'$, then $\Gamma \vdash M : A'$.

There are also many (interesting) properties that hold for specific PTSs or specific classes of PTSs. We mention one of these properties.

Definition 2. *A PTS $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is* functional*, also called* singly sorted*, if the relations A and R are functions, i.e. if the following two properties hold*

$$\forall s_1, s_2, s_2' \in \mathcal{S}(s_1, s_2'), (s_1, s_2') \in \mathcal{A} \Rightarrow s_2 = s_2',$$
$$\forall s_1, s_2, s_3, s_3' \in \mathcal{S}(s_1, s_2, s_3), (s_1, s_2, s_3') \in \mathcal{R} \Rightarrow s_3 = s_3'$$

The PTSs that we have encountered so far are functional. So are all PTSs that are used in practice. Functional PTSs share the following nice property.

Proposition 5 (Unicity of Types for functional PTSs (UT)).
For functional *PTSs, if* $\Gamma \vdash M : A$ *and* $\Gamma \vdash M : B$, *then* $A =_\beta B$.

A less interesting, very basic, property, but one needed in a proof later, is:

Proposition 6 (Π-generation). *Let* $\Gamma \vdash \Pi x{:}A.B : s$. *Then there exists a rule* $(s_1, s_2, s) \in \mathcal{R}$ *such that* $\Gamma \vdash A : s_1$ *and* $\Gamma, x{:}A \vdash B : s_2$

An important property of a type system is that types can be *computed*, i.e. there is an algorithm that given Γ and M, computes an A for which $\Gamma \vdash M : A$ holds, and if there is no such A, returns 'false'. This is usually referred to as the *type inference problem*.

There are two important properties that ensure that type inference is decidable: Church-Rosser for β-reduction and Normalization for β-reduction. Of course, when adding a fixed point combinator, normalization is lost. In the next section we will discuss why, for the relevant fragment of the system, type checking is still decidable.

2.2 Inductive Types

We briefly treat the extension of λPREDω with inductive types, by giving some examples and how they are used. λPREDω + inductive types does not fully cover the type system of Coq, but quite a bit of it. At least it covers enough to be able to describe our examples of proof search in the next section. The scheme we give is roughly the one first introduced in [Coquand and Mohring 1990] and implemented in [Dowek e.a. 1991].

We first give the (very basic) example of natural numbers $\mathtt{nat}$. One is allowed to write down the following definition.

$$\begin{array}{l}\mathsf{Inductive\ \ definition}\ \mathtt{nat} : \mathsf{Set} := \\ \qquad\qquad\qquad\qquad 0 : \mathtt{nat} \\ \qquad\qquad\qquad\qquad S : \mathtt{nat}{\to}\mathtt{nat}.\end{array}$$

to obtain the following rules.

$$(\mathrm{elim}_1)\ \frac{\Gamma \vdash A : \mathsf{Set} \quad \Gamma \vdash f_1 : A \quad \Gamma \vdash f_2 : \mathtt{nat}{\to}A{\to}A}{\Gamma \vdash \mathsf{Rec}_{\mathtt{nat}} f_1 f_2 : \mathtt{nat}{\to}A}$$

$$(\mathrm{elim}_2)\ \frac{\Gamma \vdash P : \mathtt{nat}{\to}\mathsf{Prop} \quad \Gamma \vdash f_1 : P0 \quad \Gamma \vdash f_2 : \Pi x{:}\mathtt{nat}.Px{\to}P(Sx)}{\Gamma \vdash \mathsf{Rec}_{\mathtt{nat}} f_1 f_2 : \Pi x{:}\mathtt{nat}.Px}$$

$$(\mathrm{elim}_3)\ \frac{\Gamma \vdash A : \mathsf{Type}^p \quad \Gamma \vdash f_1 : A \quad \Gamma \vdash f_2 : \mathtt{nat}{\to}A{\to}A}{\Gamma \vdash \mathsf{Rec}_{\mathtt{nat}} f_1 f_2 : \mathtt{nat}{\to}A}$$

The rule (elim_1) allows the definition of functions by primitive recursion. The rule (elim_2) allows proofs by induction. The rule (elim_3) allows the definition of

predicates (on $\mathtt{nat}$) by induction. To make sure that the functions defined by the (elim) rules compute in the correct way, Rec has the following reduction rule.

$$\begin{aligned}\mathsf{Rec}_{\mathtt{nat}} f_1 f_2 0 &\longrightarrow_\iota f_1\\ \mathsf{Rec}_{\mathtt{nat}} f_1 f_2 (St) &\longrightarrow_\iota f_2 t (\mathsf{Rec}_{\mathtt{nat}} f_1 f_2 t)\end{aligned}$$

The additional ι-reduction is also included in the *conversion*-rule (conv), where we now have as a side-condition '$A =_{\beta\iota} B$'. The subscript in $\mathsf{Rec}_{\mathtt{nat}}$ will be omitted, when clear from the context.

An example of the use of (elim_1) is in the definition of the 'double' function d, which is defined by

$$d := \mathsf{Rec}\, 0 (\lambda x{:}\mathtt{nat}.\lambda y{:}\mathtt{nat}.S(S(y))).$$

Now, $d0 \longrightarrow\!\!\!\!\rightarrow_{\beta\iota} 0$ and $d(Sx) \longrightarrow\!\!\!\!\rightarrow_{\beta\iota} S(S(dx))$. The predicate of 'being even', $\mathsf{even}(-)$, can be defined by using (elim_3):

$$\mathsf{even}(-) := \mathsf{Rec}\,(\top)(\lambda x{:}\mathtt{nat}.\lambda \alpha{:}\mathsf{Prop}.\neg\alpha).$$

Here, $\neg\varphi$ is defined as $\varphi{\to}\bot$. We obtain indeed that

$$\begin{aligned}\mathsf{even}(0) &\longrightarrow\!\!\!\!\rightarrow_{\beta\iota} \top,\\ \mathsf{even}(Sx) &\longrightarrow\!\!\!\!\rightarrow_{\beta\iota} \neg\mathsf{even}(x)\end{aligned}$$

An example of the use of (elim_2) is the proof of $\Pi x{:}\mathtt{nat}.\mathsf{even}(dx)$. Say that true is some canonical inhabitant of type $\top$. Using $\mathsf{even}(d(Sx)) =_{\beta\iota} \neg\neg\mathsf{even}(dx)$ we also find that the term $\lambda x{:}\mathtt{nat}.\lambda h{:}\mathsf{even}(dx).\lambda z{:}\neg\mathsf{even}(dx).zh$ is of type $\Pi x{:}\mathtt{nat}.\mathsf{even}(dx){\to}\mathsf{even}(d(Sx))$. So we conclude that

$$\vdash \mathsf{Rec}\,\mathsf{true}(\lambda x{:}\mathtt{nat}.\lambda h{:}\mathsf{even}(dx).\lambda z{:}\neg\mathsf{even}(dx).zh) : \Pi x{:}\mathtt{nat}.\mathsf{even}(dx).$$

Another well-known example is the type of lists over a domain D. This is usually defined as a *parametric inductive type*, taking the domain as a parameter of the inductive definition. The type of *parametric lists* can be defined as follows.

$$\begin{aligned}&\mathsf{Inductive\ \ definition\ List} : \mathsf{Set}{\to}\mathsf{Set} :=\\ &\qquad\qquad \mathsf{Nil} : \Pi D{:}\mathsf{Set}.(\mathsf{List}D)\\ &\qquad\qquad \mathsf{Cons} : \Pi D{:}\mathsf{Set}.(\mathsf{List}D){\to}D{\to}(\mathsf{List}D).\end{aligned}$$

Which generates the following elimination rules and reduction rule.

$$(\mathrm{elim}_1)\ \frac{\Gamma \vdash D : \mathsf{Set}\ \ \Gamma \vdash A : \mathsf{Set}\ \ \Gamma \vdash f_1 : A\ \ \Gamma \vdash f_2 : (\mathsf{List}D){\to}D{\to}A{\to}A}{\Gamma \vdash \mathsf{Rec}_{\mathsf{List}} f_1 f_2 : (\mathsf{List}D){\to}A}$$

$$(\mathrm{elim}_2)\ \frac{\begin{array}{ll}\Gamma \vdash D : \mathsf{Set} & \Gamma \vdash f_1 : P(\mathsf{Nil}D)\\ \Gamma \vdash P : (\mathsf{List}D){\to}\mathsf{Prop} & \Gamma \vdash f_2 : \Pi x{:}(\mathsf{List}D).\Pi d{:}D.Px{\to}P(\mathsf{Cons}xd)\end{array}}{\Gamma \vdash \mathsf{Rec}_{\mathsf{List}} f_1 f_2 : \Pi x{:}(\mathsf{List}D).Px}$$

$$(\mathrm{elim}_3)\ \frac{\Gamma \vdash D : \mathsf{Set}\ \ \Gamma \vdash A : \mathsf{Type}^p\ \ \Gamma \vdash f_1 : A\ \ \Gamma \vdash f_2 : (\mathsf{List}D){\to}D{\to}A{\to}A}{\Gamma \vdash \mathsf{Rec}_{\mathsf{List}} f_1 f_2 : (\mathsf{List}D){\to}A}$$

$$\mathsf{Rec}_{\,\mathsf{List}} f_1 f_2 (\mathsf{Nil} D) \longrightarrow_\iota f_1$$
$$\mathsf{Rec}_{\,\mathsf{List}} f_1 f_2 (\mathsf{Cons} D l d) \longrightarrow_\iota f_2 l d (\mathsf{Rec}_{\,\mathsf{List}} f_1 f_2 l)$$

Note that to be able to write down the type of the constructors Nil and Cons, we need to add the rule $(\mathsf{Type}^s, \mathsf{Set})$ to λPREDω. The constructors Nil and Cons have a dependent type. It turns out that this situation occurs more often. We treat another interesting example: the Σ-type. Let $B : \mathsf{Set}$ and $Q : B \to \mathsf{Prop}$ and suppose we have added the rule $(\mathsf{Prop}, \mathsf{Set})$ to our system.

$$\begin{array}{l}\mathsf{Inductive\ \ definition}\ \mu : \mathsf{Set} := \\ \qquad\qquad\qquad\qquad \mathsf{In} : \Pi z{:}B.(Qz) \to \mu.\end{array}$$

$$(\mathrm{elim}_1)\ \frac{\Gamma \vdash A : \mathsf{Set} \quad \Gamma \vdash f_1 : \Pi z{:}B.(Qz) \to A}{\Gamma \vdash \mathsf{Rec}_{\,\mu} f_1 : \mu \to A}$$

$$(\mathrm{elim}_2)\ \frac{\Gamma \vdash P : \mu \to \mathsf{Prop} \quad \Gamma \vdash f_1 : \Pi z{:}B.\Pi y{:}(Qz).P(\mathsf{In} z y)}{\Gamma \vdash \mathsf{Rec}_{\,\mu} f_1 : \Pi x{:}\mu.(Px)}$$

$$(\mathrm{elim}_3)\ \frac{\Gamma \vdash A : \mathsf{Type}^p \quad \Gamma \vdash f_1 : \Pi z{:}B.(Qz) \to A}{\Gamma \vdash \mathsf{Rec}_{\,\mu} f_1 : \mu \to A}$$

The ι-reduction rule is

$$\mathsf{Rec}_{\,\mu} f_1 (\mathsf{In} b q) \longrightarrow_\iota f_1 b q$$

Now, taking in (elim_1) B for A and $\lambda z{:}B.\lambda y{:}(Qz).z$ for f_1, we find that $\mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).z)(\mathsf{In} b q) \longrightarrow\!\!\!\!\rightarrow b$. Hence we define $\pi_1 := \mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).z)$. Now, taking in (elim_2) $\lambda x{:}\mu.Q(\pi_1 x)$ for P and $\lambda z{:}B.\lambda y{:}(Qz).y$ for f_1, we find that $\mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).y) : \Pi z{:}\mu.Q(\pi_1 z)$. Also, $\mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).y)(\mathsf{In} b q) \longrightarrow\!\!\!\!\rightarrow q$. Hence we define $\pi_2 := \mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).y)$ and we remark that μ together with In (as pairing constructor) and π_1 and π_2 (as projections) represents the Σ-type. In the rest of this article, we will just use the Σ-type, and will write $\langle n, p \rangle$ for the pair of n and p.

An example of an inductively defined proposition is the disjunction. Given φ and ψ of type Prop, $\varphi \vee \psi$ can be defined as follows.

$$\begin{array}{l}\mathsf{Inductive\ \ definition}\ \varphi \vee \psi : \mathsf{Prop} := \\ \qquad\qquad\qquad\qquad \mathtt{inl} : \varphi \to (\varphi \vee \psi) \\ \qquad\qquad\qquad\qquad \mathtt{inr} : \psi \to (\varphi \vee \psi)\end{array}$$

We add the $(\mathsf{Prop}, \mathsf{Set})$ rule to λPREDω, because we want to have the first two elimination rules.

$$(\mathrm{elim}_1)\ \frac{\Gamma \vdash A : \mathsf{Set} \quad \Gamma \vdash f_1 : \varphi \to A \quad \Gamma \vdash f_2 : \psi \to A}{\Gamma \vdash \mathsf{Rec}_{\,\vee} f_1 f_2 : (\varphi \vee \psi) \to A}$$

$$(\mathrm{elim}_2)\ \frac{\Gamma \vdash P : \mathsf{Prop} \quad \Gamma \vdash f_1 : \varphi \to P \quad \Gamma \vdash f_2 : \psi \to P}{\Gamma \vdash \mathsf{Rec}_{\,\vee} f_1 f_2 : (\varphi \vee \psi) \to P}$$

$$\mathsf{Rec}_{\vee} f_1 f_2(\mathtt{inl}\ q) \longrightarrow_{\iota} f_1 q,$$
$$\mathsf{Rec}_{\vee} f_1 f_2(\mathtt{inr}\ q) \longrightarrow_{\iota} f_2 q.$$

As usual we write

$$\begin{array}{l}\mathtt{case}\ t\ \mathtt{of}\ \ \mathtt{inl}\ (p) \Rightarrow M_1\\ \qquad\qquad\quad \mathtt{inr}\ (p) \Rightarrow M_2\end{array}$$

for $\mathsf{Rec}_{\vee}(\lambda p{:}\varphi.M_1)(\lambda p{:}\psi.M_2)t$.

Similarly one can define the disjoint union of two small sets, $A + B$ for $A, B : \mathsf{Set}$, inductively. We will ambiguously use the same notations for the constructors of $A + B$.

3 Proof Search in Type Theoretic Theorem Provers

We treat two examples of proof search in Coq. We try to avoid using Coq-syntax and describe the examples in terms of λPREDω with inductive types. The first example is a search for a term $n : \mathtt{nat}$ such that $Q(n)$ holds, where Q is a decidable predicate. So we let $Q : \mathtt{nat}{\to}\mathsf{Prop}$ and we assume we have a term

$$P : \Pi n{:}\mathtt{nat}.Q(n) \vee \neg Q(n).$$

Now, we want to iterate P to find the n and the proof of $Q(n)$. First suppose that we hope to find the n before N, so we want to iterate P at most N times ($N : \mathtt{nat}$).

Definition 3. *For $A : \mathsf{Set}$, $a : A$ and $n : \mathtt{nat}$ we define $Y_a^n : (A{\to}A){\to}A$ as follows.*

$$Y_a^n := \lambda f{:}A{\to}A.\mathsf{Rec}_{\mathtt{nat}} a(\lambda x{:}\mathtt{nat}.f)n.$$

The following is easily verified.

$$Y_a^0 f \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} a,$$
$$Y_a^{n+1} f \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} f(Y_a^n f),$$
$$Y_a^n f \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} f^n(a),$$

where $f^n(a)$ denotes, as usual, n times application of f on a.

Now define

$$\begin{array}{l}F :\equiv \lambda g{:}\mathtt{nat}{\to}\mathtt{nat}.\lambda n{:}\mathtt{nat}.\mathtt{case}\ (Pn)\ \mathtt{of}\ \ \mathtt{inl}\ (p) \Rightarrow n\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathtt{inr}\ (p) \Rightarrow g(n+1).\end{array}$$

So, $F : (\mathtt{nat}\to\mathtt{nat})\to(\mathtt{nat}\to\mathtt{nat})$. Now, let $N : \mathtt{nat}$ and let $I := \lambda x{:}\mathtt{nat}.x$. Then

$$\begin{array}{rll}
Y_I^N F0 \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} 0 & & \text{if } P0 \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} \mathtt{inl}\ (p) : Q(0),\\
& Y_I^{N-1}F1 & \text{if } P0 \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} \mathtt{inr}\ (p) : \neg Q(0),\\
Y_I^{N-1}F1 \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} 1 & & \text{if } P1 \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} \mathtt{inl}\ (p) : Q(1),\\
& Y_I^{N-2}F2 & \text{if } P1 \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} \mathtt{inr}\ (p) : \neg Q(1),\\
Y_I^{N-2}F2 \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} 2 & & \text{if } P2 \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} \mathtt{inl}\ (p) : Q(2),\\
& Y_I^{N-3}F3 & \text{if } P2 \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} \mathtt{inr}\ (p) : \neg Q(2),\\
\cdots & & \\
Y_I^1 F(N-1) \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} N-1 & & \text{if } P(N-1) \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} \mathtt{inl}\ (p) : Q(N-1),\\
& Y_I^0 FN & \text{if } P(N-1) \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} \mathtt{inr}\ (p) : \neg Q(N-1),\\
Y_I^0 FN \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} N. & &
\end{array}$$

So, if $Y_I^N F0 \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} n$ with $n < N$, then $Q(n)$ holds and $P(n) \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} \mathtt{inl}\ (p)$ with p a proof of $Q(n)$. If $Y_I^N F0 \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} N$, then $\neg Q(n)$ for all $n < N$.

The method above works if we know an upperbound to the n that we want to search for. Another option is to start a (possibly non-terminating) search. This can be done by adding the fixed point combinator to λPREDω with inductive types. We define the extension very generally for PTSs.

Definition 4. *Let $S = \lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ be a PTS and let s be a sort of the system S. The system $S + Y^s$ is obtained by adding the following rule.*

$$\frac{\Gamma \vdash A : s \;\; \Gamma \vdash f : A\to A}{\Gamma \vdash Y^s f : A}$$

The β-reduction is extended with

$$Y^s f \longrightarrow_Y f(Y^s f).$$

We sometimes omit the superscript s, if we know which sort we are talking about. If we add Y^s for all sorts, we just talk about $S + Y$.

In λPREDω with inductive types and Y^{Set}, we can now program an arbitrary proof search. (We omit the superscript Set.) With the above definition for F we obtain

$$YFI : \mathtt{nat}$$

and if $YFI \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota Y} n$ with n a normal form, then we know that $Q(n)$ holds and $Pn \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota Y} \mathtt{inl}\ (p)$ with $p : Q(n)$. (Moreover, we know that n is the smallest m for which $Q(m)$ holds, but only on a meta-level: the proof term does not represent this information. If YFI does not terminate, there is no n for which $Q(n)$ holds.)

We may wonder whether the extension of a type system with Y is safe. This will be the subject of the next section. Here we consider one more application of the proof search method, where we want to verify whether Q holds for $n = 0, 1, \ldots, N$.

Suppose again we have our decidable predicate Q with P a proof term of type $\Pi n{:}\mathtt{nat}.Q(n) \vee \neg Q(n)$. If we want to prove that Q holds for $n = 0, 1, \ldots, N$, we do not just want to verify that fact, but also store the proof terms of $Q(0), Q(1), \ldots, Q(N)$. Moreover, if $Q(n)$ fails for an $n \leq N$, we want to return this n. Now abbreviate

$$\mathsf{List} := \mathsf{List}(\Sigma x{:}\mathtt{nat}.Q(x)).$$

Define the function F as follows.

$$\begin{aligned} F :\equiv \lambda g.\lambda n{:}\mathtt{nat}.\mathtt{case}\ P(n)\ \mathtt{of}\ \ \mathtt{inl}\ (p) \Rightarrow (\mathtt{case}\ g(n+1)\ \mathtt{of} \\ \mathtt{inl}\ (l) \Rightarrow \mathtt{inl}\ ((\mathsf{Cons}\langle n, p\rangle l) \\ \mathtt{inr}\ (m) \Rightarrow \mathtt{inr}\ (m)) \\ \mathtt{inr}\ (p) \Rightarrow \mathtt{inr}\ (n). \end{aligned}$$

Here, the type of g is $\mathtt{nat}{\to}(\mathsf{List} + \mathtt{nat})$, with List as above, the type of lists over $\Sigma x{:}\mathtt{nat}.Q(x)$. (For readability we have omitted the Set-parameter in Cons.) So,

$$F : (\mathtt{nat}{\to}(\mathsf{List} + \mathtt{nat})){\to}(\mathtt{nat}{\to}(\mathsf{List} + \mathtt{nat})).$$

Now, iterating $F\ N+1$ times on $\lambda x{:}\mathtt{nat}.\mathtt{inl}\ (\mathsf{Nil})$ will either result in a sequence

$$[\langle 0, p_0\rangle, \langle 1, p_1\rangle, \ldots, \langle N, p_N\rangle]$$

with $p_i : Q(i)$ for each i, or in a term $n : \mathtt{nat}$ with $n \leq N$, $Pn \longrightarrow\!\!\!\!\!\rightarrow_{\beta\iota} \mathtt{inr}\ (p)$ and $p : \neg Q(n)$. Obviously, in this example one will never wish to use the fixed point combinator, as we are doing a *bounded* search.

4 Meta-theory of Pure Type Systems with Y

Most of the meta-theoretical properties of PTSs are not affected by the inclusion of a fixpoint combinator Y. (The obvious exception is strong normalization, of course!) In particular:

Proposition 7 (Church-Rosser ($\mathbf{CR}_Y$)).
The βY-reduction is Church-Rosser on the set of pseudoterms T.

Proposition 8 (Correctness of Types ($\mathbf{CT}_Y$)).
If $\Gamma \vdash_{S+Y} M : A$ then $\Gamma \vdash_{S+Y} A : s$ or $A \equiv s$ for some some $s \in \mathcal{S}$.

Proposition 9 (Subject Reduction ($\mathbf{SR}_Y$)).
If $\Gamma \vdash_{S+Y} M : A$ and $M \longrightarrow\!\!\!\!\!\rightarrow_{\beta Y} N$, then $\Gamma \vdash_{S+Y} N : A$.

Proposition 10 (Unicity of Types for functional PTSs ($\mathbf{UT}_Y$)).
For functional *PTSs, if $\Gamma \vdash_{S+Y} M : A$ and $\Gamma \vdash_{S+Y} M : B$, then $A =_{\beta Y} B$.*

In addition to the properties above, to prove conservativity of Y we also need the (very basic) ones below.

Proposition 11 (Π_Y-generation). *Let $\Gamma \vdash_{S+Y} \Pi x{:}A.B : s$. Then there exists a rule $(s_1, s_2, s) \in \mathcal{R}$ such that $\Gamma \vdash_{S+Y} A : s_1$ and $\Gamma, x{:}A \vdash_{S+Y} B : s_2$*

Proposition 12 (axiom$_Y$-generation).
Let $\Gamma \vdash_{S+Y} s : s'$ with $s, s' \in \mathcal{S}$. Then $(s, s') \in \mathcal{A}$.

All the properties of PTSs with Y above can be proved in exactly the same way as for PTSs. The trick to proving Conservativity (1 below) is to prove the following, slightly weaker, property. A direct proof of Conservativity by induction on derivations fails.

Lemma 1. *For* functional *PTSs, if $\Gamma \vdash_{S+Y} M : A$ with Γ and M not containing Y, then $\Gamma \vdash M : A'$ for some A' with $A \longrightarrow\!\!\!\!\!\rightarrow_{\beta Y} A'$.*

Proof. Induction on the derivation of $\Gamma \vdash_{S+Y} M : A$. The interesting cases are the abstraction and application rule:

- Suppose the last step in the derivation is

$$\frac{\Gamma \vdash_{S+Y} M : \Pi x{:}A.B \quad \Gamma \vdash_{S+Y} N : A}{\Gamma \vdash_{S+Y} MN : B[N/x]}$$

  By the IH $\Gamma \vdash M : C$ for some C with $\Pi x{:}A.B \longrightarrow\!\!\!\!\!\rightarrow_{\beta Y} C$. So, C is a Π-abstraction, say $C \equiv \Pi x{:}A'.B'$. Then $A \longrightarrow\!\!\!\!\!\rightarrow_{\beta Y} A'$ and $\longrightarrow\!\!\!\!\!\rightarrow_{\beta Y} B'$. By Proposition 6, $\Gamma \vdash A' : s_1$ for some s_1. By the IH $\Gamma \vdash N : A''$ for some A'' with $A \longrightarrow\!\!\!\!\!\rightarrow_{\beta Y} A''$. As $A' =_{\beta Y} A''$ and A', A'' do not contain Y, we conclude $A' =_\beta A''$ (using $\mathrm{CR}_{\beta Y}$). Hence $\Gamma \vdash N : A'$ by the (conv) rule. Now, $\Gamma \vdash MN : B'[N/x]$ by the (app) rule and indeed $B[N/x] \longrightarrow\!\!\!\!\!\rightarrow_{\beta Y} B'[N/x]$.
- Suppose the last step in the derivation is

$$\frac{\Gamma, x{:}A \vdash_{S+Y} M : B \quad \Gamma \vdash_{S+Y} \Pi x{:}A.B : s}{\Gamma \vdash_{S+Y} \lambda x{:}A.M : \Pi x{:}A.B}$$

  By the IH on the first premise $\Gamma, x{:}A \vdash M : B'$ for some B' with $B \longrightarrow\!\!\!\!\!\rightarrow_{\beta Y} B'$. Unfortunately we cannot use the IH on the second premise – $\Gamma \vdash_{S+Y} \Pi x{:}A.B : s$ – as $\Pi x{:}A.B$ may contain Y. We reason as follows: $\Gamma \vdash A : s_1$ (for some s_1). By CT (Proposition 2), $\Gamma, x{:}A \vdash B' : s_2$ for some s_2 (i) or $B' \equiv s'$ for some s' (ii). Looking at these two cases:
  (i) As $\vdash_S \subseteq \vdash_{S+Y}$ we know $\Gamma \vdash_{S+Y} A : s_1$ and $\Gamma, x : A \vdash_{S+Y} B' : s_2$. Using SR_Y we find $\Gamma \vdash_{S+Y} \Pi x{:}A.B' : s$. Combining this with Proposition 11 and UT_Y we conclude $(s_1, s_2, s) \in \mathcal{R}$. So $\Gamma \vdash \Pi x{:}A.B' : s$.
  (ii) As $\vdash_S \subseteq \vdash_{S+Y}$ we know $\Gamma \vdash_{S+Y} A : s_1$. Using SR_Y we find $\Gamma \vdash_{S+Y} \Pi x{:}A.B' : s$. Combining this with Proposition 11 and UT_Y we conclude $(s_1, s_2, s) \in \mathcal{R}$ and $\Gamma, x : A \vdash_{S+Y} B' : s_2$ for some s_2. Since $B' \equiv s'$, $s' : s_2$ must be an axiom, so $\Gamma, x : A \vdash B' : s_2$. Hence $\Gamma \vdash \Pi x{:}A.B' : s$.
  Now $\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B'$ by the (λ) rule and $\Pi x{:}A.B \longrightarrow\!\!\!\!\!\rightarrow_{\beta Y} \Pi x{:}A.B'$.

Conservativity is an easy consequence of the lemma above.

Corollary 1 (Conservativity for functional PTSs).
Consider a functional PTS. Let $\Gamma \vdash_{S+Y} M : A$ with Γ, M, and A not containing Y. Then $\Gamma \vdash M : A$.

Proof. By the previous lemma $\Gamma \vdash M : A'$ for some A' with $A \longrightarrow\!\!\!\!\!\rightarrow_{\beta Y} A'$. By CT_Y we have $\Gamma \vdash_{S+Y} A : s$ or $A \equiv s$ for some $s \in \mathcal{S}$. In the second case, $A' \equiv A \in \mathcal{S}$, so $\Gamma \vdash M : A$. In the first case, $\Gamma \vdash A : s$ by the previous Lemma, so $\Gamma \vdash M : A$ by the conversion rule.

With respect to the issue of *decidability of type inference*: in general, the addition of a fixed point combinator will make type inference undecidable. This is because Y allows us to define all partial recursive functions. So, if $F : \mathtt{nat}\to\mathtt{nat}$ is some closed term, representing a partial recursive function and we take $\Gamma = P : \mathtt{nat}\to\mathsf{Prop}, x : P0$, then

$$x : P(F0) \text{ iff } F0 = 0.$$

So, a type inference algorithm would give us a decision algorithm for the value of partial recursive functions on 0, quod non: type inference is undecidable.

Nevertheless, we still want to be able to edit the term YF that defines our proof search. That is, we would like to be able to interactively construct YF and have it type checked by the proof engine. If we look back at the examples in Section 3, we see that the 'proof search terms' YF that are given here *can* be type checked: If we apply the usual type-checking algorithm to these terms, the Y-reduction, which is the only possible source for infinite reductions (and hence undecidability), is never used.

To be more precise: the proof search terms that we have constructed can all be type-checked in the system λPREDω, where Y is treated as a constant that takes a term of type $A\to A$ to a term of type A. (The λPREDω-type-checking algorithm applies immediately to this small extension. Alternatively one could extend λPREDω with the rule $(\mathsf{Type}^s, \mathsf{Set})$. Then put $Y : \Pi\alpha{:}\mathsf{Set}.(\alpha\to\alpha)\to\alpha$ in the context and use the type-checking algorithm for this extension of λPREDω.)

This is a general situation: in the phase of constructing the proof search term we can treat Y as a constant (without reduction behaviour). So then we are dealing with well-known type systems. When we have constructed the proof search term, we let it reduce and if this results in a normal form, the conservativity property, Corollary 1, guarantees that we have found a proof in the original type system.

5 Conclusions and Related Work

We have presented a method for proof search inside the proof system of higher order predicate logic with inductive types. We have tested our method by some examples using the proof engine Coq. See [Zwanenburg e.a. 1999] for the examples; the methods turn out to be reasonably fast. In our first example we are looking for a 'witness' n of the property Q, using $Y_I^N F0$, which iterates F up to N times, starting from 0. One could do this similarly in the meta language of the

proof system (the implementation language), which may be faster, but also requires a lot of knowledge of the implementation and experience in programming in the meta-language.

We have also presented the underlying theory, why doing an unbounded search (by adding a fixed point combinator) does not spoil the logical proof system. The addition of a fixed point combinator to the Calculus of Constructions (CC) has also previously been studied in [Audebaud 1991]. His goal is to overcome the problem with the second order definable datatypes in CC, so he is using the fixed point mainly to be able to define data types (of type Set in our system) that have the desirable properties. We don't have to do that, because we use the extension with inductive types, which provides us with the necessary data types. Moreover, we are especially interested in using the fixed point combinator to define (potentially) infinite computations to search for witnesses and proof-objects.

References

[Audebaud 1991] P. Audebaud, Partial Objects in the Calculus of Constructions, in *Proceedings of the Sixth Annual Symp. on Logic in Computer Science*, Amsterdam 1991, IEEE, pp. 86 – 95.

[Barendregt 1992] H.P. Barendregt, Lambda calculi with Types. In *Handbook of Logic in Computer Science*, eds. Abramski et al., Oxford Univ. Press, pp. 117 – 309.

[Berardi 1990] S. Berardi, Type dependence and constructive mathematics, Ph.D. thesis, Universita di Torino, Italy.

[Coquand and Mohring 1990] Th. Coquand and Ch. Paulin-Mohring Inductively defined types, In P. Martin-Löf and G. Mints editors. *COLOG-88 : International conference on computer logic, LNCS 417.*

[Dowek e.a. 1991] G. Dowek, A. Felty, H. Herbelin, G. Huet, Ch. Paulin-Mohring, B. Werner, The Coq proof assistant version 5.6, user's guide. INRIA Rocquencourt - CNRS ENS Lyon.

[Geuvers 1993] H. Geuvers, Logics and Type Systems, Ph.D. Thesis, University of Nijmegen, 1993.

[Geuvers and Nederhof 1991] J.H. Geuvers and M.J. Nederhof, A modular proof of strong normalisation for the calculus of constructions. *Journal of Functional Programming*, vol 1 (2), pp 155-189.

[Terlouw 1989a] J. Terlouw, Een nadere bewijstheoretische analyse van GSTT's (incl. appendix), Manuscript, Faculty of Mathematics and Computer Science, University of Nijmegen, Netherlands, March, April 1989. (In Dutch)

[Zwanenburg e.a. 1999] J. Zwanenburg and H. Geuvers, Example of Proof Search by iteration in Coq, url:
`http://www.cs.kun.nl/~janz/proofs/proofSearch/index.html`

Monadic Presentations of Lambda Terms Using Generalized Inductive Types

Thorsten Altenkirch and Bernhard Reus

Ludwig-Maximilians-Universität, Oettingenstr. 67, 80538 München, Germany,
{alti,reus}@informatik.uni-muenchen.de
Phone: +49 89 2178 {2209,2178} Fax: +49 89 2178 {2238,2175}

Abstract. We present a definition of untyped λ-terms using a heterogeneous datatype, i.e. an inductively defined operator. This operator can be extended to a Kleisli triple, which is a concise way to verify the *substitution laws* for λ-calculus. We also observe that repetitions in the definition of the monad as well as in the proofs can be avoided by using well-founded recursion and induction instead of structural induction. We extend the construction to the simply typed λ-calculus using dependent types, and show that this is an instance of a generalization of Kleisli triples. The proofs for the untyped case have been checked using the LEGO system.
Keywords. Type Theory, inductive types, λ-calculus, category theory.

1 Introduction

The metatheory of substitution for λ-calculi is interesting maybe because it seems intuitively obvious but becomes quite intricate if we take a closer look. [Hue92] states seven formal properties of substitution which are then used to prove a general substitution theorem. When formalizing the proof of strong normalisation for System F [Alt93b,Alt93a] the first author formally verified five substitution properties quite similar to those of [Hue92].

Therefore it seems a good idea to look for a more general and elegant way to state and verify the substitution laws. Obviously, this is also related to the way lambda terms are presented.

We find a partial answer in the work of Bellegarde and Hook [BH94] who take the view that lambda terms should be represented by an operator $\mathrm{Lam} \in \mathbf{Set} \to \mathbf{Set}$, where **Set** denotes the universe of sets, such that $\mathrm{Lam}(X)$ is the set of λ-terms with variables in X. This corresponds to the presentation of terms in universal algebra as an operator $\mathrm{Term} \in \mathbf{Set} \to \mathbf{Set}$. The substitution laws are captured by verifying that Lam can be extended to a monad or equivalently to a *Kleisli triple* (cf. Section 2.1, see also [Man76,Mog91]).

In this paper we are going to revise and extend the work of Bellegarde and Hook in the following ways:

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 453–468, 1999.

- The presentation of Lam, see Section 3.2, is improved by using a *heterogeneous datatype*[1], i.e. there are no meaningless terms in our representation. Heterogeneous datatypes have already been discussed in [BM98], where they are called *nested datatypes* and modelled by initial algebras in functor categories, which seems unsatisfactory. Building on this approach, in [BP99] heterogeneous definitions of untyped λ-terms are investigated.
- Repetitions in the definition of the monad and in the verification can be avoided by using well founded recursion (along a primitive recursive well-ordering) instead of structural recursion, see section 4.
- The development has been verified using the LEGO system, see section 4.5.
- We also extend this approach to the simply typed λ-calculus, see Section 5. To do this we present a generalization of Kleisli triples, which we call Kleisli structures, see 5.1.
- We analyze the type of inductive definitions needed in every step of the formalization using initial algebras of functors. We consider two generalizations of the usual scheme of inductive definitions: heterogeneous (see 3.1) and dependent inductive definitions (see Section 5.2).

Our work seems to be closely related to recent work by Fiore, Plotkin and Turi [FPT99] who pursue a more abstract algebraic treatment of signatures with binders but do not cover the simply typed case. Higher order syntax can also be used to represent λ-terms, i.e. in [Hof99].

2 Preliminaries

As a metatheory we use an informal version of extensional Type Theory, details can be found in [Mar84,Hof97]. Since we do not exploit the proposition-as-types principle we work in a system quite close to conventional intuitionistic set theory. We use **Set** and **Prop** to denote the types of sets and propositions.

Notationally, we adopt the following conventions: We write the type of implicit parameters of dependent functions as subscripts, i.e. $\Pi_{n\in\mathrm{Nat}}\mathrm{Fin}(n) \to \mathbf{Set}$ is a type of functions whose first argument is usually omitted. The hidden argument can be made explicit by putting it in subscript, i.e. we write e.g. $f_X \in T(X)$ when we mean $f \in \Pi_{X\in C}T(X)$ for some type $C \in \mathbf{Set}$ obvious from the context. Given $P, Q \in A \to \mathbf{Prop}$ we write $P \subseteq Q$ for $\forall a \in A.P(a) \to Q(a)$. Given a curried function $f \in A_1 \to A_2 \to \ldots \to A_n$ we write the application to a sequence of arguments $a_1, a_2, \ldots, a_n$ as $f(a_1, a_2, \ldots, a_n)$. The same convention holds for Π-types.

The rest of this section briefly reviews Kleisli triples, initial algebras, and inductive datatypes and might be skipped by the experienced reader.

2.1 Kleisli Triples

We present monads as Kleisli triples, i.e.

[1] It seems that the idea for this presentation goes back to Hook, but he didn't use it in the paper because it cannot be implemented in SML.

Definition 1. *A Kleisli-Triple* $(T, \eta^T, \mathrm{bind}^T)$ *on a category* $\mathbf{C}$ *is given by*

- *an function on the objects:* $T \in |\mathbf{C}| \to |\mathbf{C}|$
- *a family of morphisms indexed by objects* $X \in |\mathbf{C}|$*:* $\eta^T_X \in \mathbf{C}(X, T(X))$
- *a family of functions indexed by* $X, Y \in |\mathbf{C}|$*:*

$$\mathrm{bind}^T_{X,Y} \in \mathbf{C}(X, T(Y)) \to \mathbf{C}(T(X), T(Y))$$

which are subject to the following equations:

1. $\mathrm{bind}^T_{X,X}(\eta^T_X) = 1_{T(X)}$
2. $\mathrm{bind}^T_{X,Y}(f) \circ \eta^T_X = f$ *where* $f \in \mathbf{C}(X, T(Y))$.
3. $\mathrm{bind}^T_{X,Z}(\mathrm{bind}^T_{Y,Z}(f) \circ g) = \mathrm{bind}^T_{Y,Z}(f) \circ \mathrm{bind}^T_{X,Y}(g)$
   where $f \in \mathbf{C}(Y, T(Z)), g \in \mathbf{C}(X, T(Y))$.

Kleisli triples were introduced in [Man76], where they are also shown to be equivalent to the conventional presentation of monads, see [ML71], pp.133.

2.2 Initial Algebras

Definition 2. *For any endofunctor* $T : \mathbf{C} \to \mathbf{C}$ *an initial* T*-algebra* $(\mu^T, \mathrm{c}^T, \mathrm{It}^T)$ *is given by*

- *an object* $\mu^T \in |\mathbf{C}|$
- *a morphism* $\mathrm{c}^T \in \mathbf{C}(T(\mu^T), \mu^T)$
- *a family of functions indexed by* $X \in |\mathbf{C}|$*:* $\mathrm{It}^T_X \in \mathbf{C}(T(X), X) \to \mathbf{C}(\mu^T, X)$

such that given a T*-algebra* $f \in \mathbf{C}(T(X), X)$*:*

$$\begin{array}{ccc} T(\mu^T) & \xrightarrow{\mathrm{c}^T} & \mu^T \\ {\scriptstyle T(\mathrm{It}^T_X(f))}\downarrow & & \downarrow{\scriptstyle \mathrm{It}^T_X(f)} \\ T(X) & \xrightarrow[f]{} & X \end{array}$$

commutes and $\mathrm{It}^T_X(f)$ *is the unique morphism with this property, i.e. given any* $h \in \mathbf{C}(\mu^T, X)$ *we have* $h = \mathrm{It}^T_X(f)$.

μ^T *is called weakly initial if* $\mathrm{It}^T_X(f)$ *exists but is not necessarily unique.*

We assume that our ambient category **Set** is bicartesian closed, i.e. has finite products $\mathbf{1}, - \times -$, coproducts $\mathbf{0}, - + -$ and function spaces $- \to -$. We say that a variable appears strictly positive in a type, if it appears never on the left hand side of an arrow type, and positive, if it appears only on the left hand side of an even number of nested arrow types.

2.3 Inductive Datatypes

To model inductive types we introduce the concept of a strictly positive operator, i.e. a function $T \in \mathbf{Set} \to \mathbf{Set}$ which is given by a definition $T(X) = \sigma(X)$ such that X appears strictly positive in $\sigma(X)$. Here we write $\sigma(X)$ for a syntactic type

expression in which the variable X may occur. Every strictly positive operator gives rise to an endofunctor on **Set**.

Given a strictly positive operator T we introduce $\mu^T = \mu X.T(X) \in \mathbf{Set}$ to denote the initial T-algebra. We extend strictly positive to μ-types, s.t. μ-types can be used to define new operators. We say that **Set** has all strictly positive[2] datatypes if all initial algebras defined by a strictly positive operator exist. This gives rise to a λ-calculus λ^μ, e.g. see [Alt98].

Examples for inductive datatypes are natural numbers $\mathrm{Nat} = \mu X.\mathbf{1} + X$, ordinal notations $\mathrm{Ord} = \mu X.\mathbf{1} + X + (\mathrm{Nat} \to X)$ or finitely branching trees $\mathrm{Tree} = \mu X.\mu Y.\mathbf{1} + X \times Y$.

Datatypes can be conveniently presented by constructors and their types, i.e. $\mathrm{Nat} \in \mathbf{Set}$ can be presented as $0 \in \mathrm{Nat}$ and $\mathrm{succ} \in \mathrm{Nat} \to \mathrm{Nat}$, analogously $\mathrm{Ord} \in \mathbf{Set}$ is given by $0' \in \mathrm{Nat}$, $\mathrm{succ}' \in \mathrm{Nat} \to \mathrm{Nat}$ and $\mathrm{lim} \in (\mathrm{Nat} \to \mathrm{Ord}) \to \mathrm{Ord}$. Nested types like Tree correspond to simultaneous inductive definitions, i.e. $\mathrm{Tree}, \mathrm{Forest} \in \mathbf{Set}$ is given by $\mathrm{nil} \in \mathrm{Forest}$, $\mathrm{cons} \in \mathrm{Tree} \to \mathrm{Forest} \to \mathrm{Forest}$, and $\mathrm{span} \in \mathrm{Forest} \to \mathrm{Tree}$.

Parametrized datatypes like lists can be defined as a function $\mathrm{List} \in \mathbf{Set} \to \mathbf{Set}$ given by $\mathrm{List}(X) = \mu Y.\mathbf{1} + Y \times X$. List is homogeneous because the parameter X does not change in the inductive definition.

Assuming weak initiality, the uniqueness property can be alternatively expressed by an induction principle, i.e. given a predicate $P \in \mu^T \to \mathbf{Prop}$ we have

$$\frac{\mathrm{c}^T(P) \subseteq P}{\forall x \in \mu^T.P(x)}\,\mathrm{Dat-Ind}$$

where $\mathrm{c}^T(P) = \{\mathrm{c}^T(x) \mid x \in P\}$.

It is well known that all positive inductive types can be encoded impredicatively (i.e. in System F, [GLT89]):

$$\begin{aligned}
\mu X.T(X) &= \Pi X \in \mathbf{Set}.(T(X) \to X) \to X \qquad \in \mathbf{Set}\\
\mathrm{It}^T &= \lambda X \in \mathbf{Set}.\lambda f \in T(X) \to X.\lambda x \in \mu X.T(X).\, x\, X\, f\\
&\in \Pi X \in \mathbf{Set}.(T(X) \to X) \to \mu^T \to X\\
\mathrm{c}^T &= \lambda x \in T(\mu X.T(X)).\lambda X \in \mathbf{Set}.\lambda f \in T(X) \to X.f(T(\mathrm{It}^T\ X\ f)\ x)\\
&\in T(\mu^T) \to \mu^T
\end{aligned}$$

Here $T(-) \in \Pi_{X,Y \in \mathbf{Set}}(X \to Y) \to T(X) \to T(Y)$ is the morphism part of the functor which can be derived from the fact that it is given by a positive definition. This encoding is weakly initial, uniqueness can be derived from parametricity [Wad89,AP93].

3 λ-Terms as a Heterogeneous Datatype

3.1 Heterogeneous Inductive Datatypes

We interpret heterogeneous datatypes by initial algebras in the category of families of sets **Fam**. Objects in **Fam** are families $F \in \mathbf{Set} \to \mathbf{Set}$ and given families

[2] *Strictly positive* can be replaced by *positive*, but it is not obvious whether this extension is still predicative.

$F, G \in |\mathbf{Fam}|$ morphisms are families of functions $f \in \Pi X \in \mathbf{Set}.\, F(X) \to G(X)$. A strictly positive operator on families is a function $H \in (\mathbf{Set} \to \mathbf{Set}) \to (\mathbf{Set} \to \mathbf{Set})$ which is given by a definition $H(F) = \lambda X \in \mathbf{Set}.\, \sigma(F, X)$ where F appears only strictly positive in $\sigma(F, X)$. Every strictly positive operator on families gives rise to an endofunctor on **Fam**.

Given a strictly positive operator H on families there exists an initial algebra $\mu^H = \mu F.\lambda X \in \mathbf{Set}.\, H(F, X) \in \mathbf{Set} \to \mathbf{Set}$. As before we define operators and inductive types simultaneously. The constructors c^H and It^H now refer to morphisms in **Fam** — this can be spelt out as follows:

$$\mathrm{c}^H \in \Pi_{X \in \mathbf{Set}} H(\mu^H, X) \to \mu^H(X)$$
$$\mathrm{It}^H \in \Pi_{F \in \mathbf{Set} \to \mathbf{Set}} (\Pi_{X \in \mathbf{Set}} H(F, X) \to F(X)) \to \Pi_{X \in \mathbf{Set}} \mu^H(X) \to F(X)$$

The uniqueness property of the inductively defined operator can be also expressed by the following induction principle: Assume a family of predicates $P \in \Pi_{X \in \mathbf{Set}} \mu^H(X) \to \mathbf{Prop}$:

$$\frac{\forall Y \in \mathbf{Set}.\ \mathrm{c}^H(P_Y) \subseteq P_Y}{\forall Y \in \mathbf{Set}. \forall x \in (\mu X.H(X))(Y).P(x)}\ \mathrm{Het} - \mathrm{Ind}$$

The λ-calculus corresponding to heterogeneous polymorphic definitions has to our knowledge not yet been explored.

Positive heterogeneous inductive types can be encoded impredicatively (i.e. in System F^ω):

$$\begin{aligned}
\mu^H &= \lambda Y \in \mathbf{Set}.\Pi F \in \mathbf{Set} \to \mathbf{Set}.(\Pi_{X \in \mathbf{Set}} H(F, X) \to F(X)) \to F(Y) \\
&\in \mathbf{Set} \to \mathbf{Set} \\
\mathrm{It}^H &= \lambda F \in \mathbf{Set} \to \mathbf{Set}.\, \lambda f \in \Pi_{X \in \mathbf{Set}} H(F, X) \to F(X).\lambda X \in \mathbf{Set}. \\
&\qquad \lambda x \in \mu^H.\ x(F, f) \\
&\in \Pi F \in \mathbf{Set} \to \mathbf{Set}.\, (\Pi_{X \in \mathbf{Set}} T(F, X) \to F(X)) \\
&\qquad \to \Pi_{X \in \mathbf{Set}} \mu F.H(F, X) \to F(X) \\
\mathrm{c}^H &= \lambda X \in \mathbf{Set}.\, \lambda x \in T(\mu^H, X).\, \lambda F \in \mathbf{Set} \to \mathbf{Set}. \\
&\qquad \lambda f \in \Pi_{X \in \mathbf{Set}} H(F, X) \to F(X).f(H(\mathrm{It}^H, F, f), x) \\
&\in \Pi_{X \in \mathbf{Set}} H(\mu^H, X) \to \mu^H(X)
\end{aligned}$$

3.2 Definition of Lam

An example for a heterogeneous inductive datatype is the operator $\mathrm{Lam} \in \mathbf{Set} \to \mathbf{Set}$ from the introduction which can be defined as

$$\mathrm{Lam} = \mu F \in \mathbf{Set} \to \mathbf{Set}.\lambda X \in \mathbf{Set}.X + (F(X) \times F(X)) + F(X_\perp)$$

where $X_\perp \cong 1 + X$ with two constructors $\mathrm{new} \in \Pi X \in \mathbf{Set}.X_\perp$ and $\mathrm{old} \in \Pi_{X \in \mathbf{Set}} X \to X_\perp$ and eliminator $\mathrm{case} \in \Pi_{X,Y \in \mathbf{Set}} Y \to (X \to Y) \to X_\perp \to Y$. Clearly $(\_)_\perp$ gives rise to a functor.

As before we can present inductively defined operators by giving the constructors, which in the case of Lam read as follows:

$$\begin{aligned}\mathrm{var} &\in \Pi_{X\in\mathbf{Set}} X \to \mathrm{Lam}(X)\\ \mathrm{app} &\in \Pi_{X\in\mathbf{Set}} \mathrm{Lam}(X) \to \mathrm{Lam}(X) \to \mathrm{Lam}(X)\\ \mathrm{abst} &\in \Pi_{X\in\mathbf{Set}} \mathrm{Lam}(X_\bot) \to \mathrm{Lam}(X)\end{aligned}$$

4 Lam is Monadic

To show that Lam has the structure of a Kleisli triple we first have to define η^{Lam}_X and $\mathrm{bind}^{\mathrm{Lam}}_{X,Y}$. The former is simply var_X and the latter can be defined *recursively* or *structural inductively* which gives rise to two different constructions.

4.1 The Recursive Construction

In this case bind and an auxiliary map lift

$$\begin{aligned}\mathrm{lift} &\in \Pi_{X,Y\in\mathbf{Set}} (X \to \mathrm{Lam}(Y)) \to X_\bot \to \mathrm{Lam}(Y_\bot)\\ \mathrm{bind} &\in \Pi_{X,Y\in\mathbf{Set}} (X \to \mathrm{Lam}(Y)) \to \mathrm{Lam}(X) \to \mathrm{Lam}(Y)\end{aligned}$$

are defined by simultaneous recursion. The equations defining lift and bind recursively are given below.

$$\begin{aligned}\mathrm{lift}(f, \mathrm{new}(X)) &= \mathrm{var}(\mathrm{new}(Y))\\ \mathrm{lift}(f, \mathrm{old}(x)) &= \mathrm{bind}(\mathrm{var} \circ \mathrm{old}, f(x))\\ \mathrm{bind}(f, \mathrm{var}(x)) &= f(x)\\ \mathrm{bind}(f, \mathrm{app}(s,t)) &= \mathrm{app}(\mathrm{bind}(f,s), \mathrm{bind}(f,t))\\ \mathrm{bind}(f, \mathrm{abst}(t)) &= \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(f), t))\end{aligned}$$

We must first prove that bind is terminating.

Definition 3. *Let* $f \in A \to \mathrm{Lam}(B)$ *for arbitrary* $A, B \in \mathbf{Set}$ *then let* $\mathrm{isVar}(f) \Leftrightarrow \exists h : A \to B.\, f = \mathrm{var}_B \circ h$ *and*

$$v(f) = \begin{cases} 0 & \textit{if} \quad \mathrm{isVar}(f)\\ 1 & \textit{otherwise}\end{cases}$$

Now we are in a position to define a termination order for bind. For any recursive call $\mathrm{bind}(f', t')$ inside of $\mathrm{bind}(f, t)$ we must have $(f', t') <_b (f, t)$. To that end we define

$$(f', t') <_b (f, t) \Leftrightarrow v(f) < v(f') \vee (v(f) = v(f') \wedge t <_s t')$$

where $<_s$ is the structural order on terms. As $<_b$ is the lexicographic order on two well-founded orders we immediately get the following observation.

Proposition 1. *The order $<_b$ is well-founded.*

For the termination of bind the fact below is important.

Proposition 2. *For any f of appropriate type it holds that $v(\mathrm{lift}(f)) \leq v(f)$.*

Proof. Assume that $v(f) = 0$ hence $f = \mathrm{var} \circ h$. By case analysis it is easily verified that $\mathrm{lift}(f) = \mathrm{var} \circ \mathrm{case}(\mathrm{new}, \mathrm{old} \circ h)$, hence $v(\mathrm{lift}(f)) = 0$. Thus, we have shown that $v(\mathrm{lift}(f)) \leq v(f)$.

Proposition 3. bind *is a terminating function.*

Proof. The only difficult case is $\mathrm{bind}(f, \mathrm{abst}(t)) = \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(f), t))$. Since $v(\mathrm{lift}(f)) \leq v(f)$ and $t <_s \mathrm{abst}(t)$ we get that $(\mathrm{lift}(f), t) <_b (f, \mathrm{abst}(t))$.

Condition 1. of Definition 1 holds by definition of bind.

Proposition 4. *Condition 2. of Definition 1 holds, i.e.*

$$\forall t \in \mathrm{Lam}(X).\, \mathrm{bind}(\mathrm{var}_X, t) = t.$$

Proof. Proof by structural induction on t: The var_X-case is trivial. Assume that $t = \mathrm{app}(a, b)$ and $\mathrm{bind}(\mathrm{var}_X, a) = a$ and $\mathrm{bind}(\mathrm{var}_X, b) = b$. Thus we obtain

$$\mathrm{bind}(\mathrm{var}_X, \mathrm{app}(a\, b)) = \mathrm{app}(\mathrm{bind}(\mathrm{var}_X, a), \mathrm{bind}(\mathrm{var}_X, b)) = \mathrm{app}(a, b).$$

Finally, assume that $t = \mathrm{abst}(s)$ and that $\mathrm{bind}(\mathrm{var}_{X_\perp}, s) = s$. Then

$$\begin{aligned} &\mathrm{bind}(\mathrm{var}_X, \mathrm{abst}(s)) = \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(\mathrm{var}_X), s)) = \mathrm{abst}(\mathrm{bind}(\mathrm{var}_{X_\perp}, s)) \\ &= \mathrm{abst}(s) = t \qquad \text{by induction hypothesis.} \end{aligned}$$

Proposition 5. *Condition 3. of Definition 1 holds:*

$$\forall f \in A \to \mathrm{Lam}(B).\, \forall g \in B \to \mathrm{Lam}(C).\, \mathrm{bind}(g) \circ \mathrm{bind}(f) = \mathrm{bind}(\mathrm{bind}(g) \circ f)$$

Proof. Using extensionality and well-founded induction this amounts to prove three cases: The var and the app-cases are again easy. We concentrate on the abst-case.

$$\begin{aligned} (\mathrm{bind}(g) \circ \mathrm{bind}(f))\,(\mathrm{abst}(t)) &= \mathrm{bind}(g, \mathrm{bind}(f, \mathrm{abst}(t))) \\ &= \mathrm{bind}(g, \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(f), t))) \\ &= \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(g), \mathrm{bind}(\mathrm{lift}(f), t))) \\ &= \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(g) \circ \mathrm{bind}(\mathrm{lift}(f)), t)) \quad \text{(ind.hyp.)} \\ &= \mathrm{abst}(\mathrm{bind}(\mathrm{bind}(\mathrm{lift}(g) \circ \mathrm{lift}(f)), t)). \end{aligned}$$

On the other hand $\mathrm{bind}(\mathrm{bind}(g) \circ f, \mathrm{abst}(t)) = \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(\mathrm{bind}(g) \circ f), t))$ such that it remains to show

$$\mathrm{lift}(\mathrm{bind}(g) \circ f) = \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{lift}(f)$$

which is proved by extensionality and case analysis on the argument. First if the argument is a "new" variable then by definition of lift and bind:

$$\mathrm{bind}(\mathrm{lift}(g) \circ \mathrm{lift}(f), \mathrm{new}(A)) = \mathrm{lift}(\mathrm{bind}(g) \circ f, \mathrm{new}(A))$$

In the other case we first distinguish whether $\mathrm{isVar}(f)$ holds or not:

1. *Case:* $\mathrm{isVar}(f)$:
   Then there is an $h \in A \to B$ such that $f = \mathrm{var}_B \circ h$.

$$\begin{aligned}
&\mathrm{lift}(\mathrm{bind}(g) \circ (\mathrm{var}_B \circ h)) \circ \mathrm{old}_A \\
&\quad = \mathrm{lift}(g \circ h) \circ \mathrm{old}_A \qquad \text{(Def. bind)} \\
&\quad = \mathrm{bind}(\mathrm{var}_{B_\perp} \circ \mathrm{old}_B) \circ g \circ h \qquad \text{(Def. lift)} \\
&\quad = \mathrm{lift}(g) \circ \mathrm{old}_B \circ h \qquad \text{(Def. bind)} \\
&\quad = \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{var}_{B_\perp} \circ \mathrm{old}_B \circ h \qquad \text{(Def. bind reverse)} \\
&\quad = \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{bind}(\mathrm{var}_{B_\perp} \circ \mathrm{old}_B) \circ \mathrm{var}_B \circ h \qquad \text{(Def. bind reverse)} \\
&\quad = \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{lift}(\mathrm{var}_B \circ h) \circ \mathrm{old}_A \qquad \text{(Def. lift reverse)}
\end{aligned}$$

2. *Case:* $\neg\mathrm{isVar}(f)$:

$$\begin{aligned}
\mathrm{lift}(\mathrm{bind}(g) \circ f) \circ \mathrm{old} &= \mathrm{bind}(\mathrm{var}_{C_\perp} \circ \mathrm{old}_C) \circ \mathrm{bind}(g) \circ f \qquad (*) \\
&= \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{bind}(\mathrm{var}_B \circ \mathrm{old}_B) \circ f \\
&= \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{lift}(f) \circ \mathrm{old}_A
\end{aligned}$$

For $(*)$ it remains to show that

$$\mathrm{bind}(\mathrm{var}_{C_\perp} \circ \mathrm{old}_C) \circ \mathrm{bind}(g) = \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{bind}(\mathrm{var}_{B_\perp} \circ \mathrm{old}_B)$$

which is proved below

$$\begin{aligned}
&\mathrm{bind}(\mathrm{var}_{C_\perp} \circ \mathrm{old}_C) \circ \mathrm{bind}(g) \\
&\quad = \mathrm{bind}(\mathrm{bind}(\mathrm{var}_{C_\perp} \circ \mathrm{old}_C) \circ g) \quad \text{(ind.hyp.)} \\
&\quad = \mathrm{bind}(\mathrm{lift}(g) \circ \mathrm{old}_B) \\
&\quad = \mathrm{bind}(\mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{var}_{B_\perp} \circ \mathrm{old}_B) \quad \text{(Def. bind \& ext.)} \\
&\quad = \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{bind}(\mathrm{var}_{B_\perp} \circ \mathrm{old}_B) \quad \text{(ind.hyp.)}
\end{aligned}$$

The induction hypothesis is used three times. As we do not use structural induction we must give a termination order $<'$ such that when proving

$$(\mathrm{bind}(g) \circ \mathrm{bind}(f))(t) = \mathrm{bind}(\mathrm{bind}(g) \circ f, t)$$

we use the induction hypothesis

$$(\mathrm{bind}(g') \circ \mathrm{bind}(f'))(t') = \mathrm{bind}(\mathrm{bind}(g') \circ f', t')$$

only if $(f', g', t') <' (f, g, t)$ for an appropriate well-founded order $<$. We define this order as follows

$$(f', g', t') <' (f, g, t) \Leftrightarrow (f = f' \wedge g = g' \wedge t' <_s t) \vee (v(f') + v(g') < v(f) + v(g)) \,.$$

For the first application of the hypotheses the condition $(f, g, t) <' (f, g, \mathrm{abst}(t))$ holds by the structural order on the last argument. For the second we have to show $(g, \mathrm{var}_{C_\perp} \circ \mathrm{old}_C, s) <' (f, g, s)$ in case $\neg\mathrm{isVar}(f)$ holds. As $\mathrm{isVar}(\mathrm{var}_{C_\perp} \circ k)$ holds for any k, $0 = v(\mathrm{var}_{C_\perp} \circ \mathrm{old}_C) < v(f) = 1$, hence $v(g) + v(\mathrm{var}_{C_\perp} \circ \mathrm{old}_C) < v(f) + v(g)$. The proof of the third case, $(\mathrm{var}_{C_\perp} \circ \mathrm{old}_C, \mathrm{lift}(g), s) <' (f, g, s)$, under the assumption $\neg\mathrm{isVar}(f)$, is similar.

One might argue that the proof is not constructive as we do a case analysis on the undecidable predicate isVar(f). However, we can instead introduce an additional precondition (isVar(f) $\vee$ True) $\wedge$ (isVar(g) $\vee$ True) where True corresponds to *don't know*. We do case analysis over the disjunctions. When using a recursive hypothesis with $f = \mathrm{var} \circ h$ we prove the precondition by a left injection (the same for g).

We summarize the result:

Corollary 1. (Lam$(-)$, var, bind) *is a Kleisli triple.*

4.2 The Construction by Structural Induction

There is also a proof by structural induction. In this case we define bind and lift and also Lam$(-)$ the morphism part of the functor:

$$\begin{aligned}
\mathrm{Lam} \in \Pi_{X,Y \in \mathbf{Set}}(X \to Y) &\to \mathrm{Lam}(X) \to \mathrm{Lam}(Y) \\
\mathrm{lift}(f, \mathrm{new}(X)) &= \mathrm{var}(\mathrm{new}(Y)) \\
\mathrm{lift}(f, \mathrm{old}(x)) &= \mathrm{Lam}(\mathrm{old}, f(x)) \\
\mathrm{Lam}(f, \mathrm{var}(x)) &= \mathrm{var}(f(x)) \\
\mathrm{Lam}(f, \mathrm{app}(s,t)) &= \mathrm{app}(\mathrm{Lam}(f,s), \mathrm{Lam}(f,t)) \\
\mathrm{Lam}(f, \mathrm{abst}(t)) &= \mathrm{abst}(\mathrm{Lam}(f_\perp, t)) \\
\mathrm{bind}(f, \mathrm{var}(x)) &= f(x) \\
\mathrm{bind}(f, \mathrm{app}(s,t)) &= \mathrm{app}(\mathrm{bind}(f,s), \mathrm{bind}(f,t)) \\
\mathrm{bind}(f, \mathrm{abst}(t)) &= \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(f), t))
\end{aligned}$$

Note that bind is defined as in the recursive case, but now lift is not defined in terms of bind so all definitions are structural inductive.

Additional to the propositions shown above, one also needs to show that Lam and $(-)_\perp$ are functorial.

Note that here Lam(h) takes the part of bind(var $\circ$ h) and thus the proof of (*) can be done by structural induction showing first the following two special instances of the third monad law:

$$\begin{aligned}
\forall f \in B \to C. \forall g \in A \to \mathrm{Lam}(B).\, \mathrm{Lam}(f) \circ \mathrm{bind}(g) &= \mathrm{bind}(\mathrm{Lam}(f) \circ g) \\
\forall f \in B \to \mathrm{Lam}(C). \forall g \in A \to B.\, \mathrm{bind}(f) \circ \mathrm{Lam}(g) &= \mathrm{bind}(f \circ g)
\end{aligned}$$

By combining those one immediately gets

$$\forall g \in A \to \mathrm{Lam}(B).\, \mathrm{bind}\,(\mathrm{lift}(g)) \circ \mathrm{Lam}(\mathrm{old}_A) = \mathrm{Lam}(\mathrm{old}_B) \circ \mathrm{bind}(g)$$

and from this one can easily derive (*) in the proof of Proposition 4

$$\mathrm{lift}(\mathrm{bind}(g) \circ f) = \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{lift}(f)\ .$$

The LEGO-code of the structural inductive and the general recursive proof is interesting in the sense that the latter version is only of half the size of the former – without the termination proof though. This emphasizes the significance of type theory *with general recursion* as long as termination can be ensured externally (possibly syntactically).

4.3 Substitution

Once we have bind^{Lam} and η^{Lam} we can define a substitution operator on Lam-terms $\text{subst} \in \Pi_{A \in \mathbf{Set}} \text{Lam}(A_\perp) \to \text{Lam}(A) \to \text{Lam}(A)$ as follows

$$\text{subst}_A(t, s) = \text{bind}(\text{case}(s, \text{var}_A), t)$$

The weakening $\text{weak} \in \Pi_{A \in \mathbf{Set}} \text{Lam}(A) \to \text{Lam}(A_\perp)$ can be written

$$\text{weak}_A = \text{bind}(\text{var}_{A_\perp} \circ \text{old}_A)$$

That substitution and weakening have the right properties follows immediately from the Kleisli properties for bind and var. As an example we show how to derive $\text{subst}(\text{weak}(t), u) = t$:

$$\begin{aligned}
\text{subst}(\text{weak}(t), u) &= \text{bind}(\text{case}(u, \text{var}), \text{bind}(\text{var} \circ \text{old}, t)) \\
&= \text{bind}(\text{bind}(\text{case}(u, \text{var}), \text{var} \circ \text{old}), t) && (3.) \\
&= \text{bind}(\text{case}(u, \text{var}) \circ \text{old}, t) && (2.) \\
&= \text{bind}(\text{var}, t) \\
&= t && (1.)
\end{aligned}$$

The numbers refer to the equations in Definition 1.

4.4 Implementations in Haskell and SML

Heteregeneous datatypes like Lam can be easily implemented in a functional language like Haskell [HJW+92]. The implementation below by Sven Panne also exploits predefined typeclasses like `Monad` and `Functor` in Haskell (where `>>=`, `return`, `Maybe`, `Just`, `Nothing`, `maybe` denote bind, η, $(-)_\perp$, old, new, and case, respectively).

```
data Lam a = Var a
           | App (Lam a) (Lam a)
           | Abs (Lam (Maybe a))

instance Functor Lam where
   fmap f x  =  x >>= return . f

instance Monad Lam where
   return = Var
   Var x   >>= f  =  f x
   App t u >>= f  =  App (t >>= f) (u >>= f)
   Abs t   >>= f  =  Abs (t >>= liftLam f)
```

```
lift :: (Monad b, Functor b) => (a -> b c)
                      -> Maybe a -> b (Maybe c)
lift f Nothing  = return Nothing
lift f (Just x) = fmap Just (f x)

subst :: Monad a => a (Maybe b) -> a b -> a b
subst t u = t >>= maybe u return
```

Although the datatype `Lam` is definable in ML [HMM86], `lift` is not accepted by the ML type system. The reason is that `lift` requires *polymorphic recursion*, which is known to be undecidable. The Haskell type system is more flexible because it does not try to infer the type of function if it is given anyway. There is also an implementation of an improved ML typechecker [EL99] which implements polymorphic recursion via a semialgorithm for semiunification. The corresponding ML-code reads as follows:

```
datatype 'a Lift = new | old of 'a;

datatype 'a Lam = var of 'a | app of ('a Lam)*('a Lam)
  | abs of ('a Lift) Lam;

fun bind f (var x) = f x
  | bind f (app (t,u)) = app (bind f t,bind f u)
  | bind f (abs t) = abs (bind (lift f) t)
and lift f new = var new
  | lift f (old x) = lam old (f x)
and lam f = bind (var o f);

fun subst t u =  bind (fn new => u | old x => var x) t;
fun weak t = lam old t;
```

4.5 Implementation in LEGO

Using the `Inductive`-statement such a heterogeneous datatype can be defined in LEGO [LP92] as follows:

```
Inductive [Lambda:Set->Type] ElimOver Type
Constructors [var:{X|Set}X->Lambda X]
             [app : {X|Set} (Lambda X)->(Lambda X)->(Lambda X)]
             [abst: {X|Set} (Lambda (Lift X)) ->(Lambda X)];
```

In the formalization we assume a constant `ext` which makes the propositional equality extensional and thus destroys the computational adequacy of Type Theory. This problem could be overcome by moving to a Type Theory as described in [Alt99]. The complete LEGO code (for both variants) can be found in [RA99].

5 Extension to Simple Types

5.1 Kleisli Structures

To capture the case of typed algebras, specifically the simply typed λ-calculus, we introduce a generalization of the Kleisli-triples, which we call *Kleisli structure*:

Definition 4. *A Kleisli structure* $(I, F, G, \eta^{F,G}, \mathrm{bind}^{F,G})$ *on a category* $\mathbf{C}$ *is given by*

- *an index set* $I \in \mathbf{Set}$
- *families of objects indexed by* I $F, G \in I \to |\mathbf{C}|$
- *a family of morphisms indexed by* $i \in I$: $\eta_i^{F,G} \in \mathbf{C}(F(i), G(i))$
- *a family of functions indexed by* $i, j \in I$:

$$\mathrm{bind}_{i,j}^{F,G} \in \mathbf{C}(F(i), G(j)) \to \mathbf{C}(G(i), G(j))$$

which are subject to the following equations:

1. $\mathrm{bind}_{i,i}^{F,G}(\eta_i^{F,G}) = 1_{G(i)}$
2. $\mathrm{bind}_{i,j}^{F,G}(f) \circ \eta_i^{F,G} = f$ *where* $f \in \mathbf{C}(F(i), G(j))$.
3. $\mathrm{bind}_{i,k}^{F,G}(\mathrm{bind}_{j,k}^{F,G}(f) \circ g) = \mathrm{bind}_{j,k}^{F,G}(f) \circ \mathrm{bind}_{i,j}^{F,G}(g)$ *where* $f \in \mathbf{C}(F(j), G(k)), g \in \mathbf{C}(F(i), G(j))$.

Kleisli triples are a special case of Kleisli structures where $I = |\mathbf{C}|$ and F is the identity. Writing $\mathbf{C}_F$ for the category whose objects are elements of I and $\mathbf{C}_F(i, j) = \mathbf{C}(F(i), F(j))$ we obtain a functor $T : \mathbf{C}_F \to \mathbf{C}_G$ which is given by the identity on objects and on morphisms $f \in \mathbf{C}_F(i, j)$ by

$$T(f) = \mathrm{bind}_{i,j}^{F,G}(\eta_j^{F,G} \circ f)$$

In the special case of Kleisli triples this is the endofunctor on $\mathbf{C}$ given in section 2.1. Since T is not an endofunctor in general it cannot be a monad.

5.2 Dependent Inductive Types

Next we model dependent inductive types, which are also called inductive families, by initial algebras in categories of families [Dyb94]. Given an index type $I \in \mathbf{Set}$, we define the category of I-indexed families: objects are $F \in I \to \mathbf{Set}$ and morphisms are I-indexed families of functions $f \in \Pi_{i \in I} F(i) \to G(i)$. An inductively defined dependent type is an initial algebra in the category of I-indexed families.

We assume that $\mathbf{Set}$ is also closed under Π-types, Σ-types and Equality types $\mathrm{Eq} \in \Pi_{A \in \mathbf{Set}} A \to A \to \mathbf{Set}$, where $A \in \mathbf{Set}$. We use the usual λ-notation for Π-types. Elements of Σ-types are given by pairs, i.e. given $A \in \mathbf{Set}, B \in A \to \mathbf{Set}$, if $a \in A$ and $b \in B(a)$ then $(a, b) \in \Sigma a \in A.B(a)$. The only inhabitant of an equality type is $\mathrm{refl} \in \Pi_{A \in \mathbf{Set}} \Pi a \in A.\mathrm{Eq}(a, a)$. We assume that the equality type is extensional, i.e. $a = b$ holds iff $\mathrm{Eq}_A(a, b)$ is inhabited. For details see e.g. [Mar84].

We define a strictly positive operator on families as a function $G \in (I \to \mathbf{Set}) \to I \to \mathbf{Set}$ which is given by a definition $G(F) = \lambda i \in I.\sigma(F, i)$ where F appears only strictly positive in $\sigma(F, i)$. Every strictly positive operator gives rise to a functor on the category of I-indexed families.

Given a strictly positive operator G we introduce

$$\mu^G = \mu F \in I \to \mathbf{Set}.\lambda i \in I.G(F,i) \in I \to \mathbf{Set}$$

to denote the initial G-Algebra. As before we define strictly positive operators simultaneously with dependent μ-types such that μ can be used in the definition of new operators. We spell out the types of the constructor and iterator:

$$\mathrm{c}^G \in \Pi i \in I.G(\mu^G, i) \to \mu^G(i)$$
$$\mathrm{It}^G \in \Pi F \in I \to \mathbf{Set}.(\Pi i \in I.G(F,i) \to F(i)) \to \Pi i \in I.\, \mu^G(i) \to F(i)$$

It is convenient to present dependent inductive types by giving the constructors. As an example consider the type of finite sets: $\mathrm{Fin} \in \mathrm{Nat} \to \mathbf{Set}$, $0_{\mathrm{Fin}} \in \Pi n \in \mathrm{Nat}.\,\mathrm{Fin}(\mathrm{succ}(n))$, $\mathrm{succ}_{\mathrm{Fin}} \in \Pi_{n\in\mathrm{Nat}}\mathrm{Fin}(n) \to \mathrm{Fin}(\mathrm{succ}(n))$. This definition can be mechanically translated into the strictly positive operator

$$G_{\mathrm{Fin}}(F \in \mathrm{Nat} \to \mathbf{Set}) = \lambda n \in \mathrm{Nat}.\Sigma m \in \mathrm{Nat}.\mathrm{Eq}(\mathrm{succ}(m), n) \times (\mathbf{1} + F(n)).$$

The type of c^G is isomorphic to the product of the types of 0_{Fin} and $\mathrm{succ}_{\mathrm{Fin}}$. Inductive dependent types which are indexed over several sets, like $\Pi a \in A.B(a) \to \mathbf{Set}$ correspond to μ-types whose index set is a Σ-type, i.e. $\Sigma a \in A.B(a)$.

Inductively defined dependent types can be encoded in the calculus of constructions along the same lines as heterogeneous datatypes, see section 3.1.

As before we can represent the uniqueness condition by an induction principle: Assume a family of predicates $P \in \Pi_{i\in I}\mu^G(i) \to \mathbf{Prop}$:

$$\frac{\forall i \in I.\ \mathrm{c}^G(P(i)) \subseteq P(i)}{\forall i \in I.\forall x \in \mu^G(i).P(x)}\ \mathrm{Dep-Ind}$$

In Type Theory it is standard to use a dependent iterator which captures both induction and iteration.

Heterogeneous datatypes as introduced previously can be seen as an instance of dependent inductive types if we assume the existence of a universe $\mathrm{U} \in \mathbf{Set}$ which reflects all the type formers introduced so far.

5.3 The Definition of Lam for Simple Types

To extend the previous construction to simply typed λ-calculus we have to use dependent inductive types and Kleisli structures instead of triples. Given a set of types Ty, the base category $\mathbf{C}$ is the category of Ty-indexed sets, whose objects are families of sets indexed by types ($F \in \mathrm{Ty} \to \mathbf{Set}$) and the morphisms are type-indexed families of functions $f \in \Pi_{\sigma\in\mathrm{Ty}}F(\sigma) \to G(\sigma)$.

The index set I is given by the inductively defined set of contexts Con and the families involved are $\mathrm{Var}(\Gamma, \sigma)$ – the set of variables of type σ in context Γ – and $\mathrm{Lam}(\Gamma, \sigma)$ – the set of terms of type σ in context Γ. $\mathrm{Var}(\Gamma)$ and $\mathrm{Lam}(\Gamma)$ are objects in our base category for any $\Gamma \in \mathrm{Con}$.

We shall present the types involved by giving the constructors. The set of types Ty and contexts Con are given by the following homogeneous definitions: $\mathrm{Ty} \in \mathbf{Set}$, $\mathrm{o} \in \mathrm{Ty}$, $- \Rightarrow - \in \mathrm{Ty} \to \mathrm{Ty} \to \mathrm{Ty}$, $\mathrm{Con} \in \mathbf{Set}$, $\mathrm{empty} \in \mathrm{Con}$, $\mathrm{cons} \in \mathrm{Ty} \to \mathrm{Con} \to \mathrm{Con}$. Here cons corresponds to $-_\perp$ in the untyped case. Var is given by a dependently typed inductive definition:

$$\begin{aligned}
&\mathrm{Var} \in \mathrm{Con} \to \mathrm{Ty} \to \mathbf{Set}\\
&\mathrm{old} \in \Pi_{\Gamma \in \mathrm{Con}} \Pi \tau \in \mathrm{Ty}.\Pi_{\sigma \in \mathrm{Ty}} \mathrm{Var}(\Gamma, \sigma) \to \mathrm{Var}(\mathrm{cons}(\tau, \Gamma), \sigma)\\
&\mathrm{new} \in \Pi \Gamma \in \mathrm{Con}.\Pi \sigma \in \mathrm{Ty}.\mathrm{Var}(\sigma, \mathrm{cons}(\sigma, \Gamma))
\end{aligned}$$

Similarly, Lam is given by a dependent inductive type:

$$\begin{aligned}
&\mathrm{Lam} \in \mathrm{Con} \to \mathrm{Ty} \to \mathbf{Set}\\
&\mathrm{var} \in \Pi_{\Gamma \in \mathrm{Con}, \sigma \in \mathrm{Ty}} \mathrm{Var}(\Gamma, \sigma) \to \mathrm{Lam}(\Gamma, \sigma)\\
&\mathrm{app} \in \Pi_{\Gamma \in \mathrm{Con}, \sigma, \tau \in \mathrm{Ty}} \mathrm{Lam}(\Gamma, \sigma \Rightarrow \tau) \to \mathrm{Lam}(\Gamma, \sigma) \to \mathrm{Lam}(\Gamma, \tau)\\
&\mathrm{abst} \in \Pi_{\Gamma \in \mathrm{Con}, \sigma, \tau \in \mathrm{Ty}} \mathrm{Lam}(\mathrm{cons}(\tau, \Gamma), \sigma) \to \mathrm{Lam}(\Gamma, \sigma \Rightarrow \tau)
\end{aligned}$$

As in the untyped case var is the unit η of our Kleisli structure. We now define bind and lift by simultaneous recursion:

$$\begin{aligned}
\mathrm{bind} \in \Pi_{\Gamma, \Delta \in \mathrm{Con}}(\Pi_{\sigma \in \mathrm{Ty}} \mathrm{Var}(\Gamma, \sigma) \to \mathrm{Lam}(\Delta, \sigma)) \to\\
\Pi_{\sigma \in \mathrm{Ty}} \mathrm{Lam}(\Gamma, \sigma) \to \mathrm{Lam}(\Delta, \sigma)\\
\mathrm{lift} \in \Pi_{\Gamma, \Delta \in \mathrm{Con}} \Pi \tau \in \mathrm{Ty}(\Pi_{\sigma \in \mathrm{Ty}} \mathrm{Var}(\Gamma, \sigma) \to \mathrm{Lam}(\Delta, \sigma)) \to\\
\Pi_{\sigma \in \mathrm{Ty}} \mathrm{Var}(\mathrm{cons}(\tau, \Gamma), \sigma) \to \mathrm{Lam}(\mathrm{cons}(\tau, \Delta), \sigma)\\
\mathrm{lift}(\sigma, f, \mathrm{new}(\Gamma, \sigma)) = \mathrm{var}(\mathrm{new}(\Delta, \sigma))\\
\mathrm{lift}(\sigma, f, \mathrm{old}(\sigma, x)) = \mathrm{bind}(\mathrm{var} \circ \mathrm{old}(\sigma), f(x))\\
\mathrm{bind}(f, \mathrm{var}(x)) = f(x)\\
\mathrm{bind}(f, \mathrm{app}(t, u)) = \mathrm{app}(\mathrm{bind}(f, t), \mathrm{bind}(g, t))\\
\mathrm{bind}(f, \mathrm{abst}(t)) = \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(\sigma, f), t))
\end{aligned}$$

The termination argument is the same as for the untyped case, see Section 4.1.

5.4 Lam is a Kleisli Structure

The verification of this fact has the same structure as the previous proof but with different types. Let us state the result precisely:

Theorem 1. Lam *gives rise to a Kleisli structure where*

- $\mathbf{C}$ *is the category of* Ty*-indexed families.*
- $I = \mathrm{Con}$
- $F = \mathrm{Var} \in \mathrm{Con} \to |\mathbf{C}|$
- $G = \mathrm{Lam} \in \mathrm{Con} \to |\mathbf{C}|$
- $\eta_\Gamma = \mathrm{var}_\Gamma \in \mathbf{C}(\mathrm{Var}(\Gamma), \mathrm{Lam}(\Gamma))$
- $\mathrm{bind}_{\Gamma,\Delta} \in \mathbf{C}(\mathrm{Var}(\Gamma), \mathrm{Lam}(\Delta)) \to \mathbf{C}(\mathrm{Lam}(\Gamma), \mathrm{Lam}(\Delta))$

Proof. See the proofs of Corollary 1.

6 Conclusions and Open Problems

We have discussed a uniform representation of untyped and typed λ-terms based on Kleisli triples in type theory using heterogeneous (generalized) datatypes. All this can be easily implemented in Haskell and in a special version of SML and formally verified in LEGO. The recursive construction of the Kleisli-triple turned out to be much simpler than the structural inductive one which emphasizes our point of view that recursive proofs are often easier and should be supported by modern type theoretical systems. It is future work to look for a generalization to terms of dependently typed λ-calculi, thus suggesting a new approach for the project of *Type Theory in Type Theory* (cf. [MP93]). A problem which needs to be tackled in this context is that the type of the substitution function in a dependently typed context may depend on its own graph.

Once having finished the examination of the Lam-monad and turning attention to other examples of heterogeneous datatypes many interesting questions arise that deserve further investigation. There exist practically interesting examples that need a stronger notion of inductively defined *functors*, not just operators. Moreover, can one find a useful characterisation of "being Kleisli" for inductive families? A challenging open question is whether inductively defined operators are proof-theoretically conservative with respect to standard inductive ones, i.e. can one define more functions on natural numbers using inductive operators?

Acknowledgements

Thanks to the anonymous referees and to Neil Ghani for comments on a draft version of the paper.

References

[Alt93a] T. Altenkirch. *Constructions, Inductive Types and Strong Normalization.* PhD thesis, University of Edinburgh, November 1993.

[Alt93b] T. Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In J.F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, LNCS 664, pages 13–28, 1993.

[Alt98] T. Altenkirch. Logical relations and inductive/coinductive types. Proceedings of CSL 98, LNCS 1584, pages 343–354,1998.

[Alt99] T. Altenkirch. Extensional equality in intensional type theory. In *Proceedings of LICS 99*, pages 412–420, 1999.

[AP93] M. Abadi and G. Plotkin. A logic for parametric polymorphism. In *Typed Lambda Calculi and Applications–TLCA '93*, pages 361–375, 1993.

[BH94] F. Bellegarde and J. Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 23(2-3), 1994.

[BM98] R. Bird and L. Meertens. Nested datatypes. In J. Jeuring, editor, *Mathematics of Program Construction*, number 1422 in LNCS, pages 52–67. Springer Verlag, 1998.

[BP99] R. Bird and R. Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9:77–91, 1999.

[Dyb94] P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

[EL99] M. Emms and H. Leiss. Extending the type checker of Standard ML by polymorphic recursion. *TCS*, 212(1), 1999.

[FPT99] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proceedings of LICS 99*, pages 193–204,1999.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[HJW+92] P. Hudak et al. Report on the programming language Haskell: a non-strict, purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[HMM86] R. Harper, D. MacQueen, and R. Milner. Standard ML. Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, 1986.

[Hof97] M. Hofmann. *Semantics of Logics of Computation*, chapter Syntax and Semantics of Dependent Types. Cambridge University Press, 1997.

[Hof99] M. Hofmann. Semantical analysis in higher order abstract syntax. In *Proceedings of LICS 99*, pages 204–213, 1999.

[Hue92] G. Huet. Constructive computation theory PART I. Notes de cours, 1992.

[LP92] Z. Luo and R. Pollack. The LEGO proof development system: A user's manual. LFCS report ECS-LFCS-92-211, University of Edinburgh, 1992.

[Man76] E. Manes. *Algebraic Theories*, volume 26 of *Graduate Texts in Mathematics*. Springer Verlag, 1976.

[Mar84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[ML71] S. Mac Lane. *Categories for the Working Mathematician*. Springer Verlag, 1971.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[MP93] J. McKinna and R. Pollack. Pure type systems formalized. In *Proceedings TLCA'93*, LNCS 664, pages 289–305, 1993.

[RA99] B. Reus and T. Altenkirch. The implementation of the λ-monad in LEGO. Available on the WWW at: `http://www.informatik.uni-muenchen.de/~reus/drafts/lambda.html`, March 1999.

[Wad89] P. Wadler. Theorems for free! In *4'th Symposium on Functional Programming Languages and Computer Architecture, ACM, London*, September 1989.

A P-Time Completeness Proof for Light Logics

Luca Roversi

Institut de Mathématiques de Luminy
UPR 9016 – 163, avenue de Luminy – Case 907
13288 Marseille Cedex 9 – France
rover@iml.univ-mrs.fr

Abstract. We explain why the original proofs of P-Time completeness for Light Affine Logic and Light Linear Logic can not work, and we fully develop a working one.
Keywords: Light Affine/Linear Logics, P-Time completeness, Programming with feasible functions.

1 Introduction

The aim of this work is twofold. On one side, it develops in full details the proof that Light Affine Logic (LAL) [1] is P-Time complete. This means showing that a P-Time Turing machine can be encoded as a derivation of LAL, which is a smart simplification of Light Linear Logic (LLL) [3]. The simplification consists of allowing the unconstrained use of weakening. This does not affect the complexity of the cut elimination for LAL. It remains bound by a polynomial in the dimension of the derivation. On the other side, this work introduces a very compact paradigmatic functional language Λ_{LA} for programming with the derivations of LAL, which represent feasible functions.

The development of the proof of P-Time completeness of LAL is not merely a programming exercise with an exotic functional notation. Many readers might get to this conclusion just recalling that the P-Time completeness of LAL was claimed to hold in [1], where the hints for proving it say to follow what Girard does in [3]. However, following Girard, one gets stuck. The problem is that the derivation of LLL, encoding the transition function of the Turing machine being represented, does not correspond to an iterable program, which we call *t_fun* for short. Let us see why this happens. Firstly, recall that an *iteration principle* can be derived in LLL: it requires that the iterated function has a type $\tau \multimap \tau$ for some "light linear" type τ. Secondly, the iteration principle serves for encoding the Turing machine: *t_fun* is iterated on the starting configuration at most as many times as the bound given by the polynomial. Assume now **config** be the name of the type for the representation in LAL of the configurations (tape, state, head position) of a given Turing machine. Following [3], *t_fun* can not be written with a type different from **config** $\multimap$ §**config**, where § is one of the two modalities "!" and "§" of LLL. So, *t_fun* can not be argument of the iteration principle, and nothing works, neither in LLL, nor in LAL.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 469–483, 1999.

The solution to the problem is to change the representation of the configurations. To see how, we use an example. Suppose we want to represent a configuration $\mathcal{C}$ of a Turing machine such that the tape is $10 \star 10$, the state is s_i, and the head is on the cell containing $\star$. The derivation used in [3] to encode $\mathcal{C}$ corresponds to the following tuple of terms of System F [4]:

$$[\lambda ozsx.o(zx), \lambda ozsx.s(z(ox)), state_i] \ , \tag{1.1}$$

which becomes:

$$\lambda OZS.\S(\lambda x.\bar{!}O(\bar{!}Z\ x)) \otimes \lambda OZS.\S(\lambda x.\bar{!}S(\bar{!}Z(\bar{!}O\ x))) \otimes state_i \tag{1.2}$$

in the language Λ_{LA} we shall introduce. The term $\lambda ozsx.o(zx)$ encodes the tape to the left of the head in reversed order, $\lambda ozsx.s(z(ox))$ is the part of tape from the head position to its right, and $state_i$ is some encoding of the state s_i. On the contrary, our encoding of $\mathcal{C}$ corresponds to the term:

$$\lambda oo'zz'ss'xx'.[o(zx), s'(z'(o'x')), state_i] \tag{1.3}$$

of System F, which becomes:

$$\lambda OO'ZZ'SS'.\S(\lambda xx'.\bar{!}O(\bar{!}Z\ x) \otimes \bar{!}S'(\bar{!}Z'(\bar{!}O'\ x')) \otimes state_i) \tag{1.4}$$

in Λ_{LA}. The difference between the two choices is evident. The old one separates the components of the tape in two different λ-abstractions, while the new one keeps them merged into a single λ-body. We are now in the position to have a good intuition about why (1.2) can not work. Firstly, recall that in LAL there is the §-box constructor "§". Secondly, recall that LAL does not allow any box opening. We can only merge the borders of two boxes, but we can never drop one border completely: this is the key point for proving the complexity bound! Any encoding *t_fun* of the transition function working on (1.2) needs to access the components of (1.2). This can be accomplished by a *t_fun* with a §-box which border must be merged with all those in (1.2). The lack of dereliction does not allow to get rid of such a §-box in *t_fun*: it gets recorded in the co-domain of the type of *t_fun*, which can only be $\mathbf{config} \multimap \S\mathbf{config}$. This problem disappears if *t_fun* is written for manipulating (1.4). Indeed, the §-box of *t_fun* used to access the components of (1.4) is the same as the one needed to build the new configuration. Encoding the configurations as in (1.4) we get:

Theorem. Any P-Time Turing machine with a polynomial $p(x)$ of maximal non null degree ϑ, bounding its computational complexity, can be encoded in a term M of Λ_{LA}, such that M has the Linear Affine Logic formula $\mathbf{tape} \multimap \S^{\vartheta+5}\mathbf{tape}$ as its type, for some suitable type **tape**.

Contents: Section 2 introduces the language Λ_{LA}. Section 3 recalls Intuitionistic Light Affine Logic and decorates its derivations with Λ_{LA}. Section 4 defines the encoding of the P-Time Turing machine in the typable fragment of Λ_{LA}, mainly focusing on the encoding of the transition function. The representation in Λ_{LA} of all the remaining details are demanded to Appendix A, and B. Section 5 recalls the justification about writing this work. Section 6 concludes the work.

Acknowledgments: Many thanks to Yves Lafont for his fundamental help about how to terminate this work. Thanks also to the anonymous referees whose suggestions helped me to eliminate some inaccuracies. The work has been partially supported by the TMR-Marie Curie Grant, contract n. ERBFMBICT972805.

2 The Functional Language

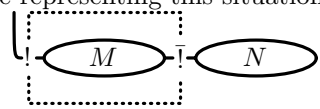
Syntax. Let $c, h, j, o, s, x, y, w, z$ range over the set of *linear* names $\mathtt{T_{vars}}$, and J, O, X, Y, Z range over the set of *exponential* identifiers $!\mathtt{T_{vars}}$. Let also χ be ranging over $\mathtt{T_{vars}} \cup !\mathtt{T_{vars}}$. The set of *patterns* is generated by:

$$R ::= \mathtt{T_{vars}} \mid\ !\mathtt{T_{vars}} \mid\ R \otimes R$$

and is ranged over by $\wp$. The set Λ of the functional terms is given by:

$$M, N, P, Q ::= \mathtt{T_{vars}} \cup !\mathtt{T_{vars}} \mid\ \lambda\wp.M \mid\ MN \mid\ P \otimes Q \mid\ !M \mid\ \bar{!}M \mid\ \S M \mid\ \bar{\S} M$$

For any pattern $\chi_1 \otimes \ldots \otimes \chi_n$, the set $\mathtt{FV}(\chi_1 \otimes \ldots \otimes \chi_n)$ of its free variables is $\{\chi_1, \ldots, \chi_n\}$. As usual, λ binds the variables of M so that $\mathtt{FV}(\lambda\wp.M)$ is $\mathtt{FV}(M) \setminus \mathtt{FV}(\wp)$. The free variable sets of all the remaining terms are obvious as the constructors $\otimes, !, \S, \bar{!}$, and $\bar{\S}$ do not bind variables. Both ! and § build !-boxes and §-boxes, respectively, being M the *body*. The term constructor $\bar{!}$ can mark one of the entry points of both !-boxes and §-boxes, while $\bar{\S}$ can mark only those of §-boxes. An idea about what we mean by "entry point" can be given pictorially. Assume $!M$ be a !-box, having a single entry point with a closed N, plugged into it. The figure representing this situation is:



where the dashed line stands for the ideal box containing M, ! is its exit, $\bar{!}$ is its entry point, and N has its root plugged into the entry point of $!M$.

The elements of Λ are considered up to the usual α-equivalence. It allows the renaming of the bound variables of a term M. For example, $!(\lambda x.(\bar{!}y)\ x)$ and $!(\lambda X.(\bar{!}y)\ X)$ are each other α-equivalent.

The substitution of M for χ in N is denoted by $N\{^M\!\downarrow_\chi\}$. It is the obvious extension to Λ of the capture-free substitution of terms for variables, defined for the λ-Calculus [2] . For example, $X\{^x\!\downarrow_X\}$ yields x.

It can be observed that in the definition of the substitution the existence of two sets of variables is overlooked. Both the dynamics on the terms of Λ (introduced below), and the way we give a type to them with the formulas of Intuitionistic Light Linear Logic (introduced in Section 3) will establish a substitution policy about how correctly substitute terms for variables, as follows: the substitution $N\{^M\!\downarrow_\chi\}$ will become equivalent to a *partial* substitution that behaves as usual *only* in one of the two, mutually exclusive, cases:

- if χ is an exponential identifier, then M must be either a !-box, or an exponential identifier;

– if χ is a linear identifier, then M can be any term.

Otherwise, the substitution is undefined. For example, $X\{^{x}\!\downarrow_{X}\}$ will not work.

The substitutions can be generalized to $\{^{M_1}\!\downarrow_{\chi_1} \cdots {}^{M_n}\!\downarrow_{\chi_n}\}$, meaning the simultaneous replacement of M_i for χ_i, for every $1 \leq i \leq n$.

The notation $M[N_1, \ldots, N_n]$ means that M *may* contain $N_1, \ldots, N_n$ as its sub-terms. In particular, $!M[\bar{!}N]$ means that $!M$ *may* contain $\bar{!}N$ as its sub-term. Notice that, under the definition of Λ, the notation $!M[\bar{!}N]$ is ambiguous: if we let M be $(x\ \bar{!}X\ \bar{!}Y)$, then $!M$ can be written both as $!M[\bar{!}X]$ and $!M[\bar{!}Y]$.

We shall use $\equiv$ as syntactic coincidence.

Finally, a relation *unpack* on $\otimes$-tuples of terms is defined:

$$\begin{array}{l}
\mathrm{unpack}(M_1 \otimes \ldots \otimes M_m, P_1 \otimes \ldots \otimes P_m\{^{N_{j_1}}\!\downarrow_{X_{j_1}} \cdots {}^{N_{j_n}}\!\downarrow_{X_{j_n}}\}) \textbf{ iff} \\
\{j_1, \ldots, j_n\} \subseteq \{1, \ldots, m\} \text{ where } 1 \leq k \neq i \leq n \text{ implies } j_k \neq j_i \ , \\
\{k_1, \ldots, k_{m-n}\} \text{ is } \{1, \ldots, m\} \setminus \{j_1, \ldots, j_n\} \ , \\
M_{j_i} \equiv !Q_{j_i}[\bar{!}N_{j_i}] \text{ with } N_{j_i} \not\equiv Z \text{ for any } Z \ , \\
\text{if } M_{k_l} \text{ is a !-box } !Q_{k_l}[\bar{!}N_{k_l}], \text{ then } N_{k_l} \equiv Z \text{ for some } Z \ , \\
P_{j_i} \equiv !Q_{j_i}[\bar{!}X_{j_i}],\ P_{k_l} \equiv M_{k_l} \ .
\end{array}$$

Dynamics. The rewriting system $\rightsquigarrow$ on Λ is the contextual closure of the union of two rewriting relations $\rhd_\beta$ and $\rhd_d$ on $\Lambda \times \Lambda$. The first is:

$$\begin{array}{l}
(\lambda \chi_1 \otimes \ldots \otimes \chi_m . M) M_1 \otimes \ldots \otimes M_m \rhd_\beta \\
\quad (\lambda X_{j_1} \otimes \ldots \otimes X_{j_n} . M\{\cdots {}^{P_{j_i}}\!\downarrow_{\chi_{j_i}} \cdots {}^{P_{k_l}}\!\downarrow_{\chi_{k_l}} \cdots\}) N_{j_1} \otimes \ldots \otimes N_{j_n} \\
\quad \textbf{if } \mathrm{unpack}(M_1 \otimes \ldots \otimes M_m, P_1 \otimes \ldots \otimes P_m\{^{N_{j_1}}\!\downarrow_{X_{j_1}} \cdots {}^{N_{j_n}}\!\downarrow_{X_{j_n}}\}) \ .
\end{array}$$

The relation unpack formalizes the idea that an exponential variable can duplicate both exponential variables and !-boxes, but nothing else. On the other hand, every term can replace a linear variable. An example of $\rhd_\beta$-reduction is:

$$(\lambda X \otimes x . M)\ (!(\lambda y . (\bar{!}(wz))y) \otimes (w'z')) \rhd_\beta (\lambda Y . M\{^{!(\lambda y . (\bar{!}Y)y)}\!\downarrow_{X}\ {}^{w'z'}\!\downarrow_{x}\})(wz) \ .$$

No problem arises replacing x by $w'z'$. The part of $!(\lambda y.(\bar{!}(wz))y)$ that can be duplicated by X, possibly occurring more than once in M, is only $!(\lambda y.(\bar{!}Y)y)$. So wz is kept as a single argument after the reduction.

The second rewriting relation *merges* the borders of two boxes:

$$\bar{\Diamond}\Diamond M \ \rhd_d \ M \text{ with } \Diamond \in \{!, \S\} \ .$$

The α-equivalence must be used to avoid variable clashes when rewriting terms, so that linear (respectively exponential) variables rename linear (respectively exponential) variables.

The reflexive, and transitive closure of $\rightsquigarrow$ on Λ is $\rightsquigarrow^*$.

Finally, the pair $\langle \Lambda, \rightsquigarrow \rangle$ is the functional language Λ_{LA}.

3 Intuitionistic Light Affine Logic

This section recalls the sequent calculus of Intuitionistic Light Affine Logic, and decorates its inference rules by the terms of Λ_{LA}. For a correct decoration we need some adjustments of the logical formulas that the sequent calculus can derive, with respect to [1].

Logical Formulas. Let α, β, γ range over the set of *linear* identifiers $\mathtt{F}_{\mathtt{vars}}$, and let δ range over the set of *exponential* identifiers $!\mathtt{F}_{\mathtt{vars}}$. Let also ς range over $\mathtt{F}_{\mathtt{vars}} \cup !\mathtt{F}_{\mathtt{vars}}$. The language of the formulas of Intuitionistic Light Linear Logic is generated by:

$$\begin{aligned} \rho, \sigma, \tau ::= {} & L \mid E \\ L ::= {} & \mathtt{F}_{\mathtt{vars}} \mid \rho \multimap \sigma \mid \sigma \otimes \tau \mid \S\tau \mid \forall\varsigma.L \\ E ::= {} & !\mathtt{F}_{\mathtt{vars}} \mid !\rho \mid \forall\varsigma.E \ . \end{aligned}$$

As usual, $\forall$ is a binder: the free variables of $\forall\varsigma_1 \ldots \varsigma_n.\tau$ are $\mathtt{FV}(\tau) \setminus \{\varsigma_1 \ldots \varsigma_n\}$ with $\mathtt{FV}(\tau)$ having the obvious inductive definition. The formulas are taken up to α-equivalence. The *linear* formulas are those generated by the grammar with L as its start symbol. The *exponentials* start from E.

Any *linear* formula τ not having $\otimes$ as its principal operator is thought of as a *degenerate* tensor, namely a tuple with a single element.

A *basic set of assumptions* is a set of pairs $\{\chi_1 : \sigma_1, \ldots, \chi_n : \sigma_n\}$ such that:

1. Every χ_i is an *exponential* (respectively *linear*) term variable if, and only if, σ_i is an *exponential* (respectively *linear*) formula;
2. $\{\chi_1 : \sigma_1, \ldots, \chi_n : \sigma_n\}$ is a function with finite domain $\{\chi_1, \ldots, \chi_n\}$. Namely, if $i \neq j$ then $\chi_i \neq \chi_j$.

An *extended set of assumptions* is a basic set containing also pairs $\wp : \sigma$, and satisfying some constraints. Assume $\wp$ be $\chi_1 \otimes \ldots \otimes \chi_m$. Then:

1. σ must be $\sigma_1 \otimes \ldots \otimes \sigma_p$, with $p \geq m$;
2. $\{\chi_1 : \tau_1, \ldots, \chi_m : \tau_m\}$ is a basic set of assumptions, where every τ_i is a, possibly degenerate, tensor of formulas in $\{\sigma_1, \ldots, \sigma_p\}$.

For example, $\{X : \delta, y : \beta\}$ is a legal extended set. $\{X \otimes x : \delta, y : \beta\}$ is not.

From now on, by "assumptions" we mean "extended set of assumptions". Meta-variables for ranging over the assumptions are Γ and Δ.

The *substitutions* on formulas replace *linear* (respectively *exponential*) formulas for *linear* (respectively *exponential*) variables. The simultaneous substitution of $\tau_1 \ldots \tau_n$ for $\varsigma_1 \ldots \varsigma_n$ is denoted by $\{^{\tau_1}\!\downarrow_{\varsigma_1} \cdots {}^{\tau_n}\!\downarrow_{\varsigma_n}\}$.

Logical Rules. We recall the sequent calculus for Intuitionistic Light Affine Logic [1] by decorating it with the terms of Λ_{LA}. The judgments have form:

$\Gamma \vdash M : \tau$, where Γ is a set of assumptions, M is a term of Λ_{LA}, and τ is a formula. The decorated system is:

$$(\mathrm{Ax})\frac{}{\chi : \tau \vdash \chi : \tau} \qquad (\mathrm{Cut})\frac{\Gamma \vdash M : \sigma \quad \Delta, \chi : \sigma \vdash N : \tau}{\Gamma, \Delta \vdash N\{^{M}\!\downarrow_{\chi}\} : \tau}$$

$$(\mathrm{W})\frac{\Gamma \vdash M : \tau}{\Gamma, \chi : \sigma \vdash M : \tau} \qquad (\mathrm{C})\frac{\Gamma, X : !\sigma, Y : !\sigma \vdash M : \tau}{\Gamma, Z : !\sigma \vdash M\{^{Z}\!\downarrow_{X}\ ^{Z}\!\downarrow_{Y}\} : \tau}$$

$$(\multimap_{\mathrm{l}})\frac{\Gamma \vdash M : \sigma \qquad \Delta, \chi : \tau \vdash N : \rho}{\Gamma, \Delta, x : \sigma \multimap \tau \vdash N\{^{xM}\!\downarrow_{\chi}\} : \rho} \qquad (\multimap_{\mathrm{r}})\frac{\Gamma, \wp : \tau_1 \otimes \ldots \otimes \tau_n \vdash M : \tau}{\Gamma \vdash \lambda\wp.M : \tau_1 \otimes \ldots \otimes \tau_n \multimap \tau}$$

$$(\otimes_{\mathrm{l}})\frac{\Gamma, \chi_1 : \tau_1, \chi_2 : \tau_2 \vdash M : \tau}{\Gamma, \chi_1 \otimes \chi_2 : \tau_1 \otimes \tau_2 \vdash M : \tau} \qquad (\otimes_{\mathrm{r}})\frac{\Gamma \vdash M : \tau \quad \Delta \vdash N : \sigma}{\Gamma, \Delta \vdash M \otimes N : \tau \otimes \sigma}$$

$$(!)\frac{\ldots \chi_i : \sigma_i \ldots \vdash M : \tau \qquad 0 \leq i \leq n \leq 1}{\ldots X_i : !\sigma_i \ldots \vdash !M\{\ldots\ ^{\bar{!}X_i}\!\downarrow_{\chi_i} \ldots\} : !\tau}$$

$$(\S)\frac{\ldots \chi_i : \tau_i \ldots \chi'_j : \sigma_j \ldots \vdash M : \tau \qquad 0 \leq i \leq m \quad 0 \leq j \leq n}{\ldots X_i : !\tau_i \ldots x_j : \S\sigma_j \ldots \vdash \S M\{\ldots\ ^{\bar{!}X_i}\!\downarrow_{\chi_i} \ldots\ ^{\bar{\S}x_j}\!\downarrow_{\chi'_j} \ldots\} : \S\tau}$$

$$(\forall_{\mathrm{l}})\frac{\Gamma, \chi : \{^{\tau}\!\downarrow_{\varsigma}\}\sigma \vdash M : \rho}{\Gamma, \chi : \forall\varsigma.\sigma \vdash M : \rho} \qquad (\forall_{\mathrm{r}})\frac{\Gamma \vdash M : \sigma \qquad \varsigma \notin \mathtt{FV}(\Gamma)}{\Gamma \vdash M : \forall\varsigma.\sigma}$$

Observe that (!)-rule can have at most one assumption. So the notation $!M[\bar{!}N]$ can not be anymore ambiguous.

Observe that Λ_{LA} gives a very parsimonious representation of the derivations. The contraction is left implicit, allowing multiple occurrences of the same exponential variable. The pattern matching avoids the use of any *let*-like binder that would require some commuting conversions in $\rightsquigarrow$. The representation of the boxes is much more compact than that used in [1, 5, 6], and this prevents the need of a lot of commuting conversions. Somebody might object about the (relative) complexity of the $\rhd_\beta$-reduction. It is the side effect of the lack of explicit encoding for the (C)-rule. Assuming we have it, we could redefine $\rhd_\beta$ simply as:

$$(\lambda\chi_1 \otimes \ldots \otimes \chi_m.M)M_1 \otimes \ldots \otimes M_m \rhd_\beta M\{^{M_1}\!\downarrow_{\chi_1} \ldots\ ^{M_m}\!\downarrow_{\chi_m}\}\ .$$

In this case, the duplication of every M_i with form $!P_i[\bar{!}Q_i]$ as many times as the occurrences of every χ_i should be filtered by the explicit contraction. The aim: forbidding the duplication of any Q_i different from both an exponential variable and a !-box. Moreover, the explicit term for contraction would induce commuting conversions. Hence, we would pay in term of more reductions, and

more constructs. So, the detailed encoding of the P-Time Turing machines in Appendix A, and B would get (much) more unreadable.

The language Λ_{LA} does not encode explicitly the II order quantification. This only means that it has less reduction steps than Asperti's original functional notation to speak about the derivations of LAL [1]. So, the computational complexity is preserved.

Splitting the logical formulas into linear and exponential ones is a consequence of splitting the set of assumptions for the derivations. The logic, however, is unchanged: as soon as the term decorations are forgotten, any difference on the logical formulas can be forgotten as well.

4 Encoding P-Time Turing Machines

The encoding morally divides into two parts that we call *quantitative* and *qualitative*[1]. The quantitative part is relative to the representation of the polynomial which bounds the computational complexity. The qualitative one is about encoding the transition function of the machine being encoded.

The quantitative part gets the representation of the initial tape as input. Its main tasks are: calculating the integer $\mathcal{B}$ which bounds the number of computational steps, and using $\mathcal{B}$ for iterating the representation *t_fun* of the transition function. The qualitative part takes a *configuration* as input, namely a copy of the representation of a tape and a state. The qualitative part implements the transition function *t_fun* which shifts the head of the machine on the tape, according to the actual state and to the last character read.

We assume to encode a machine $\langle \Sigma, \mathcal{S}, \mathcal{F} \rangle$ where Σ is the *input* tape alphabet $\{0,1\}$, $\mathcal{S}$ is the set of states with cardinality m, $\mathcal{F}$ is the transition function $(\Sigma \cup \{\star\}) \times \mathcal{S} \longrightarrow \Sigma \times \mathcal{S} \times \{L, R\}$. The symbol $\star$ stands for the "blank" cell, while L and R are the directions of the head moves. The head is supposed to write a symbol into the last read cell of the tape, before moving. Among the states in $\mathcal{S}$, we distinguish the initial one S_0. The alphabet $\Sigma \cup \{\star\}$ is ranged over by ζ, the set of states by S, and $\{L, R\}$ by μ.

Configurations. A configuration is determined by a tape, a position of the head on it, and a state. The representation we choose is:

$$\lambda O O' Z Z' J J'.\S(\lambda x x'.(\chi_1(\dots(\chi_p\ x)\dots) \otimes \chi'_1(\dots(\chi'_q\ x')\dots)) \otimes \mathit{state}_i) \ ,$$

where $\chi_i \in \{\bar{!}O, \bar{!}Z, \bar{!}J\}$ with $1 \leq i \leq p$, $\chi'_j \in \{\bar{!}O', \bar{!}Z', \bar{!}J'\}$ with $1 \leq j \leq q$, being $p, q \geq 0$. The type **config** of any term *config* like the one here above is:

$$\begin{aligned}\mathbf{config} \stackrel{\mathrm{def}}{\equiv} \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \\ !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \\ !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \\ \S(\alpha \multimap \alpha \multimap (\alpha \otimes \alpha \otimes \mathbf{state})) \ ,\end{aligned}$$

[1] Lafont suggested this terminology.

where $\mathbf{state} \stackrel{\text{def}}{\equiv} \forall\alpha\beta.(\overbrace{(\alpha \multimap \beta)\otimes\ldots\otimes(\alpha \multimap \beta)}^{m}\otimes\alpha) \multimap \beta$ is the type of the terms $state_i$, each one representing an element of $\mathcal{S}$, which, recall, has cardinality m.

Example 1. The configuration where the head is on $\star$ of the tape:

$$10 \star 10 \ , \tag{4.1}$$

and the actual state is S_i, is encoded with:

$$\lambda OO'ZZ'JJ'.\S((\lambda xx'.\bar{!}Z(\bar{!}O\ x)\otimes\bar{!}J'(\bar{!}O'(\bar{!}Z'\ x')))\otimes state_i) \ . \tag{4.2}$$

The leftmost component of the tensor in the body of the λ-abstraction is the part of the tape to the left of the head, in reversed order. The cell read by the head, and the part of the tape to its right is the central component of the tensor.

States. The term $state_i$ is:

$$\lambda x_0\otimes\ldots\otimes x_{m-1}\otimes v.x_i\ v \qquad (0 \leq i \leq m-1) \ .$$

Every $state_i$ is designed to extract a row from an array. The parameter x_i stands for the row, realized by a *closed* term. The parameter v stands for the variables that the rows of the array would share additively, if they were not closed terms. Namely, $state_i$ is the i^{th} projection for the representation of an additive tuple with Intuitionistic Light Affine Logic. We can summarize as follows the behavior of $state_i$ on a tuple with two elements. Assume you want to encode a pair containing two typable terms M and N of Λ_{LA}, of which only one between them will be used in the computation. Suppose also $x_1,\ldots,x_n$ be *all* the *linear* free variables *common* to both M and N. Then $M\otimes N$ is not a legal term. It can not be typed because any x_j here above occurs twice in it, every x_j requiring an exponential type. This contrasts with the effective use of x_j we are going to do: since we assume to use *either* M, *or* N, every x_j is eventually used linearly. Like in [1], the pair is represented as the triple:

$$(\lambda x_1\otimes\ldots\otimes x_n.M)\otimes(\lambda x_1\otimes\ldots\otimes x_n.N)\otimes(x_1\otimes\ldots\otimes x_n) \ .$$

The leftmost component M is extracted by means of a projection that applies $\lambda x_1\otimes\ldots\otimes x_n.M$ to $x_1\otimes\ldots\otimes x_n$. The rightmost component N is obtained analogously, by a projection that applies $\lambda x_1\otimes\ldots\otimes x_n.N$ to $x_1\otimes\ldots\otimes x_n$. Both every $state_i$, and the array, to which $state_i$ is applied, generalize the projections, and the pair of the example here above.

Starting Configurations. Any *starting configuration* $config_0$ has form:

$$\lambda OO'ZZ'JJ'.\S((\lambda xx'.x\otimes\chi'_1(\ldots(\chi'_q\ x')\ldots))\otimes state_0) \ ,$$

where every χ'_j ranges over $\{\bar{!}O',\bar{!}Z'\}$. Namely, the tape has only input characters on it, and the head is on its leftmost input symbol: the part of the tape to the left of the tape is empty. The term $state_0$ encodes S_0.

4.1 The Quantitative Part

The quantitative part calculates the bound on the length of the computation of the P-Time Turing machine encoded. We only state the main representation theorem. More details are in Appendix A. Assume $p(x)$ be the polynomial $\sum_{i=0}^{\vartheta} a_i x^i$, associated to our Turing machine, with maximal non null degree ϑ. Then, $p(x)$ can be encoded by P in Λ_{LA} with type $\mathbf{int} \multimap \S^{\vartheta+3}\mathbf{int}$, where:

$$\mathbf{int} \stackrel{\mathrm{def}}{\equiv} \forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha) \ .$$

Of course, $p(n) = m$, if, and only if, $P\overline{n}$ evaluates to $\overline{m}$, where, for every $n \geq 0$:

$$\overline{n} \stackrel{\mathrm{def}}{\equiv} \lambda X.\S(\lambda y.\underbrace{\bar{!}X(\ldots(\bar{!}X}_{n}\ y)\ldots)) : \mathbf{int} \ .$$

4.2 The Qualitative Part

The qualitative part implements the transition function of the P-Time Turing machine being encoded.

Some preliminary definitions are worth giving:

$$\begin{aligned}
I &\stackrel{\mathrm{def}}{\equiv} \lambda x.x : \mathbf{i}_\alpha \\
\pi_1 &\stackrel{\mathrm{def}}{\equiv} \lambda x\otimes y.x : \mathbf{bool}_\alpha \\
\pi_2 &\stackrel{\mathrm{def}}{\equiv} \lambda x\otimes y.y : \mathbf{bool}_\alpha \\
\Pi_{111} &\stackrel{\mathrm{def}}{\equiv} \lambda(((x\otimes y)\otimes w)\otimes z)\otimes x'.x\ x' : \mathbf{a_bool}_{\alpha,\beta} \\
\Pi_{112} &\stackrel{\mathrm{def}}{\equiv} \lambda(((x\otimes y)\otimes w)\otimes z)\otimes x'.y\ x' : \mathbf{a_bool}_{\alpha,\beta} \\
\Pi_{12} &\stackrel{\mathrm{def}}{\equiv} \lambda(((x\otimes y)\otimes w)\otimes z)\otimes x'.w\ x' : \mathbf{a_bool}_{\alpha,\beta} \\
\Pi_{2} &\stackrel{\mathrm{def}}{\equiv} \lambda(((x\otimes y)\otimes w)\otimes z)\otimes x'.z\ x' : \mathbf{a_bool}_{\alpha,\beta} \\
\mathbf{i}_\alpha &\stackrel{\mathrm{def}}{\equiv} \alpha \multimap \alpha \\
\mathbf{bool}_\alpha &\stackrel{\mathrm{def}}{\equiv} (\alpha\otimes\alpha) \multimap \alpha \\
\mathbf{a_bool}_{\alpha,\beta} &\stackrel{\mathrm{def}}{\equiv} ((((\alpha \multimap \beta)\otimes(\alpha \multimap \beta))\otimes(\alpha \multimap \beta))\otimes(\alpha \multimap \beta))\otimes\alpha \multimap \beta \ .
\end{aligned}$$

I is the identity and π_1, π_2 the projections on usual booleans, while Π_{111}, Π_{112}, Π_{12}, and Π_2 are projections analogous to the states.

Moreover, let $o\vec{z}j$ abbreviate $o\otimes o'\otimes z\otimes z'\otimes j\otimes j'$, and let $O\vec{Z}J$ stand for $\bar{!}O\otimes\bar{!}O'\otimes\bar{!}Z\otimes\bar{!}Z'\otimes\bar{!}J\otimes\bar{!}J'$. The respective types are: $\vec{\alpha} \stackrel{\mathrm{def}}{\equiv} \mathbf{i}_\alpha\otimes\mathbf{i}_\alpha\otimes\mathbf{i}_\alpha\otimes\mathbf{i}_\alpha\otimes\mathbf{i}_\alpha$, and $!\vec{\alpha} \stackrel{\mathrm{def}}{\equiv} !\mathbf{i}_\alpha\otimes!\mathbf{i}_\alpha\otimes!\mathbf{i}_\alpha\otimes!\mathbf{i}_\alpha\otimes!\mathbf{i}_\alpha\otimes!\mathbf{i}_\alpha$.

We are now ready for introducing the main terms.

The encoding of the transition function is:

$$
\begin{array}{l}
t\_fun \stackrel{\text{def}}{\equiv} \\
\quad \lambda cOO'ZZ'JJ'.\S(\lambda xx'.(comp\ O\vec{Z}J) \\
\qquad\qquad\qquad\qquad (\bar{\S}(c\ !(step\ \Pi_{111}\ \bar{!}O)\ !(step\ \Pi_{111}\ \bar{!}O') \\
\qquad\qquad\qquad\qquad\quad !(step\ \Pi_{112}\ \bar{!}Z)\ !(step\ \Pi_{112}\ \bar{!}Z') \\
\qquad\qquad\qquad\qquad\quad !(step\ \Pi_{12}\ \bar{!}J)\ !(step\ \Pi_{12}\ \bar{!}J') \\
\qquad\qquad\qquad\qquad)(base\ \Pi_2\ x)\ (base\ \Pi_2\ x') \\
\qquad\qquad\qquad\qquad) \\
\qquad\qquad\qquad) \ .
\end{array}
$$

It gets a configuration c for yielding a new one. For example, substituting (4.2) for c, the evaluation of the whole sub-term of t_fun, which is argument of $(comp\ O\vec{Z}J)$, ends up with:

$$((\Pi_{112}\otimes\bar{!}Z)\otimes(\bar{!}O\ x))\otimes((\Pi_{12}\otimes\bar{!}J')\otimes(\bar{!}O'(\bar{!}Z'\ x')))\otimes state_i\ . \qquad (4.3)$$

Namely, (4.2) iterates every *step* from *base*, both defined as:

$$
\begin{array}{l}
step \stackrel{\text{def}}{\equiv} \lambda xy.\lambda(u\otimes v)\otimes z.(x\otimes y)\otimes(v\ z) \\
base \stackrel{\text{def}}{\equiv} \lambda xy.(x\otimes I)\otimes y\ ,
\end{array}
$$

and extracts the topmost symbol of both parts of the tape, together with a corresponding projection. The projection serves for choosing an element in a row of an array. In particular, Π_{111} will always be associated to both $\bar{!}O$ and $\bar{!}O'$, Π_{112} to both $\bar{!}Z$ and $\bar{!}Z'$, and Π_{12} to both $\bar{!}J$ and $\bar{!}J'$. For a better exposition, call "*head pair*" each pair like $(\Pi_{112}\otimes\bar{!}Z)$ in (4.3) here above.

The definition of *comp* is:

$$comp \stackrel{\text{def}}{\equiv} \lambda o\vec{z}j.\lambda(((h_l^l\otimes h_l^r)\otimes t_l)\otimes((h_r^l\otimes h_r^r)\otimes t_r))\otimes s.h_r^l(s(table\ o\vec{z}j))\ h_l^r\ t_l\ t_r\ .$$

Firstly, $(comp\ O\vec{Z}J)$ is a λ-abstraction containing the occurrences of the terms $\bar{!}O, \bar{!}O', \bar{!}Z, \bar{!}Z', \bar{!}J$, and $\bar{!}J'$ among which choosing those that must be used for generating the new configuration. The terms $\bar{!}O, \bar{!}O' \ldots$ feed the transition table:

$$
\begin{array}{l}
table \stackrel{\text{def}}{\equiv} \lambda o\vec{z}j. \qquad (\lambda x.(((Q_{0,1}\otimes Q_{0,2})\otimes Q_{0,3})\otimes Q_{0,4})\otimes x) \qquad \otimes \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad (\lambda x.(((Q_{m-1,1}\otimes Q_{m-1,2})\otimes Q_{m-1,3})\otimes Q_{m-1,4})\otimes x)\otimes o\vec{z}j\ .
\end{array}
$$

As said earlier, it represents an array. In our running example, the parameter s of $(comp\ O\vec{Z}J)$ is $state_i$. Then $s(table\ o\vec{z}j)$ is the i^{th} row of the array. From this row, the left component of the head pair $(\Pi_{112}\otimes\bar{!}Z)$ extract a term which is responsible of moving the head. In this case it would be Q_{i2}. With the head pair

$(\Pi_{111} \otimes \bar{!} O)$, the obtained element would be Q_{i1}, while $(\Pi_{12} \otimes \bar{!} J)$ would extract Q_{i3}. Finally, every Q_{ij} in *table* is one among the two shifting terms here below:

$$\begin{aligned} \mathit{left}_{ij} &\stackrel{\text{def}}{\equiv} \lambda o \vec{z} j.\lambda h_l t_l t_r.(t_l \otimes h_l(\chi^l_{ij}\ t_r)) \otimes \mathit{state}^l_{ij} \\ \mathit{right}_{ij} &\stackrel{\text{def}}{\equiv} \lambda o \vec{z} j.\lambda h_l t_l t_r.(\chi^r_{ij}(h_l\ t_l) \otimes t_r) \otimes \mathit{state}^r_{ij}\ , \end{aligned}$$

where χ^l_{ij} ranges over $\{o', z', j'\}$, and χ^r_{ij} over $\{o, z, j\}$. Every *left* and *right* term describes what to do when moving the head leftward or rightward, respectively. Of course, the form of *table* must be defined to satisfy the obvious link between the encoding and the machine encoded. The link is: $\mathcal{F}\langle\zeta, S\rangle \mapsto \langle\zeta', S', \mu\rangle$ iff the term $h^l_r\ (\mathit{state}\ (\mathit{table}\ o\vec{j}z))$ of *comp* $\rightsquigarrow$-reduces to Q, where: if $\mu \equiv L$, then $Q \equiv \mathit{left}[\chi^l, \mathit{state}']$, if $\mu \equiv R$, then $Q \equiv \mathit{right}[\chi^r, \mathit{state}']$, if $\zeta \equiv 1$, then $h^l_r \equiv \Pi_{111}$, if $\zeta \equiv 0$, then $h^l_r \equiv \Pi_{112}$, if $\zeta \equiv \star$, then $h^l_r \equiv \Pi_{12}$, *state* encodes S, *state*$'$ encodes S', if $\zeta' \equiv 1$, then both $\chi^l \equiv o$ and $\chi^r \equiv o'$, if $\zeta' \equiv 0$, then both $\chi^l \equiv z$ and $\chi^r \equiv z'$, if $\zeta' \equiv \star$, then both $\chi^l \equiv j$ and $\chi^r \equiv j'$.

For those who want to check that the compulsory requirement about *t_fun* is satisfied, namely, that *t_fun* has an *iterable* type $\mathbf{config} \multimap \mathbf{config}$, we give some hints about the intermediate typing: $\mathit{step} : \mathbf{step}_{\alpha,\beta}$, $\mathit{base} : \mathbf{base}_{\alpha,\beta}$, $\mathit{comp} : \mathbf{comp}$, $\mathit{table} : \mathbf{table}_\alpha$ $\mathit{left}_{ij} : \mathbf{shift}_\alpha$, and $\mathit{right}_{ij} : \mathbf{shift}_\alpha$, where:

$$\begin{aligned} \mathbf{step}_{\alpha,\beta} &\stackrel{\text{def}}{\equiv} \mathbf{a\_bool}_{\alpha,\beta} \multimap \mathbf{i}_\alpha \multimap \sigma_{\alpha,\beta} \multimap \sigma_{\alpha,\beta} \\ \mathbf{base}_{\alpha,\beta} &\stackrel{\text{def}}{\equiv} \mathbf{a\_bool}_{\alpha,\beta} \multimap \alpha \multimap \sigma_{\alpha,\beta} \\ \mathbf{comp}_\alpha &\stackrel{\text{def}}{\equiv} \vec{\alpha} \multimap ((\sigma_{\vec{\alpha},\tau_\alpha} \otimes \sigma_{\vec{\alpha},\tau_\alpha}) \otimes \mathbf{state}) \multimap (\alpha \otimes \alpha \otimes \mathbf{state}) \\ \mathbf{table}_\alpha &\stackrel{\text{def}}{\equiv} \vec{\alpha} \multimap \underbrace{\mathbf{row}_\alpha \otimes \ldots \otimes \mathbf{row}_\alpha}_{m} \otimes \vec{\alpha} \\ \mathbf{row}_\alpha &\stackrel{\text{def}}{\equiv} \vec{\alpha} \multimap (((\mathbf{shift}_\alpha \otimes \mathbf{shift}_\alpha) \otimes \mathbf{shift}_\alpha) \otimes \mathbf{shift}_\alpha) \otimes \vec{\alpha} \\ \mathbf{shift}_\alpha &\stackrel{\text{def}}{\equiv} \vec{\alpha} \multimap \tau_\alpha \\ \sigma_{\alpha,\beta} &\stackrel{\text{def}}{\equiv} ((\mathbf{a\_bool}_{\alpha,\beta} \otimes \mathbf{i}_\alpha) \otimes \alpha) \\ \tau_\alpha &\stackrel{\text{def}}{\equiv} (\alpha \multimap \alpha) \multimap \alpha \multimap \alpha \multimap ((\alpha \otimes \alpha) \otimes \mathbf{state})\ . \end{aligned}$$

It may help also saying that the projections $\Pi_{111}, \Pi_{112}, \Pi_{12}, \Pi_2$ are used in *t_fun* with the types $\mathbf{a\_bool}_{\vec{\alpha},\tau_\alpha}$ here above, because they serve as actual parameter for replacing h^l_r in *comp*.

4.3 Gluing all Together

The whole encoding of the P-Time Turing machine with polynomial $p(x)$ of maximal non null degree ϑ is a term with type $\mathbf{tape} \multimap \S^{\vartheta+5}\mathbf{tape}$, where:

$$\mathbf{tape} \stackrel{\text{def}}{\equiv} \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha \multimap \alpha)\ .$$

The general scheme defining a term representing a tape is:

$$\lambda OO'ZZ'.\S(\lambda xx'.\chi_1(\dots(\chi_p\ x)\dots)\otimes\chi'_1(\dots(\chi'_q\ x')\dots))\ ,$$

where $\chi_i \in \{\bar{!}O, \bar{!}Z\}$ with $1 \leq i \leq p$, $\chi'_j \in \{\bar{!}O', \bar{!}Z'\}$ with $1 \leq j \leq q$, and $p, q \geq 0$.

Appendix B has the details about the whole encoding of the Turing machine. The relation between such encoding and the Turing machine is the obvious one: if the Turing machine yields a tape t_o from an input tape t_i on the alphabet $\{0, 1\}$, the same relation holds between the encodings of t_i and t_o thanks to the term in Appendix B. Its main tasks are: feeding the quantitative part with an integer, obtained from the initial tape, and feeding the qualitative part by the starting configuration, namely the initial tape and the initial state.

5 On the Obvious Encoding

In [1], P-Time completeness of LAL is claimed to hold by saying that it can be proved following [3], where the proof of P-Time completeness for Light Linear Logic is sketched. The proof in [3] is developed on the obvious representation of configurations. For example, the tape $10 \star 10$ with the head reading $\star$ in the state s_i, would be:

$$\lambda OZJ.\S(\lambda x.\bar{!}Z(\bar{!}O\ x))\otimes\lambda OZJ.\S(\lambda x.\bar{!}J(\bar{!}O(\bar{!}Z\ x)))\otimes state_i : \mathbf{config}'\ , \quad (5.1)$$

where $state_i$ encodes s_i, and:

$$\begin{aligned}\mathbf{config}' &\stackrel{\text{def}}{\equiv} \mathbf{tape}'\otimes\mathbf{tape}'\otimes\mathbf{state}'\\ \mathbf{tape}' &\stackrel{\text{def}}{\equiv} \forall\alpha.!(\alpha\multimap\alpha)\multimap!(\alpha\multimap\alpha)\multimap!(\alpha\multimap\alpha)\multimap\S(\alpha\multimap\alpha)\ ,\end{aligned}$$

for some suitable type $\mathbf{state}'$.

Any transition function t_fun' working on (5.1) needs to access the bodies of the §-boxes of the λ-abstractions for producing a new configuration. This can be done *only* by endowing t_fun' with a §-box as well, which border can be merged with those in (5.1). Recall, indeed, that the boxes can not be opened in LAL, but only merged. So, the use of a §-box in t_fun' gets recorded in the co-domain of its type: $\mathbf{config}'\multimap\S\mathbf{config}'$. This type does not allow to iterate t_fun'. As can be seen in the definition of *iter* in Appendix A, the iteration works only on terms with coinciding domain and co-domains. Since the encoding of the P-Time Turing machines in LLL rests on iterating t_fun', we get stuck.

Our, more "parallel", representation of the configurations allows to get an encoding which of the P-TIME Turing machines in LAL. The differences between the sequent calculi of LLL and LAL make Λ_{LA} useless for verifying directly with it that our encoding works on LLL as well. However, we do not see any reasons why the principles our encoding rests on could not be used successfully on LLL. The interested reader could try to follow our encoding idea on a Proof Nets language for LLL. The Proof Nets would avoid useless syntax overheads, caused by the high number of rules defining LLL.

6 Conclusions

This paper has been written as consequence of an attempt to study Light Affine Logic as a programming language, using Curry-Howard principles. The final aim is writing a compact language for Light Affine Logic, which is automatically typable, and P-Time complete. The study of the completeness led to the need of writing down all the details about the encoding of a P-Time Turing machine in Light Affine Logic, following [3]. Something went wrong, so the proof about P-Time completeness of Light Affine Logic was still waiting to be worked out correctly. This is what we have just done, also contributing with introducing a very compact language to program feasible functions.

References

1. A. Asperti. Light Affine Logic. In *Proceedings of Symposium on Logic in Computer Science LICS'98*, 1998.
2. H.P. Barendregt. *The Lambda Calculus*. North-Holland, second edition, 1984.
3. J.-Y. Girard. Light Linear Logic. *Information and Computation*, 143:175 – 204, 1998.
4. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.
5. L. Roversi. Concrete syntax for intuitionistic light affine logic with polymorphic type assignment. In *Sixth Italian Conference on Theoretical Computer Science*. World Scientific, 9 – 11 November (Prato – Italy) 1998.
6. L. Roversi. A polymorphic language which is typable and poly-step. In *Advances in Computing Science – ASIAN'98*, volume LNCS 1538. Springer-Verlag, 8 – 10 December (Manila – The Philippines) 1998.

A Details on the Quantitative Part

Let denote $\tau \otimes \ldots \otimes \tau$, with n elements, by τ_n. Let $p(x)$ be the polynomial $\sum_{i=0}^{\vartheta} a_i x^i$ describing the computational bound of the Turing machine being encoded. Let also $\kappa = \frac{\vartheta(\vartheta+1)}{2}$. The term P encoding $p(x)$ is defined as:

$$
\begin{array}{l}
\lambda x.\S((\lambda y_0^1 \otimes \ldots \otimes y_0^i \otimes \ldots \otimes y_{i-1}^i \otimes \ldots \otimes y_0^\vartheta \otimes \ldots \otimes y_{\vartheta-1}^\vartheta. \\
\quad sum\_int_{\vartheta+1}^{\vartheta+2}\ \S^1(coerc\_int^{\vartheta,0}\ \bar{\S}^1 \langle\!\langle a_0 x^0 \rangle\!\rangle_{y^0}) \\
\qquad\qquad \vdots \\
\qquad\qquad \otimes \S^{i+1}(coerc\_int^{\vartheta-i,0}\ \bar{\S}^{i+1} \langle\!\langle a_i x^i \rangle\!\rangle_{y^i}) \\
\qquad\qquad \vdots \\
\qquad\qquad \otimes \S^{\vartheta+1}(coerc\_int^{0,0}\ \bar{\S}^{\vartheta+1} \langle\!\langle a_\vartheta x^\vartheta \rangle\!\rangle_{y^\vartheta}) \\
)\ \bar{\S}(tuple\_int_\kappa\ x)) : \mathbf{int} \multimap \S^{\vartheta+3}\mathbf{int}
\end{array}
$$

where:

$$\begin{array}{l}\langle\!\langle ax^0 \rangle\!\rangle_y \mapsto \mathit{coerc\_int}^{0,0}\ \overline{a} : \S\mathbf{int} \\ \langle\!\langle ax^n \rangle\!\rangle_y \mapsto \mathit{mult\_int}^n\ \langle y^{n-1}\rangle\ (\mathit{coerc\_int}^{n-1,1}\ \overline{a}) : \S^{n+1}\mathbf{int} \qquad (n \geq 1)\ ,\end{array}$$

being $y_1 \ldots y_n$ the free variables of $\langle\!\langle ax^n \rangle\!\rangle_y$, and:

$$\begin{array}{l}\langle x^0 \rangle \mapsto \mathit{coerc\_int}^{0,0}\ x_0 : \S\mathbf{int} \\ \langle x^n \rangle \mapsto \mathit{mult\_int}^n\ \langle x^{n-1}\rangle\ (\mathit{coerc\_int}^{n-1,1}\ x_n) : \S^{n+1}\mathbf{int} \qquad (n \geq 1)\end{array}$$

being $x_1 \ldots x_n$ the free variables of $\langle x^n \rangle$, and:

$$\begin{array}{rl}
\overline{0} & \stackrel{\text{def}}{\equiv} \lambda X.\S(\lambda x.x) : \mathbf{int} \\
\overline{1} & \stackrel{\text{def}}{\equiv} \lambda X.\S(\lambda x.!X\ x) : \mathbf{int} \\
\mathit{sum_int}_n & \stackrel{\text{def}}{\equiv} \lambda x_1 \otimes \ldots \otimes x_n X.\S(\lambda y.\bar{\S}(x_1\ X)(\ldots(\bar{\S}(x_n\ X)\ y)\ldots)) : \mathbf{int}_n \multimap \mathbf{int} \\
\mathit{sum_int}^p_n & \stackrel{\text{def}}{\equiv} \lambda x_1 \otimes \ldots \otimes x_n.\S^p(\mathit{sum_int}_n\ \bar{\S}^p x_1 \otimes \ldots \otimes \bar{\S}^p x_n) : (\S^p\mathbf{int})_n \multimap \S^p\mathbf{int} \\
\mathit{succ_int} & \stackrel{\text{def}}{\equiv} \lambda x.\mathit{sum_int}_2\ \overline{1} \otimes x : \mathbf{int} \multimap \mathbf{int} \\
\mathit{succ_int}^{p,q} & \stackrel{\text{def}}{\equiv} \lambda x.\S^p(!^q(\mathit{succ_int}\ !^q(\bar{\S}^p x))) : \S^p!^q\mathbf{int} \multimap \S^p!^q\mathbf{int} \\
\overline{0}^{p,q} & \stackrel{\text{def}}{\equiv} \S^p!^q\overline{0} : \S^p!^q\mathbf{int} \\
\mathit{coerc_int}^{p,q} & \stackrel{\text{def}}{\equiv} \lambda x.\S(\bar{\S}(x\ !\mathit{succ_int}^{p,q})\ \overline{0}^{p,q}) : \mathbf{int} \multimap \S^{p+1}!^q\mathbf{int} \\
\mathit{iter} & \stackrel{\text{def}}{\equiv} \lambda x X y.\S(\bar{\S}(\bar{\S}(x\ X)\ \bar{\S}y) : \mathbf{int} \multimap !(\tau \multimap \tau) \multimap \S\tau \multimap \S\tau \\
\mathit{iter}^p & \stackrel{\text{def}}{\equiv} \lambda x y z.\S^p(\mathit{iter}\ \bar{\S}^p x\ \bar{\S}^p y\ \bar{\S}^p z) : \S^p\mathbf{int} \multimap \S^p!(\tau \multimap \tau) \multimap \S^{p+1}\tau \multimap \S^{p+1}\tau \\
\mathit{mult_int} & \stackrel{\text{def}}{\equiv} \lambda x X.\mathit{iter}\ x\ !(\lambda y.\mathit{sum_int}\ !X\ y)\ \overline{0}^{1,0} : \mathbf{int} \multimap !\mathbf{int} \multimap \S\mathbf{int} \\
\mathit{mult_int}^p & \stackrel{\text{def}}{\equiv} \lambda x y.\S^p(\mathit{mult_int}\ \bar{\S}^p x\ \bar{\S}^p y) : \S^p\mathbf{int} \multimap \S^p!\mathbf{int} \multimap \S^{p+1}\mathbf{int} \\
\mathit{tuple_int}_n & \stackrel{\text{def}}{\equiv} \lambda x.\S(\bar{\S}(x\ !(\lambda x_1 \otimes \ldots \otimes x_n.\mathit{succ_int}\ x_1 \otimes \ldots \otimes \mathit{succ_int}\ x_n))\ \overline{0}_n) : \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{int} \multimap \S(\mathbf{int}_n)
\end{array}$$

where $p, q \geq 0$ and $n \geq 1$.

B Details on Gluing all Together: The Turing Machine

Let $P : \mathbf{int} \multimap \S^{\vartheta+3}\mathbf{int}$ be the term encoding the quantitative polynomial of degree ϑ, and T the term encoding the table which the transition function $\mathcal{F}$ rests on. The term encoding the Turing machine is defined as:

$$\begin{array}{l}\lambda t.\mathit{config2tape}^{\vartheta+5}(\S((\lambda t_1 \otimes t_2.\mathit{iter}^{\vartheta+3}\ (P(\mathit{tape2int}\ t_1)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\S^{\vartheta+3}!T) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{starting\_config}^{\vartheta+3}\ t_2) \\ \qquad\qquad\qquad\qquad\qquad\qquad)\ \bar{\S}(\mathit{dbl\_tape}\ t))) : \mathbf{tape} \multimap \S^{\vartheta+5}\mathbf{tape}\end{array}$$

where, for every $p, q \geq 0$ and $n \geq 1$:

$$\begin{aligned}
\mathit{empty_tape} &\stackrel{\text{def}}{\equiv} \lambda OO'ZZ'.\S(\lambda xx'.x \otimes x') \\
\mathit{succ_tape}_{\chi,\chi'} &\stackrel{\text{def}}{\equiv} \lambda tOO'ZZ'.\S(\lambda xx'.(\lambda w \otimes w'.\bar{!}\chi w \otimes \bar{!}\chi' w') \\
&\qquad (\bar{\S}(t\ O\ O'\ Z\ Z')\ x\ x') \\
&\qquad) : \mathbf{tape} \multimap \mathbf{tape} \\
\mathit{succ_tape}_{\chi} &\stackrel{\text{def}}{\equiv} \mathit{succ_tape}_{\chi,I} : \mathbf{tape} \multimap \mathbf{tape} \\
\mathit{succ_tape}_{\chi'} &\stackrel{\text{def}}{\equiv} \mathit{succ_tape}_{I,\chi'} : \mathbf{tape} \multimap \mathbf{tape} \\
\mathit{dbl_succ_tape}_{\chi''} &\stackrel{\text{def}}{\equiv} \lambda x \otimes y.(\mathit{succ_tape}_{\chi''}\ x) \otimes (\mathit{succ_tape}_{\chi''}\ y) : \mathbf{tape}_2 \multimap \mathbf{tape}_2 \\
\mathit{merge_tape} &\stackrel{\text{def}}{\equiv} \lambda w \otimes z.\lambda OO'ZZ'.\S(\lambda xx'.\pi_1(\bar{\S}(w\ O\ O'\ Z\ Z')\ x\ I) \\
&\qquad \otimes \pi_2(\bar{\S}(z\ O\ O'\ Z\ Z')\ I\ x') \\
&\qquad) : \mathbf{tape}_2 \multimap \mathbf{tape} \\
\mathit{merge_tape}^p &\stackrel{\text{def}}{\equiv} \lambda x \otimes y.\S^p(\mathit{merge_tape}\ \bar{\S}^p x\ \bar{\S}^p y) : (\S^p\mathbf{tape})_2 \multimap \S^p\mathbf{tape} \\
\mathit{dbl_merge_tape} &\stackrel{\text{def}}{\equiv} \lambda w_t^1 \otimes w_t^2 \otimes z_t^1 \otimes z_t^2.(\mathit{merge_tape}\ w_t^1\ z_t^1) \otimes (\mathit{merge_tape}\ w_t^2\ z_t^2) : \\
&\qquad \mathbf{tape}_4 \multimap \mathbf{tape}_2 \\
\mathit{empty_tape}^p &\stackrel{\text{def}}{\equiv} \S^p\mathit{empty_tape} : \S^p\mathbf{tape} \\
\mathit{succ_tape}^p_{\chi} &\stackrel{\text{def}}{\equiv} \lambda t.\S^p(\mathit{succ_tape}_{\chi}\ \bar{\S}^p t) : \S^p\mathbf{tape} \multimap \S^p\mathbf{tape} \\
\mathit{coerc_tape}^p &\stackrel{\text{def}}{\equiv} \lambda t.\S(\mathit{merge_tape}^p\ (\bar{\S}(t\ !(\mathit{succ_tape}^p_O)\ !(\mathit{succ_tape}^p_{O'}) \\
&\qquad !(\mathit{succ_tape}^p_Z)\ !(\mathit{succ_tape}^p_{Z'}) \\
&\qquad)\ \mathit{empty_tape}^p\ \mathit{empty_tape}^p)) : \mathbf{tape} \multimap \S^{p+1}\mathbf{tape} \\
\mathit{config2tape} &\stackrel{\text{def}}{\equiv} \lambda cOO'ZZ'.\S(\lambda xx'.(\lambda y \otimes w \otimes z.y \otimes w)(\bar{\S}(c\ O\ O'\ Z\ Z')\ x\ x')) : \\
&\qquad \mathbf{config} \multimap \mathbf{tape} \\
\mathit{config2tape}^p &\stackrel{\text{def}}{\equiv} \lambda c.\S^p(\mathit{config2tape}\ \bar{\S}^p c) : \S^p\mathbf{config} \multimap \S^p\mathbf{tape} \\
\mathit{flatten_tape} &\stackrel{\text{def}}{\equiv} \lambda x \otimes y.\lambda X.\S(\lambda z.\pi_1(\bar{\S}(x\ X\ X\ X\ X)(\pi_2(\bar{\S}(y\ X\ X\ X\ X)\ I\ z))\ I)) : \\
&\qquad \mathbf{tape}_2 \multimap \mathbf{int} \\
\mathit{tape2int} &\stackrel{\text{def}}{\equiv} \lambda t.\mathit{flatten_tape}\ (\bar{\S}(t\ !\mathit{succ_tape}_O\ !\mathit{succ_tape}_O \\
&\qquad !\mathit{succ_tape}_{O'}\ !\mathit{succ_tape}_{O'} \\
&\qquad)\ \mathit{empty_tape}\ \mathit{empty_tape}) : \mathbf{tape} \multimap \mathbf{int} \\
\mathit{starting_config}^p &\stackrel{\text{def}}{\equiv} \lambda t.\S^{p+1}(\lambda OO'ZZ'.\S(\lambda xx'.\bar{\S}(\bar{\S}^{p+1}(\mathit{coerc_tape}^p\ t)\ O\ O'\ Z\ Z')\ x\ x' \\
&\qquad \otimes \mathit{state}_0)) : \mathbf{tape} \multimap \S^{p+1}\mathbf{config} \\
\mathit{dbl_tape} &\stackrel{\text{def}}{\equiv} \lambda t.\S(\mathit{dbl_merge_tape}(\bar{\S}(t\ !\mathit{dbl_succ_tape}_O\ !\mathit{dbl_succ_tape}_{O'} \\
&\qquad !\mathit{dbl_succ_tape}_Z\ !\mathit{dbl_succ_tape}_{Z'} \\
&\qquad)\ \mathit{empty_tape}_2\ \mathit{empty_tape}_2 \\
&\qquad)) : \mathbf{tape} \multimap \S(\mathbf{tape} \otimes \mathbf{config})
\end{aligned}$$

with $\chi \in \{O, Z\} \Leftrightarrow \chi' \equiv I$, $\chi \equiv I \Leftrightarrow \chi' \in \{O', Z'\}$, and $\chi'' \in \{O, O', Z, Z'\}$.

On Guarding Nested Fixpoints

Helmut Seidl and Andreas Neumann

FB IV – Informatik, Universität Trier, D–54286 Trier, Germany
{seidl,neumann}@uni-trier.de

Abstract. For every hierarchical system of equations S over some complete and distributive lattice we construct an equivalent system with the same set of variables which additionally is *guarded*. The price to be paid is that the resulting right-hand sides may grow exponentially. We therefore present methods how the exponential blow-up can be avoided. Especially, the loop structure of the variable dependence graph is taken into account. Also we prove that size $\mathcal{O}(m \cdot |S|)$ suffices whenever S originates from a fixpoint expression where the nesting-depth of fixpoints is at most m. Finally, we sketch an application to regular tree pattern-matching.

Keywords: guardedness, μ-calculus, distributive lattices, loop-connectedness.

1 Introduction

Since Kozen's seminal paper [13] in 1983, the modal μ–calculus has been widely used for specification and verification of properties of concurrent processes. Fixpoint expressions or (slightly more convenient) *hierarchical* systems of equations, however, are considered to be difficult to understand – especially in presence of deep nesting of alternating fixpoints. Therefore, various kinds of normal forms have been suggested in order to ease both theoretical and practical manipulations. One useful additional property of fixpoint expressions (or hierarchical systems of equations) is *guardedness*.

A variable x is *guarded* in the expression e if x occurs only nested inside some application, i.e., as $f(\ldots x \ldots)$ for some operator f. A hierarchical system of equations is called guarded if it does not contain a cyclic variable dependence through unguarded variable occurrences only. Hierarchical systems of *Boolean* equations only use "$\sqcup$" (least upper bound) and "$\sqcap$" (greatest lower bound) in right-hand sides and thus no operators at all. Therefore, *all* occurrences of variables in right-hand sides are unguarded. Consequently, finding equivalent guarded systems means removing cyclic variable dependences completely. Such acyclic systems can be solved in polynomial (even linear) time. Therefore, finding equivalent guarded systems in general cannot be easier than computing solutions of hierarchical systems of Boolean equations.

For hierarchical systems of equations over more complicated complete lattices and with non-empty sets of operators, a guardedness transformation need not necessarily break *all* cyclic variable dependences. It does, however, eliminate

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 484–498, 1999.

"useless" fixpoint iterations, namely those which can be removed without touching operator applications. This preprocessing has been used for simplifying proofs about the μ-calculus [20], in automata constructions [10,16] or constructions of direct proofs of satisfiability [9,11,17].
Notions related to guardedness have been considered in various contexts. In recursive process definitions, guardedness is commonly assumed [1]. Guardedness plays an important role in universal algebra for polynomial equations to have *unique* solutions [5]. Uniqueness of solutions is also central when equations are to be solved over metric spaces [2] (see also section 7).
It is well-known that hierarchical systems of equations can always be transformed into equivalent guarded ones (see, e.g., [20,11]) – if not even part of the "folklore". Here, however, we are interested in designing *efficient* transformation techniques which minimize the encountered overhead. We start our considerations by separating the transformation into two stages (section 3). The first stage performs a (appropriately generalized) control-flow analysis to determine which subexpressions may arrive at which unguarded variables. In the second phase then the actual transformation is performed. What is important here is that each phase operates on the original system – simultaneously on all levels of fixpoints. By this trick, we avoid a potential explosion in size through repeatedly feeding partially transformed systems (possibly of increased sizes) into the same algorithm (e.g., as in [20,11]). While the number of variables of the system produced by the two-stage transformation has not increased, sizes of right-hand sides may have increased exponentially. In order to reduce this extra space, we take into account, *how* the new right-hand sides are constructed through fixpoint iteration. For arbitrary hierarchical systems, we obtain a new upper bound which is related to structural properties of the variable dependence graph (section 5). In the worst case, the blow-up in size of the system still can only be bounded to be exponential in the *alternation-depth* of the original system. Therefore we exhibit useful special classes where just a small polynomial increase suffices – independent of the alternation-depth (section 6). In case of equations over languages of finite trees, we finally show how guardedness transformations can be used to *replace* greatest fixpoints by least ones and thus to remove alternation of fixpoints altogether. This observation can be exploited for compiling powerful tree patterns to finite tree automata (section 7).

2 Hierarchical Systems of Equations

Instead of formally introducing fixpoint expressions, let us immediately consider the slightly more flexible concept of *hierarchical systems of equations*. Assume we are given a complete lattice $\mathbb{D}$ which, as such, is equipped with the binary operations "$\sqcup$" (least upper bound) and "$\sqcap$" (greatest lower bound). Let Σ denote a set of *further* operator symbols where each $f \in \Sigma$ denotes a monotonic function $[\![f]\!] : \mathbb{D}^k \to \mathbb{D}$ for some $k \geq 1$. Operator symbols from Σ denote "real" operations whose applications will not be touched by our transformations.
As right-hand sides of equations we allow expressions built up from formal variables from some set $\mathcal{Z}$ and constants by application of operators from Σ together

with "$\sqcup$" and "$\sqcap$". The set of all these expressions is denoted by $\mathcal{E}_{\Sigma,\mathbb{D}}(\mathcal{Z})$. Every expression $e \in \mathcal{E}_{\Sigma,\mathbb{D}}(\mathcal{Z})$ denotes a function $[\![e]\!] : (\mathcal{Z} \to \mathbb{D}) \to \mathbb{D}$. This function is monotonic, since "$\sqcup$", "$\sqcap$" and all operators from Σ are monotonic.
A *hierarchical* system of equations with free variables from $\mathcal{F}$ is a pair $(S, \mathcal{H})$. S is the finite basic set of equations $z = e_z, z \in \mathcal{Z}$, where for every z, e_z is an expression in $\mathcal{E}_{\Sigma,\mathbb{D}}(\mathcal{Z} \cup \mathcal{F})$, and $\mathcal{H}$ is a *hierarchy* on $\mathcal{Z}$. A hierarchy consists of a sequence $\mathcal{H} = (\langle \mathcal{Z}_r, \lambda_r \rangle, \ldots, \langle \mathcal{Z}_1, \lambda_1 \rangle)$ of mutually disjoint sets $\mathcal{Z}_k$ where $\mathcal{Z} = \mathcal{Z}_r \cup \ldots \cup \mathcal{Z}_1$ together with qualifications $\lambda_k \in \{\mu, \nu\}$. $\mathcal{Z}_k$ is also called the k-th *block* of variables, whereas length r of the hierarchy is called the *alternation-depth* of S. Intuitively, hierarchy $\mathcal{H}$ on S describes the nesting of scopes of variables within which fixpoint iteration is performed: iteration on variables from the same block is performed jointly whereas iteration on variables from block $\mathcal{Z}_i$ should be thought of as nested inside the iteration on variables from $\mathcal{Z}_j, i < j$.

Example 1. Assume we are given the fixpoint expression

$$\mu x_1.a \sqcup (\mu x_2.f\,(x_1, x_2) \sqcup (x_1 \sqcap (\nu x_3.g\,x_3 \sqcup (x_2 \sqcap x_3))))$$

A representation of this expression by a hierarchical system is obtained by introducing an extra equation for each fixpoint subexpression. For our example this gives a set S consisting of:

$$\begin{array}{ll} x_1 = a \sqcup x_2 & x_3 = g\,x_3 \sqcup (x_2 \sqcap x_3) \\ x_2 = f\,(x_1, x_2) \sqcup (x_1 \sqcap x_3) & \end{array}$$

Hierarchy $\mathcal{H}$ is obtained by dividing the set of fixpoint variables into blocks of fixpoints of the same kind for which fixpoint iteration can be performed jointly. For our example expression, we choose $\mathcal{H} = (\langle \mathcal{Z}_2, \mu \rangle, \langle \mathcal{Z}_1, \nu \rangle)$ where $\mathcal{Z}_1 = \{x_3\}$ and $\mathcal{Z}_2 = \{x_1, x_2\}$. □

Usually, if $\mathcal{H}$ is understood, we write S for the hierarchical system.
Fix some $1 \leq k \leq r$, and let $\mathcal{Z}^{(k)} = \mathcal{Z}_k \cup \ldots \cup \mathcal{Z}_1$. Then the k–th *subsystem* S_k of S is given by the set of equations $z = e_z, z \in \mathcal{Z}^{(k)}$, together with hierarchy $\mathcal{H}_k = (\langle \mathcal{Z}_k, \lambda_k \rangle, \ldots, \langle \mathcal{Z}_1, \lambda_1 \rangle)$. Note that the free variables of S_k are contained in $\mathcal{F} \cup \mathcal{Z}_r \cup \ldots \cup \mathcal{Z}_{k+1}$.
An *environment* ρ for S is a mapping $\rho : \mathcal{F} \to \mathbb{D}$. The *semantics* $[\![S]\!]\,\rho$ of S in $\mathbb{D}$ relative to environment ρ is a mapping $\mathcal{Z} \to \mathbb{D}$ defined by induction on the alternation-depth r. For $r \geq 1$, consider the monotonic function $G : (\mathcal{Z}_r \to \mathbb{D}) \to \mathcal{Z}_r \to \mathbb{D}$ given by $G\,\sigma\,z = [\![e_z]\!]\,(\rho + \sigma + [\![S_{r-1}]\!]\,(\rho + \sigma))$ where S_{r-1} is empty in case $r = 1$. Note that we use the "+"–operator to combine two functions with disjoint domains into one. In case $\mathcal{Z}_r$ is qualified as μ, let $\bar{\sigma}$ denote the least fixpoint of G. Otherwise, let $\bar{\sigma}$ denote the greatest fixpoint of G. Then $[\![S]\!]\,\rho$ is defined by $[\![S]\!]\,\rho\,z = \bar{\sigma}\,z$ if $z \in \mathcal{Z}_r$ and $[\![S]\!]\,\rho\,z = [\![S_{r-1}]\!]\,(\rho + \bar{\sigma})\,z$ otherwise.
The set $\mathcal{U}[\![e]\!]$ of *unguarded* variables occurring in e is inductively defined by:

$$\begin{array}{lll} \mathcal{U}[\![d]\!] & = \emptyset & (d \in \mathbb{D}) \\ \mathcal{U}[\![z]\!] & = \{z\} & (z \text{ a variable}) \\ \mathcal{U}[\![f\,(e_1, \ldots, e_k)]\!] & = \emptyset & (f \in \Sigma) \\ \mathcal{U}[\![e_1 \sqcup e_2]\!] & = \mathcal{U}[\![e_1]\!] \cup \mathcal{U}[\![e_2]\!] & \\ \mathcal{U}[\![e_1 \sqcap e_2]\!] & = \mathcal{U}[\![e_1]\!] \cup \mathcal{U}[\![e_2]\!] & \end{array}$$

Sequence $z_1, \ldots, z_m$ of variables of S is called *unguarded cycle* if z_m occurs

unguarded in e_{z_1} and likewise, for $j = 2, \ldots, m$, z_{j-1} occurs unguarded in e_{z_j}. System S is *guarded* iff S does not contain unguarded cycles.

Example 2. Consider the hierarchical system of equations from ex. 1. It contains, e.g., the unguarded cycle x_2, x_3. □

When analyzing guardedness of hierarchical systems, we find it useful to assume w.l.o.g. that right-hand sides e of variables are of one of the following forms:

1. e is a Boolean expression, i.e., in $\mathcal{E}_{\emptyset,\{\bot,\top\}}[\mathcal{Z} \cup \mathcal{F}]$; or
2. e is an operator application $f\,(e_1, \ldots, e_k)$, $k \geq 1$, or a constant from $\mathbb{D}$.

This special form can always be achieved, possibly by introduction of extra auxiliary variables for constants and operator applications.

Example 3. Consider our hierarchical system from ex. 1. We introduce extra variables y_1, y_2, y_3 for expressions a, $f\,(x_1, x_2)$ and $g\,x_3$, respectively, and obtain the equations:

$$x_1 = y_1 \sqcup x_2 \qquad y_1 = a \qquad x_3 = y_3 \sqcup (x_2 \sqcap x_3) \qquad y_3 = g\,x_3$$
$$x_2 = y_2 \sqcup (x_1 \sqcap x_3) \qquad y_2 = f\,(x_1, x_2)$$

The new hierarchy is obtained by adding the new variables to the corresponding blocks: $(\langle \{x_1, x_2, y_1, y_2\}, \mu\rangle, \langle \{x_3, y_3\}, \nu\rangle)$. □

3 The Basic Transformation

A lattice $\mathbb{D}$ is called *distributive* iff it has a least element $\bot$, a greatest element $\top$ and the equations $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$ and $a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$ hold for all $a, b, c \in \mathbb{D}$. Let $\mathbb{B} = \{\bot \sqsubset \top\}$ denote the Boolean lattice, and for (finite) set $\mathcal{Y}$, $\mathbb{B}[\mathcal{Y}]$ denote the complete lattice consisting of all monotonic functions $(\mathcal{Y} \to \mathbb{B}) \to \mathbb{B}$. Facts 1 and 2 are well-known:

Fact 1 *Every element $\phi \in \mathbb{B}[\mathcal{Y}]$, can be uniquely represented by its minimal disjunctive normal form, i.e., $\phi = m_1 \sqcup \ldots \sqcup m_k$ where $m_i = \bigsqcap_{y \in Y_i} y$ for pairwise incomparable subsets $Y_i \subseteq \mathcal{Y}$.* □

Fact 2 *Every mapping $\rho : \mathcal{Y} \to \mathbb{D}$, $\mathbb{D}$ a distributive lattice, can be uniquely extended to a mapping $\rho^* : \mathbb{B}[\mathcal{Y}] \to \mathbb{D}$ with the following properties:*

- $\rho^*\, y = \rho\, y$ *for every* $y \in \mathcal{Y}$;
- $\rho^*\, \bot = \bot$ *and* $\rho^*\, \top = \top$;
- $\rho^*\,(\phi_1 \sqcup \phi_2) = \rho^*\, \phi_1 \sqcup \rho^*\, \phi_2$;
- $\rho^*\,(\phi_1 \sqcap \phi_2) = \rho^*\, \phi_1 \sqcap \rho^*\, \phi_2$.

□

A mapping with the properties listed in fact 2 is also called *morphism* (between distributive lattices). Fact 2 states that $\mathbb{B}[\mathcal{Y}]$ is the *free* distributive lattice (generated from $\mathcal{Y}$). Because of facts 1 and 2, we no longer distinguish between elements in $\mathbb{B}[\mathcal{Y}]$ and expressions built up from constants $\bot, \top$ and variables in $\mathcal{Y}$ by means of applications of $\sqcup$ and $\sqcap$. We obtain:

Proposition 3. *Assume S is a hierarchical system of equations $x = e_x$, $x \in \mathcal{X}$, over distributive lattice $\mathbb{D}$ with free variables from $\mathcal{F}$ where each right-hand side e_x is contained in $\mathbb{B}[\mathcal{X} \cup \mathcal{F}]$. Then for every environment $\rho : \mathcal{F} \to \mathbb{D}$,*

$$\rho^*\,([\![S]\!]\,\emptyset\,x) = [\![S]\!]\,\rho\,x \qquad \text{for all } x \in \mathcal{X}.$$

□

Here, the semantics of S on the left-hand side in the equation is computed over $\mathbb{B}[\mathcal{F}]$ w.r.t. the empty environment whereas on the right-hand side it is computed over $\mathbb{D}$ where the values for the free variables are taken from ρ.
Assume S is a hierarchical system of equations $z = e_z, z \in \mathcal{Z}$, of alternation depth r where the hierarchy of S is given by $\mathcal{H} = (\langle \mathcal{Z}_r, \lambda_r \rangle, \ldots, \langle \mathcal{Z}_1, \lambda_1 \rangle)$. Let $\mathcal{X}$ and $\mathcal{Y}$ denote the sets of variables with Boolean expressions as right-hand sides, and of variables with constants or operator applications as right-hand sides, respectively, and $\mathcal{Z}_k \cap \mathcal{X} = \mathcal{X}_k$. Moreover, let $\bar{S}$ denote the subsystem of S for variables in $\mathcal{X}$. Thus, all the variables in $\mathcal{Y}$ are free variables of $\bar{S}$.
For $k = 1, \ldots, r$, let $\mathcal{F}_k$ denote the set of free variables of subsystem S_k. Then construct $\mathbb{D}_k = \mathbb{B}[\mathcal{Y} \cup \mathcal{F}_k]$, and let $\bar{\sigma}_k : (\mathcal{X}_k \cup \ldots \cup \mathcal{X}_1) \to \mathbb{D}_k$ denote the semantics of $\bar{S}_k$ over $\mathbb{D}_k$ relative to the empty environment. We define a new hierarchical system S' with the same hierarchy as S but the following set of equations:

$$x = \bar{\sigma}_k\, x, \qquad x \in \mathcal{X}_k, 1 \le k \le r \quad y \qquad = e_y,\, y \in \mathcal{Y}$$

In S', variables from $\mathcal{X}_j$ may occur unguarded only in right-hand sides of variables from $\mathcal{X}_i$ where $i < j$. Thus, system S' is guarded.

Example 4. Consider the hierarchical system from ex. 3. The set of free variables is empty whereas the set $\mathcal{Y}$ of variables for operator applications and constants is given by $\mathcal{Y} = \{y_1, y_2, y_3\}$. We obtain:

$$\bar{\sigma}_2\, x_1 = y_1 \sqcup y_2 \qquad \bar{\sigma}_2\, x_2 = y_2 \sqcup (y_1 \sqcap y_3) \qquad \bar{\sigma}_1\, x_3 = y_3 \sqcup x_2$$

Consequently, the newly constructed hierarchical system is constituted by the same hierarchy $\mathcal{H} = (\langle \{x_1, x_2, y_1, y_2\}, \mu \rangle, \langle \{x_3, y_3\}, \nu \rangle)$ together with the equations:

$$\begin{array}{llll} x_1 = y_1 \sqcup y_2 & y_1 = a & x_3 = y_3 \sqcup x_2 & y_3 = g\, x_3 \\ x_2 = y_2 \sqcup (y_1 \sqcap y_3) & y_2 = f\,(x_1, x_2) & & \end{array}$$

□

We claim:

Theorem 1. *Assume S is a hierarchical system of equations over a complete and distributive lattice. Then S and the guarded system S' are equivalent.*

Proof. The following two observations can be deduced from prop. 3:

Fact 4 *Assume $1 \le k \le r$.*

1. *Then $\bar{\sigma}_{k-1}$ is the unique solution over $\mathbb{D}_{k-1}$ of the set of equations*
$$x = \bar{\sigma}_j\, x \quad x \in \mathcal{X}_j, j < k.$$
2. *Assume the k-th block of $\bar{S}$ is qualified μ (ν). Then $\bar{\sigma}_k$ is the least (greatest) solution over $\mathbb{D}_k$ of the set of equations*
$$x = e_x, \quad x \in \mathcal{X}_k \qquad x = \bar{\sigma}_{k-1}\, x, \quad x \in \mathcal{X}_j, j < k$$

□

With fact 4 we prove for $k = 1, \ldots, r$ all ρ and $z \in \mathcal{Z}_k$ that $[\![S_k]\!]\, \rho\, z = [\![S'_k]\!]\, \rho\, z$. Assume this assertion holds for S_{k-1} (if it exists). We successively will transform S_k into the system S'_k. Each of the applied steps will preserve the semantics for the variables in $\mathcal{Z}_k$.
Step 0: We replace the subsystem S_{k-1} of S_k (if existing) with S'_{k-1}. Subsystems S_{k-1} and S'_{k-1} are equivalent by induction hypothesis. In the following, we only transform the k-th block and within this block only the equations with left-hand sides not from $\mathcal{Y}$. Let us call this the *Boolean part* of the k-th block.

Step 1: We *add* a fresh variable x' to (the Boolean part of) the k-th block for every $x \in \mathcal{X}_j$, $j < k$, together with the equation $x' = \bar{\sigma}_j\, x$. Thus, the new right-hand side of variable x' is a copy of the right-hand side of the corresponding variable x. Therefore they evaluate to the same values, and we can replace every occurrence of $x \in \mathcal{X}_j, j < k$, in the Boolean part of the k-th block with the corresponding variable x'.
Step 2: By Bekic principle [3,16], the resulting k-th block is equivalent to a block where the right-hand sides of the x' equal the (unique) solution $\{x' \mid x \in \mathcal{X}_j, j < k\} \to \mathbb{D}_{k-1}$ of the corresponding subset of equations. By fact 4.1, this solution is given by $x' \mapsto \bar{\sigma}_{k-1}\, x$. Therefore in step 2, we *replace* the right-hand side of $x', x \in \mathcal{X}_j\, j < k$, with $\bar{\sigma}_{k-1}\, x$.
Step 3: In the Boolean part of the k-th block, we now rename every variable $x \in \mathcal{X}_k$ with corresponding x', and add new equations $x = x', x \in \mathcal{X}_k$.
Thus, step 3 consists in splitting of the fixpoint computation for the k-th block into an inner iteration on the primed variables x' within the Boolean part, nested inside an iteration on the unprimed variables x. This again preserves the semantics (see, e.g., [3,16]).
Step 4: Assume w.l.o.g. that block k in S_k is qualified μ. By fact 4.2, the least solution of the set of equations over $\mathbb{D}_k$ with left-hand sides $x', x \in \mathcal{X}_j, j \leq k$, is given by $x' \mapsto \bar{\sigma}_k x$. Therefore again by Bekic principle, we now can replace the right-hand sides of all x' with $\bar{\sigma}_k\, x$.

Example 5. Consider the hierarchical system of equations from ex. 3 and let $k = 2$. Then S_1' is given by the equations

$$x_3 = y_3 \sqcup x_2 \qquad y_3 = g\, x_3$$

together with the hierarchy $(\langle \{x_3, y_3\}, \nu \rangle)$. This part of the system will remain unchanged throughout the construction. The only block of equations which we are going to modify is the k-th (i.e., second) block. Initially, it is given by:

$$\begin{array}{ll} x_1 = y_1 \sqcup x_2 & y_1 = a \\ x_2 = y_2 \sqcup (x_1 \sqcap x_3) & y_2 = f\,(x_1, x_2) \end{array}$$

Step 1 adds variable x_3' with right-hand side $\bar{\sigma}_1\, x_3 = y_3 \sqcup x_2$ and results in the set of equations:

$$\begin{array}{lll} x_1 = y_1 \sqcup x_2 & x_3' = y_3 \sqcup x_2 & y_1 = a \\ x_2 = y_2 \sqcup (x_1 \sqcap x_3') & & y_2 = f\,(x_1, x_2) \end{array}$$

Notice that the reference to x_3 in the equation for x_2 has been replaced by a reference to the new variable x_3'. Step 2 is vacuous in this example. Step 3 then renames x_i with x_i' $(i = 1, 2)$ and then adds the equations $x_i = x_i'$. It results in the set of equations:

$$\begin{array}{llll} x_1 = x_1' & x_1' = y_1 \sqcup x_2' & x_3' = y_3 \sqcup x_2' & y_1 = a \\ x_2 = x_2' & x_2' = y_2 \sqcup (x_1' \sqcap x_3') & & y_2 = f\,(x_1, x_2) \end{array}$$

The least solution of the equations for x_1', x_2', x_3' over $\mathbb{D}_2 = \mathbb{B}[\{y_1, y_2, y_3\}]$ is

$$\bar{\sigma}\, x_1' = y_1 \sqcup y_2 \qquad \bar{\sigma}\, x_2' = y_2 \sqcup (y_1 \sqcap y_3) \qquad \bar{\sigma}\, x_3' = y_3 \sqcup y_2$$

which precisely equals $x_i' \mapsto \bar{\sigma}_2\, x_i$. Thus, $\bar{\sigma}$ gives the new right-hand sides for the x_i' in step 4. □

After step 4, the primed variables do no longer occur in right-hand sides – besides in the equations $x = x'$. Therefore, we can replace these equations by $x = \bar{\sigma}_k\, x$

and subsequently remove all variables x' together with their defining equations. The resulting system precisely equals S'_k, and we are done. □

Our construction of an equivalent guarded system is an improvement of the "classical" folklore method for fixpoint expressions sketched in [20,17,11] which introduces a huge (even doubly exponential) increase in size. Note, however, that for the case of fixpoint expressions, our methods will allow us even to construct an equivalent guarded hierarchical system of *polynomial* size (see section 6). The present transformation prepares the ground for such further improvements in the construction since it allows to clearly separate the transformation into two stages. The first stage computes the mappings $\bar{\sigma}_k$, $k = 1, \ldots, r$, whereas only the second stage ultimately transforms S.
Let $n \leq m \leq |S|$ denote the numbers of elements in $\mathcal{X}$ and in $\mathcal{X} \cup \mathcal{Y} \cup \mathcal{F}$, respectively. Then each value in $\mathbb{D}_k$ can be represented by an expression of size $\mathcal{O}(m \cdot 2^m)$. The overall size of the transformed system therefore is bounded by $\mathcal{O}(|S| + n \cdot m \cdot 2^m)$. In order to improve on the (potentially) exponential space to store the expressions $\bar{\sigma}_k\, x$ we take into account *how* the $\bar{\sigma}_k$ can be constructed.

4 Blind Algorithms

Assume we are given a (finite) set S of equations over some lattice $\mathbb{D}$ without free variables and cyclic variable dependences. Then S has a unique solution σ which, given a suitable topological ordering $x_1 < \ldots < x_n$ of the variables, can be computed by successively evaluating the right-hand sides for $x_i, i = 1, \ldots, n$. In this sense, we can view S as a *straight-line program* computing variable assignment σ. Therefore, our goal can be rephrased to design efficient straight-line programs that compute variable assignments $\bar{\sigma}_k$. In contrast to straight-line programs, we will allow redefinitions of variables and use programming-language constructs as **for**-loops or **switch**-statements whose conditions, however, may not depend on $\mathbb{D}$-valued variables. Formally, this can be assured by viewing the lattice elements as abstract values for which there are assignments and operations $\sqcup$ and $\sqcap$, but which are lacking any kind of comparison. Let us call such algorithms *blind*.
Every terminating blind algorithm can be unrolled into a finite sequence of variable assignments. By possibly introducing auxiliary variables, we can always bring this sequence into single-assignment form. Therefore, we obtain:

Fact 5 *For every terminating blind algorithm computing $\sigma : \mathcal{X} \to \mathbb{D}$, there is a straight-line program computing a variable assignment σ' which uses the same number of operations in $\mathbb{D}$ such that $\sigma\, x = \sigma'\, x$ for all $x \in \mathcal{X}$.* □

We conclude that time complexity of blind algorithms for computing $\bar{\sigma}_k$, $k = 1, \ldots, r$, directly can be translated into the *output space* of corresponding guardedness transformations. In the following, we therefore will design efficient blind algorithms for computing the semantics of hierarchical equation systems over distributive complete lattices with $\Sigma = \emptyset$, i.e., operators only from $\{\sqcup, \sqcap\}$.

```
forall (x ∈ 𝒳)  x = ⊥;
for (j = 1, j ≤ k, j++) {
    forall (x ∈ 𝒳)  x' = e_x;
    forall (x ∈ 𝒳)  x = x';
}
```

Fig. 1. Lock-Step Iteration.

5 The General Case

Let S denote a set of equations $x = e_x, x \in \mathcal{X}$, without free variables. over a distributive lattice $\mathbb{D}$ where $\Sigma = \emptyset$. The first algorithm one may think of is *lock-step* iteration as in fig. 1. This algorithm successively computes the n-th approximation of the least fixpoint. Since all values of the next round are computed w.r.t. the old values of the variables, we use a set $\{x' \mid x \in \mathcal{X}\}$ of fresh variables to receive the new values. These are then copied into the $x \in \mathcal{X}$.
The algorithm from fig. 1 finds the least fixpoint after $k = \#\mathcal{X}$ rounds. A straight forward application to hierarchical systems would successively remove fixpoints outside-in by an appropriate unrolling. The structure of variable dependences, however, is not taken into account. Therefore, we prefer to replace lock-step iteration with a *Round-Robin* strategy. For (intra-procedural) data-flow analysis, such an approach has been considered, e.g., by Kam and Ullman [12].
The (variable) *dependence graph* of the set of equations S is the directed graph $G = (\mathcal{X}, E)$ where E consists of all edges (x_1, x_2) where variable x_1 occurs in the right-hand side of variable x_2. A set B of edges of G is called *set of back-edges* if G without edges from B is a dag. The maximal number of edges from B on any cycle-free path in G is called *loop-connectedness* of G (relative to B).
Notice that the loop-connectedness of G relative to B is at most $\#B$ or even $\#\{v \mid (u, v) \in B\}$ which sometimes is less. Deciding in general whether the loop-connectedness relative to some B is $\geq k$ for arbitrary k is NP-complete [7]. Determining a set of back-edges which minimizes the loop-connectedness seems to be an even harder problem. In case, however, graph G is "well-structured" (*reducible*), polynomial algorithms are known both for computing such minimal B as well as the corresponding loop-connectedness [7] (see [8] for precise definitions of reducibility). The polynomial algorithm for reducible graphs also provides us with a heuristics (running in linear time) to compute small sets B of back-edges in arbitrary graphs: just determine a DFS forest T of G, and then choose B as the set of all edges (u, v) of G where v is an ancestor of u w.r.t. T. The resulting set B is at least *locally minimal* in so far as no proper subset is a set of back-edges for G as well.
A good choice for a set B of back-edges as well as a safe approximation of the loop-connectedness relative to B will do for all our subsequent constructions. Worse B as well as less accurate approximations for the loop-connectedness may result in larger outputs but will not affect the correctness of the construction. *Any* choice of B, however, will provide us with an algorithm which is *not worse* than the lock-step algorithm. For the following let us fix a suitable set B of

```
for (i = 1, i ≤ n, i++)  x_i = ⊥;
for (j = 1, j ≤ k, j++)
    for (i = 1, i ≤ n, i++)  x_i = e_i;
```

Fig. 2. Round-Robin Iteration.

back-edges. Let $<$ denote a topological ordering of the variables in $\mathcal{X}$ w.r.t. to the dag obtained from G by removing all edges from B. Assume this ordering is given as $x_1 < x_2 < \ldots < x_n$ where the right-hand side for x_i is given by e_i. Then the new strategy loops through all variables where for variable x_{i+1} we use those values for variables $x_1, \ldots, x_i$ which already have been computed within the same round. The version for least solutions is shown in fig. 2. The dual version for greatest fixpoints differs in that values of variables x_i are initialized with $\top$ (instead of $\bot$). For the case of least solutions, we prove:

Proposition 6. *The Round-Robin algorithm of fig. 2 computes the least solution of S in $c+1$ rounds where c equals the loop-connectedness of S (relative to B).*

Example 6. Consider the set of equations (with $\Sigma = \emptyset$) given by:

$$x_1 = y_1 \sqcup x_2 \qquad x_2 = y_2 \sqcup (x_1 \sqcap x_3) \qquad x_3 = y_3 \sqcup x_2$$

The variable dependence graph is shown in fig. 3. One set of back-edges is given

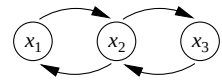


Fig. 3. The Variable Dependences for Ex. 6.

by $B = \{(x_1, x_2), (x_2, x_3)\}$. The loop-connectedness relative to B is 2. Another set of back-edges, however, is given by $B' = \{(x_3, x_2), (x_1, x_2)\}$. For set B', the loop-connectedness equals 1 implying that Round-Robin iteration according to ordering $x_2 < x_1 < x_3$ terminates already after 2 rounds. Starting with initial values $\bot$, we obtain:

	x_2	x_1	x_3
1	y_2	$y_1 \sqcup y_2$	$y_3 \sqcup y_2$
2	$y_2 \sqcup (y_1 \sqcap y_3)$	$y_1 \sqcup y_2$	$y_3 \sqcup y_2$

□

Proof of prop. 6. For simplicity, let us assume that right-hand sides e_i are of one of the forms $a \in \mathbb{D}$, x_j, $x_j \sqcup x_k$ or $x_j \sqcap x_k$ for $x_j, x_k \in V$. The sets $\mathcal{I}_i$ of *intersection trees* for x_i, $i = 1, \ldots, n$, inductively are defined as follows:

- If $e_i \in \mathbb{D}$, then $x_i \in \mathcal{I}_i$;
- If $e_i \equiv x_j$, then $x_i(I) \in \mathcal{I}_i$ for every $I \in \mathcal{I}_j$;
- If $e_i \equiv x_j \sqcup x_k$, then $x_i(I) \in \mathcal{I}_i$ for every $I \in \mathcal{I}_j \cup \mathcal{I}_k$;

Fig. 4. The Graphs $G^{(1)}$ and $G^{(2)}$ for Ex. 3.

- If $e_i \equiv x_j \sqcap x_k$, then $x_i(I_j, I_k) \in \mathcal{I}_i$ for every $I_i \in \mathcal{I}_j$ and $I_k \in \mathcal{I}_k$.

I is called *cycle-free* iff no path in I has more than one occurrence of the same variable. Each intersection tree $I \in \mathcal{I}_i$ represents a value $[\![I]\!]$, namely, the meet over all values corresponding to the leafs of I. Formally, $[\![I]\!]$ is defined as follows:

$$\begin{array}{ll} [\![x_i]\!] & = d \quad \text{if } e_i \equiv d \in \mathbb{D} \\ [\![x_i(I)]\!] & = [\![I]\!] \\ [\![x_i(I_1, I_2)]\!] & = [\![I_1]\!] \sqcap [\![I_2]\!] \end{array}$$

Let $\sigma : \mathcal{X} \to \mathbb{D}$ denote the least solution of S. Then $\sigma\, x_i = \bigsqcup\{[\![I]\!] \mid I \in \mathcal{I}_i\}$, $x_i \in \mathcal{X}$. Now consider an intersection tree $I \in \mathcal{I}_i$ which is not cycle-free. Then we can construct a cycle-free $I' \in \mathcal{I}_i$ such that $[\![I]\!] = [\![I']\!] \sqcap d$ for some $d \in \mathbb{D}$ implying that $[\![I]\!] \sqsubseteq [\![I']\!]$. Hence, it suffices to take least upper bounds just over *cycle-free* intersection trees. Thus the following claim implies our assertion:

Claim: Assume $j \geq 1$. After round j, the value of x_i is an upper bound for $[\![I]\!]$ whenever $I \in \mathcal{I}_i$ is cycle-free and has at most $j-1$ back-edges on every path from a leaf to the root. □

In presence of distributivity, our prop. 6 can be seen as a generalization of Kam and Ullman's result [12] to more general forms of systems of equations.

Let us apply prop. 6 to hierarchical systems with $\Sigma = \emptyset$. We propose an iteration strategy which for alternation-depth r consists in r nested **for**-loops. Each iteration of the outermost loop first evaluates the variables from block r; then it descends into an iteration on the variables of the lower blocks.

Assume $G = (\mathcal{X}, E)$ is the variable dependence graph of hierarchical system S. We construct directed graphs $G^{(k)}$, $k = 1, \ldots, r$, as follows.

- The set of vertices of $G^{(k)}$ is given by $\mathcal{X}$;
- The set of edges of $G^{(k)}$ consists of all pairs (z, x) where $x \in \mathcal{X}_k$ and z occurs in e_x (*primary* edges) together with all pairs (x, z) where $x \in \mathcal{X}_k$, $z \in \mathcal{X}_j, j < k$, and there is a path in $G^{(k-1)}$ from x to z (*derived* edges).

Let c_k denote the minimal loop-connectedness of $G^{(k)}$ relative to sets of back-edges consisting of primary edges only. Then the variables in $\mathcal{X}_k$ can be arranged in such a way that $(c_k + 1)$ iterations of the k-th **for**-loop are sufficient. We call c_k the k-th *derived* loop-connectedness.

Example 7. Consider the hierarchical equation system of ex. 3. The graphs $G^{(1)}$ and $G^{(2)}$ are shown in fig. 4. Since $G^{(1)}$ has only a self-loop, derived loop-connectedness c_1 equals 0. The other graph, $G^{(2)}$, has already been considered in ex. 6. There we found as set of back-edges $B' = \{(x_3, x_2), (x_1, x_2)\}$. Since all edges in B' are primary, we conclude that $c_2 = 1$. □

Let n_k denote the number of variables of the k-th block. By construction, $c_k \leq n_k$ for all k. Recall that $(n_1 + 1) \cdot \ldots \cdot (n_r + 1) \leq (\frac{n}{r} + 1)^r$ where $n = n_1 + \ldots + n_r$.

Combining theorem 1 with the sketched blind algorithm, we obtain:

Theorem 2. *Assume S is a hierarchical system of equations with n variables and alternation-depth r over a complete and distributive lattice, and let $c_1, \ldots, c_r$ denote the sequence of derived loop-connectednesses. Then an equivalent*[1] *guarded hierarchical system can be constructed of size*

$$\mathcal{O}(r \cdot (c_1 + 1) \ldots (c_r + 1) \cdot |S|) \leq \mathcal{O}(r \cdot (\tfrac{n}{r} + 1)^r \cdot |S|). \qquad \square$$

The size of the resulting system is linear in the size of S but still may be exponential in the alternation-depth. Observe, however, that the left estimation of theorem 2 usually is sharper than the right bound which just counts variables and ignores variable dependences.

6 Polynomial Special Cases

Often, the variable dependences of (hierarchical) systems are not "arbitrary". In particular, this is the case when the system is derived from a fixpoint expression as in ex. 1. The key idea for this case is to recursively descend into strongly connected components. We obtain a forest-like decomposition similar to [4,6].
Assume G is a directed graph. A *decomposition forest* (df for short) w for a set V of nodes of G is defined as follows. If $V = \emptyset$, then $w = \epsilon$ (the empty list of trees). Otherwise, let G' denote the subgraph of G with nodes in V.
Case 1: G' is strongly connected. Then $w = (w', x)$ where $x \in V$ and w' is a df for $V \backslash \{x\}$. We call x *exit* of strong component G'.
Case 2: G' is not strongly connected. Then $w = w_1 \ldots w_k$, $k > 1$, where w_j is a df for V_j, and the sequence $V_1, \ldots, V_k$ is a topological ordering of the strong components of G', i.e., whenever an edge of G' goes from V_i to V_j, then $i \leq j$.
Thus, a df is obtained from G by recursively applying two steps: first, decomposition into strongly connected components; second, extracting exits from these.
$\mathsf{depth}(w, x)$ of variable x relative to df w equals the number of parentheses within which x is nested. Formally, if $w = (w_1, x_1) \ldots (w_k, x_k)$ then $\mathsf{depth}(w, x_j) = 1$ for $j = 1, \ldots, k$, and for x occurring in w_j, $\mathsf{depth}(w, x) = 1 + \mathsf{depth}(w_j, x)$. The depth of w then is the maximal depth of a variable occurring in w.
Every directed graph has decomposition forests, but only "well-structured" graphs have decomposition forests which are *exit-post-dominated*. Here, w is an *exit-post-dominated* df (edf for short) iff w is a df where for every subtree (v, h) of w and every edge (x, y), x in (v, h) and y not in (v, h) implies $x = h$.

Example 8. Consider the hierarchical system from ex. 3. Then a df for the variable dependence graph of $\bar{S}$ is given by $w = (((\epsilon, x_3), x_2), x_1)$. Df w is indeed exit-post-dominated. Observe that the exits in this decomposition are nothing but the fixpoint variables of the expression. Another edf, however, which has smaller depth is given by $w' = ((\epsilon, x_1)(\epsilon, x_3), x_2)$. $\square$

The post-dominator relation can be computed in polynomial time [19]. Therefore, it takes only polynomial time to decide whether or not a graph has an edf and,

[1] up to extra auxiliary variables, of course

```
solve(w) {                          scan(w) {
  switch (w) {                        switch (w) {
  case ϵ          : return;           case ϵ          : return;
  case (w1, x)w2 : scan(w1, x);       case (w1, x)w2 : x = ⊥;
                   solve(w1);                          scan(w1);
                   solve(w2);                          x = ex;
}   }                                                  scan(w2);
                                    }   }
```

Fig. 5. The EDF-Algorithm for w.

in case it has, to construct such an edf. Our new algorithm for equation systems over distributive lattices with $\Sigma = \emptyset$ corresponding to edf's is shown in fig. 5. It processes one strong component after the other; within a strong component, it first performs a scan over the whole component. This scan evaluates each variable within the strong component from left to right. After this scan, the final value for the exit has been reached. Then the algorithm descends one level down the edf w. Note that this recursive call of solve reinitializes each variable in the subforest. In case, we are just interested in the least solution, this reinitialization can be abandoned. This is no longer possible, however, for alternating fixpoints.

Proposition 7. *Assume S is a set of equations $x = e_x, x \in \mathcal{X}$, without free variables over a distributive lattice where $\Sigma = \emptyset$, and w is an edf of the variable dependence graph of S. Then:*

1. *The edf-algorithm from fig. 5 computes the least solution of S.*
2. *It evaluates each variable x exactly* $\mathsf{depth}(w, x)$ *times.* □

The correctness of the edf-algorithm crucially depends on w being exit-post-dominated. The advantage of this algorithm (whenever applicable), however, is that some variables may be evaluated significantly fewer times than others. Also, if we are only interested in the variables of an upper fragment of w, reevaluation of the remaining variables can be discarded. The other advantage is that it can be extended to hierarchical systems of equations – without further increase in complexity. Assume S is a hierarchical system of equations where $\Sigma = \emptyset$. Let G denote the variable dependence graph (i.e., the one obtained from S by ignoring the hierarchy). An edf w for G is *leveled* iff for each subtree $t = (w', h)$ of w, variable h has a block number which is at least as big as the block number of every variable occurring in t. We say that S is *expression-like* iff G has a leveled edf.

The main motivation for this definition is that the hierarchical equation systems derived from fixpoint expressions naturally have leveled edf's as defined above. Expression-*like* hierarchical systems, however, are more "liberal" than fixpoint expressions, e.g., by allowing sharing of identical subsystems.

Let us now modify the algorithm from fig. 5 by changing procedure scan to initialize $x = \top$ whenever x is a greatest-fixpoint variable and $x = \bot$ otherwise.

Proposition 8. *Assume S is a hierarchical system of equations without free variables over a distributive lattice with $\Sigma = \emptyset$. If w is a leveled edf for the dependence graph of S, then the modified edf-algorithm for w computes $[\![S]\!]\,\emptyset$.* □

Example 9. Consider the hierarchical system from ex. 1 together with edf $w' = ((\epsilon, x_1)(\epsilon, x_3), x_2)$. First, let us determine the values of $\bar{\sigma}_2$ for x_1 and x_2 in $\mathbb{B}[\{y_1, y_2, y_3\}]$. The computation of the modified edf-algorithm produces the following sequence of variable evaluations:

$$x_1 \;:\; y_1 \qquad x_3 \;:\; y_3 \qquad x_2 \;:\; \boxed{y_2 \sqcup (y_1 \sqcap y_3)} \qquad x_1 \;:\; \boxed{y_1 \sqcup y_2}$$

Final results for $\bar{\sigma}_2$ are enclosed into frame boxes. Notice that we avoided reevaluation of x_3, since the values of $\bar{\sigma}_2$ are just needed for variables from the second block. In order to compute $\bar{\sigma}_1$ for x_3 from the first block, we switch the lattice to $\mathbb{D}_1 = \mathbb{B}[\{y_3, x_1, x_2\}]$. One single evaluation step yields $\bar{\sigma}_1\, x_3 = y_3 \sqcup x_2$. □

Combining theorem 1 with prop. 8, we obtain:

Theorem 3. *Assume S is a hierarchical system of equations over a complete and distributive lattice where $\bar{S}$ is expression-like. Then an equivalent guarded hierarchical system can be constructed of size $\mathcal{O}(m \cdot |S|)$ where m is the depth of a leveled edf for the dependence graph of $\bar{S}$.* □

Applied to some fixpoint expression e, theorem 3 states that an equivalent guarded hierarchical system of equations can be constructed which is just a factor m larger (m the depth of nesting of fixpoints in e).
Another important subclass of (hierarchical) systems of equations is obtained by restricting the usage of "$\sqcap$". Assume S is a hierarchical system of equations with $\Sigma = \emptyset$. S is called *disjunctive* iff each right-hand side e is of the form

$$e ::= x \mid d \mid e_1 \sqcup e_2 \mid e \sqcap d$$

where x denotes a variable and d elements in $\mathbb{D}$. This special form is the generalization of disjunctive Boolean equation systems as considered by Mader [14] to arbitrary distributive lattices. They closely correspond to distributive fixpoint expressions [18]. A simplification of the ideas from the latter paper can be applied to reduce the alternation-depth beforehand to (at most) 2. Using this transformation together with the technique from section 5, we obtain:

Theorem 4. *Assume S is a hierarchical system of equations over a complete and distributive lattice where $\bar{S}$ is disjunctive. Then an equivalent guarded hierarchical system can be constructed of size $\mathcal{O}((n+1) \cdot (c+1) \cdot |S|)$ where n is the number of greatest fixpoint variables and c is the loop-connectedness of the variable dependence graph of $\bar{S}$.* □

7 Applications

Guardedness transformations are especially useful whenever operator applications are "contracting" in some sense. Let us make this idea precise. Let $\mathbb{D}$ denote a complete lattice. Let us consider a metric δ on $\mathbb{D}$ which satisfies the following properties:

$$\begin{array}{lll} (1) & \delta(d_1,d_2) & \leq \ \mathsf{max}(\delta(d_1,d),\delta(d,d_2)) \\ (2) & \delta(d\sqcup d_1, d\sqcup d_2) & \leq \ \delta(d_1,d_2) \\ (3) & \delta(d\sqcap d_1, d\sqcap d_2) & \leq \ \delta(d_1,d_2) \end{array}$$

for all $d, d_1, d_2 \in \mathbb{D}$. A metric satisfying (1) is also called *ultra-metric*. In case, inequalities (2) and (3) hold, we call δ *invariant*. We illustrate these definitions by the following example.

Example 10. Let $\mathbb{T}_\Sigma$ denote the set of languages of *finite* trees over signature Σ. Then $\mathbb{T}_\Sigma$ is a complete and distributive lattice (w.r.t. set inclusion as natural ordering). On $\mathbb{T}_\Sigma$ we define $\delta(L_1, L_2) = 2^{-h}$ where h is the *minimal* depth of a tree in the symmetric difference of L_1 and L_2. Then δ is a metric which satisfies inequalities (1), (2) and (3). □

Operator $f : \mathbb{D}^k \to \mathbb{D}$ is called *contracting*, iff there exists some $0 < \lambda < 1$ such that for all $d_i, d_i' \in \mathbb{D}$ and every j, $\delta(f(d_1,\ldots,d_k), f(d_1',\ldots,d_k')) \leq \lambda \cdot \delta(d_j, d_j')$.

Example 11. Consider the distributive and complete lattice $\mathbb{T}_\Sigma$ from ex. 10. For $a \in \Sigma$, let $[\![a]\!]$ denote the operation of formal application of a. Then $[\![a]\!]$ is contracting with factor $\lambda = \frac{1}{2}$. □

The following theorem is analogous to Banach's fixpoint theorem.

Theorem 5. *Assume $\mathbb{D}$ is a complete lattice and all operators are contracting w.r.t. some invariant ultra-metric on $\mathbb{D}$. Then every finite system of equations over $\mathbb{D}$ without unguarded cycles has a* unique *solution.* □

Proof. W.l.o.g. let us assume that no right-hand side contains unguarded variable occurrences. By structural induction, we find that for variable assignments σ_1, σ_2 and expression e without unguarded variable occurrences, $\delta([\![e]\!]\,\sigma_1, [\![e]\!]\,\sigma_2) \leq \lambda \cdot \mathsf{max}\{\delta(\sigma_1\,x, \sigma_2\,x) \mid x \in \mathcal{X}\}$ for some $0 \leq \lambda < 1$. Now let σ_1 and σ_2 denote the least and greatest solutions of S, respectively, and assume that $\sigma_1 \neq \sigma_2$. Thus, $r = \mathsf{max}\{\delta(\sigma_1\,x, \sigma_2\,x) \mid x \in \mathcal{X}\} > 0$, and there exists some $x \in \mathcal{X}$ such that $r = \delta(\sigma_1\,x, \sigma_2\,x) = \delta([\![e_x]\!]\,\sigma_1, [\![e_x]\!]\,\sigma_2) \leq \lambda \cdot r$ – a contradiction. □

Note that we did not assume that $\mathbb{D}$ is a *complete* metric space. Existence of solutions follows since $\mathbb{D}$ is a complete lattice. The metric is only used to guarantee unicity of solutions.

In [15], we have proposed techniques for pattern matching in finite trees. Here, patterns denote recognizable tree languages for which the element problem must be solved. As a convenient and expressive specification language for recognizable sets we suggested fixpoint expressions. Expressions containing just least fixpoints naturally correspond to (alternating) finite tree automata. In order to allow easy complementation, greatest fixpoints are useful as well. According to ex. 10 and 11, Theorem 5 exhibits an interesting method how greatest fixpoints can be removed. Given a fixpoint expression over $\mathbb{T}_\Sigma$, we proceed as follows:

(0) We construct an equivalent hierarchical system of equations;
(1) We construct an equivalent guarded hierarchical system of equations;
(2) We replace all greatest fixpoints by least ones.

Acknowledgments: We thank André Arnold, Damian Niwinski and Igor Walukiewicz for many inspiring discussions and helpful remarks.

References

1. H.R. Andersen, C. Stirling, and G. Winskel. A Compositional Proof System for the Modal Mu-Calculus. In *IEEE Conf. on Logic in Computer Science (LICS)*, 144–153, 1994.
2. A. Arnold and M. Nivat. Metric Interpretations of Infinite Trees and Semantics of Non-Deterministic Recursive Programs. *Theoretical Computer Science (TCS)*, 11:181–205, 1980.
3. H. Bekic. Definable Operations in General Algebras, and the Theory of Automata and Flowcharts. Technical Report, IBM Labor, Wien, 1967.
4. F. Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *Int. Conf. on Formal Methods in Programming and their Applications*, 128–141. LNCS 735, 1993.
5. B. Courcelle. Basic Notions of Universal Algebra for Language Theory and Graph Grammars. *Theoretical Computer Science (TCS)*, 163(1&2):1–54, 1996.
6. C. Fecht and H. Seidl. A Faster Solver for General Systems of Equations. *Science of Computer Programming (SCP)*, 1999. To appear.
7. A.C. Fong and J.D. Ullman. Finding the Depth of a Flow Graph. *J. of Computer and System Sciences (JCSS)*, 15(3):300–309, 1977.
8. M.S. Hecht. *Flow Analysis of Computer Programs*. North Holland, 1977.
9. R. Kaivola. A Simple Decision Method for the Linear Time Mu-Calculus. In *Int. Workshop on Structures in Concurrency Theory (STRICT) (ed. J. Desel)*, 190–204. Springer-Verlag, Berlin, 1995.
10. R. Kaivola. Fixpoints for Rabin Tree Automata Make Complementation Easy. In *23rd Int. Coll. on Automata, Languages and Programming (ICALP)*, 312–323. LNCS 1099, 1996.
11. R. Kaivola. *Using Automata to Characterise Fixpoint Temporal Logics*. PhD thesis, Dept. of Computer Science, Univ. of Edinburgh, 1996.
12. J.B. Kam and J.D. Ullman. Global Data Flow Analysis and Iterative Algorithms. *J. of the Association for Computing Machinary (JACM)*, 23(1):158–171, 1976.
13. D. Kozen. Results on the Propositional μ-Calculus. *Theoretical Computer Science (TCS)*, 27:333–354, 1983.
14. A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. PhD thesis, TU München, 1997.
15. A. Neumann and H. Seidl. Locating Matches of Tree Patterns in Forests. Technical Report 98-08, University of Trier, 1998. Short version in *18th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, LNCS 1530, 134-145, 1998.
16. D. Niwinski. Fixed Point Characterization of Infinite Behavior of Finite State Systems. *Theoretical Computer Science (TCS)*, 189:1–69, 1997.
17. D. Niwinski and I. Walukiewicz. Games for the μ-Calculus. *Theoretical Computer Science (TCS)*, 163(1&2):99–116, 1996.
18. H. Seidl and D. Niwinski. On Distributive Fixpoint Expressions. Technical Report TR 98-04 (253), University of Warsaw, 1998. To appear in RAIRO.
19. R.E. Tarjan. Finding Dominators in Directed Graphs. *SIAM J. of Computing*, 2(3):211–216, 1974.
20. I. Walukiewicz. Notes on the Propositional μ-Calculus: Completeness and Related Results. Technical Report NS-95-1, BRICS Notes Series, 1995.

A Logical Viewpoint on Process-Algebraic Quotients

Antonín Kučera[⋆1] and Javier Esparza[⋆⋆2]

[1] Faculty of Informatics, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic, tony@fi.muni.cz

[2] Institut für Informatik, Technische Universität München, Arcisstr. 21, D-80290 München, Germany, esparza@in.tum.de

Abstract. We study the following problem: Given a transition system $\mathcal{T}$ and its quotient $\mathcal{T}/\!\sim$ under an equivalence $\sim$, which are the sets $\mathcal{L}$, $\mathcal{L}'$ of Hennessy-Milner formulae such that: if $\varphi \in \mathcal{L}$ and $\mathcal{T}$ satisfies φ, then $\mathcal{T}/\!\sim$ satisfies φ; if $\varphi \in \mathcal{L}'$ and $\mathcal{T}/\!\sim$ satisfies φ, then $\mathcal{T}$ satisfies φ.

1 Introduction

In the equivalence approach to formal verification, the specification and the implementation of a system are typically formalised as transition systems $\mathcal{S}$ and $\mathcal{I}$, and the informal statement 'the implementation satisfies the specification' is formalized as '$\mathcal{S}$ is equivalent to $\mathcal{I}$'. In the modal logic approach, the specification is a modal formula φ, and the statement is formalised as '$\mathcal{I}$ is a model of φ'.

In a seminal paper [7], Hennessy and Milner proved that bisimulation equivalence admits a *modal characterization*: Two (finitely branching) processes are bisimilar if and only if they satisfy exactly the same formulae of Hennessy-Milner logic. This result was later extended to the modal μ-calculus, a much more powerful logic strictly containing many other logics, like CTL, CTL*, and LTL. This showed that it was possible to link two different approaches to formal verification, based on equivalences and modal logics, respectively.

Modal characterizations play an important rôle in practice: Given a very large, or even infinite, transition system $\mathcal{T}$, we would like to obtain a smaller, or at least simpler, transition system $\mathcal{T}'$ which satisfies the specification if and only if $\mathcal{T}$ does. If the specification belongs to a set of formulae $\mathcal{L}$ characterizing an equivalence $\sim$, then we can safely take any $\mathcal{T}'$ satisfying $\mathcal{T} \sim \mathcal{T}'$.

An interesting possibility is to take $\mathcal{T}'$ as the *quotient* $\mathcal{T}/\!\sim$ of $\mathcal{T}$ under $\sim$, whose states are the equivalence classes of the states of $\mathcal{T}$, and whose transitions are given by $[s] \xrightarrow{a} [t]$ only if $s \xrightarrow{a} t$. This works for all equivalences in van Glabbeek's spectrum [18] because they satisfy $\mathcal{T} \sim \mathcal{T}/\!\sim$ (as proved in [13]). Quotients are particularly interesting for bisimulation equivalence for practical reasons, of which we give just two. First, in this case $\mathcal{T}/\!\sim$ can be very efficiently computed for finite transition systems, as shown in [16]. Second, for some classes of real-time and hybrid systems [2,8], the quotient

⋆ Supported by a Research Fellowship granted by the Alexander von Humboldt Foundation and by a grant GA ČR No. 201/97/0456.

⋆⋆ Partially supported by the Teilprojekt A3 of the Sonderforschungsbereich 342

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 499–514, 1999.

under bisimulation of an infinite transition system can be proved to be finite; this makes automatic verification possible, at least in principle.

$\mathcal{T}/\sim$ is guaranteed to satisfy a property of $\mathcal{L}$ if and only if $\mathcal{T}$ does, but maybe this holds for other properties as well? We study this question (in a slightly refined form) within the framework of Hennessy-Milner logic, for arbitrary equivalences. Given a set of formulae characterizing $\sim$, our results determine the sets $\mathcal{L}', \mathcal{L}'' \supset \mathcal{L}$ such that $\mathcal{T}/\sim$ satisfies $\varphi \in \mathcal{L}'$ if $\mathcal{T}$ does, and $\mathcal{T}$ satisfies $\varphi \in \mathcal{L}''$ if $\mathcal{T}/\sim$ does. As we shall see, $\mathcal{L}' \cap \mathcal{L}'' = \mathcal{L}$; the additional formulae of $\mathcal{L}', \mathcal{L}''$ which do not belong to $\mathcal{L}$ can be used by efficient verification semi-algorithms (which produce *yes/no/don't know* answers) – if we want to find out whether $\mathcal{T}$ satisfies some $\varphi \in \mathcal{L}' \cup \mathcal{L}''$, we can first check if $\mathcal{T}/\sim$ satisfies φ; if it is the case and $\varphi \in \mathcal{L}''$, we can conclude that $\mathcal{T}$ satisfies φ. If $\mathcal{T}/\sim$ does not satisfy φ and $\varphi \in \mathcal{L}'$, we conclude that $\mathcal{T}$ does not satisfy φ. In the other cases we 'don't know'.

The paper is organized as follows. Section 2 contains preliminary definitions. In Section 3.1, as a warm-up, we determine the set $\mathcal{L}'$ preserved by *any* transition system $\mathcal{T}'$ satisfying $\mathcal{T} \sim \mathcal{T}'$. In Section 3.2, the core of the paper, we determine the set $\mathcal{L}'$ preserved by the quotient $\mathcal{T}/\sim$. In Section 4 we apply our results to the equivalences in van Glabbeek's hierarchy. Section 5 contains conclusions and comments on related and future work.

2 Definitions

Let $Act = \{a, b, c, \dots\}$ be a countably infinite set of *atomic actions* (which is fixed for the rest of this paper).

Definition 1. *A transition system (T.S.) is a triple $\mathcal{T} = (S, \mathcal{A}, \rightarrow)$ where S is a set of* states, *$\mathcal{A} \subseteq Act$, and $\rightarrow \subseteq S \times \mathcal{A} \times S$ is a* transition relation. *We say that $\mathcal{T}$ is* finitely-branching *iff for every $s \in S$, $a \in \mathcal{A}$ the set $\{t \mid s \xrightarrow{a} t\}$ is finite.* Processes *are understood as (being associated with) states in finitely-branching transition systems.*

In the rest of this paper we only consider finitely-branching T.S. (this restriction is harmless from the 'practical' point of view, but it has important 'theoretical' consequences as it, e.g., allows to prevent the use of infinite conjunctions in our future constructions).

As usual, we write $s \xrightarrow{a} t$ instead of $(s, a, t) \in \rightarrow$ and we extend this notation to elements of $\mathcal{A}^*$ in a standard way. A state t is *reachable* from a state s iff $s \xrightarrow{w} t$ for some $w \in \mathcal{A}^*$. If s is a state of $\mathcal{T}$, then $\mathcal{T}(s)$ denotes the transition system $(S', \mathcal{A}, \rightarrow')$ where $S' = \{t \in S \mid s \xrightarrow{w} t\}$ for some $w \in \mathcal{A}^*$, and $\rightarrow'$ is the induced restriction of $\rightarrow$. The set of actions which is used in the underlying transition system of a process p is denoted by $Act(p)$ (sometimes we work with processes whose associated transition system has not been explicitly defined). Properties which have been originally defined for transition systems are often also used for processes; in that case we always mean that the underlying transition system has the property (for example, we can speak about the set of states and actions of a given process).

Definition 2. *Let $\mathcal{T}_1 = (S_1, \mathcal{A}_1, \rightarrow_1)$, $\mathcal{T}_2 = (S_2, \mathcal{A}_2, \rightarrow_2)$ be transition systems. A (total) function $f : S_1 \rightarrow S_2$ is a* homomorphism *from $\mathcal{T}_1$ to $\mathcal{T}_2$ iff $\forall s, t \in S_1, a \in Act$: $s \xrightarrow{a}_1 t \Longrightarrow f(s) \xrightarrow{a}_2 f(t)$.*

Definition 3. *A* renaming *is an (arbitrary) injective function* $r : Act \to Act$. *For every transition system* $\mathcal{T} = (S, \mathcal{A}, \to)$ *we define the* r-renamed *transition systems* $r(\mathcal{T}) = (S, r(\mathcal{A}), \to')$ *where* $s \xrightarrow{b}{}' t$ *iff* $s \xrightarrow{a} t$ *and* $r(a) = b$.

2.1 Process Descriptions

In this section we briefly introduce and motivate the problem which is considered in this paper.

Transition systems are widely accepted as a convenient model of concurrent and distributed systems. A lot of verification problems (safety, liveness, etc.) can be thus reduced to certain properties of processes (states). A major difficulty is that in practice we often meet systems which have a very large (or even infinite) state-space. A natural idea how to decrease computational costs of formal verification is to replace a given process with some 'equivalent' and smaller one (which can be then seen as its 'description').

In this paper we consider two types of process descriptions ($\sim$-representations and $\sim$-characterizations), which are determined by a chosen *process equivalence* $\sim$ (by a 'process equivalence' we mean an arbitrary equivalence on the class of all processes, i.e., states in finitely-branching T.S.).

Definition 4. *Let* $\sim$ *be a process equivalence. A process* t *is a* $\sim$-representation *of a process* s *iff* $s \sim t$.

Definition 5. *Let* $\sim$ *be a process equivalence. The* $\sim$-characterization *of a process* s *of a transition system* $\mathcal{T} = (S, \mathcal{A}, \to)$ *is the process* $[s]$ *of* $\mathcal{T}/\!\sim = (S/\!\sim, \mathcal{A}, \mapsto)$ *where* $S/\!\sim$ *is the set of all* $\sim$*-classes of* S *(the class containing* s *is denoted by* $[s]$*) and* $\mapsto$ *is the least relation satisfying* $s \xrightarrow{a} t \Longrightarrow [s] \overset{a}{\mapsto} [t]$.

Observe that the $\sim$-characterization of s is essentially the quotient of s under $\sim$. We use the word 'characterization' because for every 'reasonable' process equivalence $\sim$ (see Lemma 6) we have that $s \sim [s]$ for each process s; hence, the $\sim$-characterization of s describes not only the behaviour of s (as $\sim$-representations of s do), but also the behaviour of all reachable states of s, i.e., it *characterizes* the whole state-space of s. More precisely, for every state t of the process s there is an equivalent state $[t]$ of the process $[s]$. Therefore, we intuitively expect that $\sim$-characterizations should be more robust than $\sim$-representations. This intuition is confirmed by main theorems of Section 3. Also note that the same process can have many different $\sim$-representations, but its $\sim$-characterization is unique.

Definition 6. *Let* P *be a property of processes,* $\sim$ *a process equivalence. We say that* P *is*

- preserved *by* $\sim$*-representations (or* $\sim$*-characterizations) iff whenever* t *is a* $\sim$*-representation (or the* $\sim$*-characterization) of* s *and* s *satisfies* P*, then* t *satisfies* P*;*
- reflected *by* $\sim$*-representations (or* $\sim$*-characterizations) iff whenever* t *is a* $\sim$*-representation (or the* $\sim$*-characterization) of* s *and* t *satisfies* P*, then* s *satisfies* P.

An immediate consequence of the previous definition is the following:

Lemma 1. *Let $\sim$ a process equivalence. A property P is preserved by $\sim$-representations (or $\sim$-characterizations) iff $\neg P$ is reflected by $\sim$-representations (or $\sim$-characterizations).*

The question considered in this paper is what properties expressible in Hennessy-Milner logic (see the next section) are preserved and reflected by $\sim$-representations and $\sim$-characterizations for a given process equivalence $\sim$, i.e., to what extent are the two kinds of process descriptions 'robust' for a given $\sim$. As we shall see, we can give a complete classification of those properties if the equivalence $\sim$ satisfies certain (abstractly formulated) conditions. Intuitively, we put more and more restrictions on $\sim$ which allow us to prove more and more things; as we shall see in Section 4, all those restrictions are 'reasonable' in the sense that (almost) all existing (i.e., studied) process equivalences satisfy them. See Section 4 for details.

2.2 Hennessy-Milner Logic

Formulae of Hennessy-Milner (H.M.) logic have the following syntax (a ranges over Act):

$$\varphi ::= \mathtt{tt} \mid \varphi \wedge \varphi \mid \neg\varphi \mid \langle a \rangle \varphi$$

The *denotation* $[\![\varphi]\!]$ of a formula φ on a transition system $\mathcal{T} = (S, \mathcal{A}, \rightarrow)$ is defined as follows:

$$\begin{aligned} [\![\mathtt{tt}]\!] &= S \\ [\![\varphi \wedge \psi]\!] &= [\![\varphi]\!] \cap [\![\psi]\!] \\ [\![\neg\varphi]\!] &= S - [\![\varphi]\!] \\ [\![\langle a \rangle \varphi]\!] &= \{s \in S \mid \exists t \in S : s \xrightarrow{a} t \wedge t \in [\![\varphi]\!]\} \end{aligned}$$

Instead of $s \in [\![\varphi]\!]$ we usually write $s \models \varphi$. The other boolean connectives are introduced in a standard way; we also define $\mathtt{ff} \equiv \neg\mathtt{tt}$ and $[a]\varphi \equiv \neg\langle a \rangle \neg\varphi$. The *depth* of a formula φ, denoted $depth(\varphi)$, is defined inductively by

- $depth(\mathtt{tt}) = 0$,
- $depth(\varphi \wedge \psi) = \max\{depth(\varphi), depth(\psi)\}$,
- $depth(\neg\varphi) = depth(\varphi)$,
- $depth(\langle a \rangle \varphi) = 1 + depth(\varphi)$.

The set of actions which are used in a formula φ is denoted by $Act(\varphi)$ (note that $Act(\varphi)$ is always finite).

Definition 7. *Let $\mathcal{A} \subseteq Act$. A* Tree *over $\mathcal{A}$ is any directed binary tree with root r whose edges are labelled by elements of $\mathcal{A}$ satisfying the following condition: if p, q are a-successors of a node s, where $a \in \mathcal{A}$, then the subtrees rooted by p, q are not isomorphic.* Tree-processes *are associated with roots of Trees (we do not distinguish between Trees and Tree-processes in the rest of this paper). Note that for every $k \in \mathbb{N}_0$ and every finite $\mathcal{A} \subseteq Act$ there are only finitely many Trees over $\mathcal{A}$ whose depth is at most k (up to isomorphism). We denote this finite set of representatives by $Tree(\mathcal{A})_k$.*

It is a standard result that for every process s there is a Tree T_s over $Act(s)$ (possibly of infinite depth) such that s and T_s satisfy exactly the same H.M. formulae (cf. [15]). One can also easily prove the following:

Lemma 2. *Formulae φ, ψ of H.M. logic are equivalent iff they agree on every element of $Tree(\mathcal{A})_k$ where $\mathcal{A} = Act(\varphi) \cup Act(\psi)$ and $k = \max\{depth(\varphi), depth(\psi)\}$.*

For every renaming r and a H.M. formula φ we define the formula $r(\varphi)$ which is obtained from φ by substituting each $\langle a \rangle$ with $\langle r(a) \rangle$.

Lemma 3. *For every process s, renaming r, and H.M. formula φ we have that $s \models \varphi$ iff $r(s) \models r(\varphi)$.*

In the next section we also need the following tools:

Definition 8. *Let φ be a H.M. formula, s a process. For a given occurrence of a subformula ψ in φ we define its* diamond-depth, *denoted $d(\psi)$, to be the number of $\langle b \rangle$-modalities which have the occurrence of ψ in their scope. The set of all actions which are used in those modalities is denoted by $\mathcal{A}_d(\psi)$. Finally, we use $\mathcal{R}_s(\psi)$ to denote the set of all states which are reachable from s via a sequence of (exactly) $d(\psi)$ transitions whose actions are contained in $\mathcal{A}_d(\psi)$.*

Lemma 4. *Let φ be a H.M. formula. Let φ' be the formula obtained from φ by substituting (given occurrences of) its subformulae $\psi_1, \ldots, \psi_n$ by H.M. formulae $\xi_1, \ldots, \xi_n$, respectively. Let s be a process such that $s \models \varphi$ and for all $i \in \{1, \ldots, n\}$, $s' \in \mathcal{R}_s(\psi_i)$ one of the following conditions holds:*

1. *$s' \models \psi_i \iff s' \models \xi_i$*
2. *$s' \models \xi_i$ and the occurrence of ψ_i in φ is not within the scope of any negation.*

Then $s \models \varphi'$.

3 The Classification

In this section we give a complete classification of H.M. properties which are preserved/reflected by $\sim$-representations and $\sim$-characterizations for certain classes of process equivalences which satisfy some (abstractly formulated) conditions. From the very beginning, we restrict ourselves to those equivalences which have a *modal characterization*.

Definition 9. *Let $\sim$ be a process equivalence. We say that $\sim$ has a* modal characterization *iff there is a set $\mathcal{H}$ of H.M. formulae s.t. for all processes s, t we have that $s \sim t$ iff s and t satisfy exactly the same formulae of $\mathcal{H}$.*

Observe that the same equivalence can have many different modal characterizations. Sometimes we also use the following notation (where s is a process): $\mathcal{H}_\mathcal{A} := \{\varphi \mid \varphi \in \mathcal{H} \wedge Act(\varphi) \subseteq \mathcal{A}\}$, $\mathcal{H}^k_\mathcal{A} := \{\varphi \mid \varphi \in \mathcal{H}_\mathcal{A} \wedge depth(\varphi) \leq k\}$, $\mathcal{H}(s) := \{\varphi \mid \varphi \in \mathcal{H} \wedge s \models \varphi\}$, and $\mathcal{H}_\mathcal{A}(s) := \{\varphi \mid \varphi \in \mathcal{H}_\mathcal{A} \wedge s \models \varphi\}$. Note that if $\mathcal{A}$ is finite, then $\mathcal{H}^k_\mathcal{A}$ contains only finitely many pairwise nonequivalent formulae. In that case we can thus consider $\mathcal{H}^k_\mathcal{A}$ to be a *finite* set.

3.1 H.M. Properties Preserved by $\sim$-Representations

Theorem 1. *Let $\mathcal{H}$ be a modal characterization of a process equivalence $\sim$. Then every formula ϑ which is a boolean combination of formulae from $\mathcal{H}$ is preserved by $\sim$-representations.*

The previous theorem is in fact a trivial consequence of Definition 9. Now we would like to prove a kind of 'completeness' result saying that nothing else (except for formulae which are equivalent to boolean combinations of formulae from $\mathcal{H}$) is preserved by $\sim$-representations. However, this property does *not* hold for an arbitrary modal characterization $\mathcal{H}$; it is demonstrated by the following counterexample:

Example 1. Let $\sim$ be defined as follows: $s \sim t$ iff $a \in Act(s) \cap Act(t)$, or $Act(s) = Act(t)$. Let $\mathcal{M} = \{(\mathcal{A}_1, \mathcal{A}_2) \mid \mathcal{A}_1, \mathcal{A}_2$ are finite, nonempty, and disjoint subsets of $Act\}$. The equivalence $\sim$ has a modal characterization

$$\mathcal{H} = \{\langle a\rangle \mathtt{tt} \vee (\bigwedge_{b \in \mathcal{A}_1} \langle b\rangle \mathtt{tt} \wedge \bigwedge_{c \in \mathcal{A}_2} \neg\langle c\rangle \mathtt{tt}) \mid (\mathcal{A}_1, \mathcal{A}_2) \in \mathcal{M}\}$$

Now observe that the formula $\langle a\rangle \mathtt{tt}$ is preserved by $\sim$-representations, but it is not equivalent to any boolean combination of formulae from $\mathcal{H}$.

However, a simple assumption about $\mathcal{H}$ which is formulated in the next definition makes a completeness proof possible.

Definition 10. *We say that a modal characterization $\mathcal{H}$ of a process equivalence $\sim$ is* well-formed *iff whenever $\varphi \in \mathcal{H}$ and $\langle a\rangle\psi$ is an occurrence of a subformula in φ, then also $\varphi' \in \mathcal{H}$ where φ' is obtained from φ by substituting the occurrence of $\langle a\rangle\psi$ with $\mathtt{ff}$.*

As we shall see in Section 4, all 'real' process equivalences which have a modal characterization also have a well-formed modal characterization. An important (and naturally-looking) property of process equivalences which have a well-formed modal characterization is presented in the following lemma:

Lemma 5. *Let $\sim$ be a process equivalence having a well-formed modal characterization $\sim$. Let $\mathcal{A} \subseteq Act$, $k \in \mathbb{N}_0$. For all $T, T' \in Tree(\mathcal{A})_k$ we have that $T \sim T'$ iff T and T' satisfy exactly the same formulae of $\mathcal{H}^k_{\mathcal{A}}$.*

Proof. The '$\Rightarrow$' direction is obvious. Now if suffices to realize that if T and T' are distinguished by some $\varphi \in \mathcal{H}$, then they are also distinguished by the formula $\varphi' \in \mathcal{H}^k_{\mathcal{A}}$ which is obtained form φ by substituting every occurrence of a subformula $\langle a\rangle\psi$, which is within the scope of k other $\langle b\rangle$-modalities or where $a \notin \mathcal{A}$, with $\mathtt{ff}$. The formulae φ and φ' agree on every element of $Tree(\mathcal{A})_k$, because the occurrences of subformulae in φ which have been substituted by $\mathtt{ff}$ during the construction of φ' are evaluated to false anyway. □

Theorem 2. *Let $\sim$ be a process equivalence having a well-formed modal characterization $\mathcal{H}$. Then every formula φ of H.M. logic which is preserved by $\sim$-representations is equivalent to a boolean combination of formulae from $\mathcal{H}$.*

Proof. Let φ be a formula preserved by $\sim$-representations, $k = depth(\varphi)$, $\mathcal{A} = Act(\varphi)$ (note that $\mathcal{A}$ is *finite*). For every $T \in Tree(\mathcal{A})_k$ we construct the formula

$$\psi_T \equiv \bigwedge_{\substack{\varrho \in \mathcal{H}^k_{\mathcal{A}} \\ T \models \varrho}} \varrho \ \wedge \bigwedge_{\substack{\varrho \in \mathcal{H}^k_{\mathcal{A}} \\ T \not\models \varrho}} \neg \varrho$$

Now let

$$\psi \equiv \bigvee_{\substack{T \in Tree(\mathcal{A})_k \\ T \models \varphi}} \psi_T$$

We show that φ and ψ are equivalent. To do that, it suffices to show that φ and ψ agree on every $T_1 \in Tree(\mathcal{A})_k$ (see Lemma 2).

- Let $T_1 \in Tree(\mathcal{A})_k$ s.t. $T_1 \models \varphi$. As $T_1 \models \psi_{T_1}$, we also have $T_1 \models \psi$.
- Let $T_1 \in Tree(\mathcal{A})_k$ s.t. $T_1 \models \psi$. Then there is $T_2 \in Tree(\mathcal{A})_k$ s.t. $T_2 \models \varphi$ and $T_1 \models \psi_{T_2}$. As $T_1 \models \psi_{T_2}$, the Trees T_1, T_2 satisfy exactly the same formulae of $\mathcal{H}^k_{\mathcal{A}}$. Hence, $T_1 \sim T_2$ due to Lemma 5. As φ is preserved by $\sim$-representations, T_1 is a $\sim$-representation of T_2, and $T_2 \models \varphi$, we also have $T_1 \models \varphi$. □

Theorem 1 and 2 give a complete classification of those H.M. properties which are preserved and reflected (see Lemma 1) by $\sim$-representations for a process equivalence $\sim$ which has a well-formed modal characterization $\mathcal{H}$.

3.2 H.M. Properties Preserved by $\sim$-Characterizations

Now we establish analogous results for $\sim$-characterizations. As we shall see, this problem is more complicated.

The first difficulty has been indicated already in Section 2.1 – it does not have too much sense to speak about $\sim$-characterizations if we are not guaranteed that $s \sim [s]$ for every process s. Unfortunately, there *are* process equivalences (even with a well-formed modal characterization) which do not satisfy this basic requirement.

Example 2. Let $\sim$ be defined as follows: $s \sim t$ iff for each $w \in Act^*$ s.t. $length(w) = 2$ we have that $s \stackrel{w}{\rightarrow} s'$ for some s' iff $t \stackrel{w}{\rightarrow} t'$ for some t'. The equivalence $\sim$ has a well-formed modal characterization

$$\mathcal{H} = \{\langle a \rangle \langle b \rangle \mathtt{tt} \mid a, b \in Act\} \cup \{\langle a \rangle \mathtt{ff} \mid a \in Act\} \cup \{\mathtt{ff}\}$$

Now let s be a process where $s \stackrel{a}{\rightarrow} t, s \stackrel{b}{\rightarrow} u, u \stackrel{c}{\rightarrow} v$, and t, u, v do not have any other transitions. Then $t \sim u \sim v$, hence $[s] \stackrel{ac}{\rightarrow} [v]$, and therefore $s \not\sim [s]$.

However, there is a simple (and reasonable) condition which guarantees what we need.

Definition 11. *Let $\sim$ be a process equivalence. We say that $\sim$ has a* closed *modal characterization iff it has a modal characterization $\mathcal{H}$ which is closed under subformula (i.e., whenever $\varphi \in \mathcal{H}$ and ψ is a subformula of φ, then $\psi \in \mathcal{H}$).*

A closed modal characterization is a particular case of a *filtration*. The next lemma is a well-known result of modal logic, stating that a model and its quotient through a filtration agree on every formula of the filtration [4]. We include a proof for the sake of completeness .

Lemma 6. *Let $\sim$ be a process equivalence having a closed modal characterization. Then $s \sim [s]$ for every process s.*

Proof. Let $\mathcal{H}$ be a closed modal characterization of $\sim$. We prove that for every $\varphi \in \mathcal{H}$ and every process s we have $s \models \varphi \iff [s] \models \varphi$ (i.e., $s \sim [s]$). By induction on the structure of φ.

- $\varphi \equiv \mathtt{tt}$. Immediate.
- $\varphi \equiv \neg\psi$. Then $\psi \in \mathcal{H}$ and $s \models \psi \iff [s] \models \psi$ by induction hypotheses. Hence also $s \models \neg\psi \iff [s] \models \neg\psi$ as required.
- $\varphi \equiv \psi \wedge \xi$. Then $\psi, \xi \in \mathcal{H}$. If $\psi \wedge \xi$ distinguishes between s and $[s]$, then ψ or ξ distinguishes between the two processes as well; we obtain a contradiction with induction hypotheses.
- $\varphi \equiv \langle a \rangle \psi$.
  - ($\Rightarrow$) Let $s \models \langle a \rangle \psi$. Then there is some t such that $s \xrightarrow{a} t$ and $t \models \psi$. Therefore, $[s] \overset{a}{\mapsto} [t]$ and as $\psi \in \mathcal{H}$, we can use induction hypothesis to conclude $[t] \models \psi$. Hence, $[s] \models \langle a \rangle \psi$.
  - ($\Leftarrow$) Let $[s] \models \langle a \rangle \psi$. Then $[s] \overset{a}{\mapsto} [t]$ for some $[t]$ s.t. $[t] \models \varphi$. By Definition 5 there are s', t' such that $s \sim s'$, $t \sim t'$, and $s' \xrightarrow{a} t'$. As $[t] = [t']$, we have $[t'] \models \psi$ and hence $t' \models \psi$ by induction hypotheses. Therefore, $s' \models \langle a \rangle \psi$. As $s \sim s'$ and $\langle a \rangle \psi \in \mathcal{H}$, we also have $s \models \langle a \rangle \psi$ as needed (remember that formulae of $\mathcal{H}$ cannot distinguish between equivalent processes by Definition 9).

□

According to our intuition presented in Section 2.1, $\sim$-characterizations should be more robust then $\sim$-representations, i.e., they should preserve more properties. The following definition gives a 'syntactical template' which allows to construct such properties.

Definition 12. *Let $\mathcal{S}$ be a set of H.M. formulae. The set of* diamond *formulae over $\mathcal{S}$, denoted $\mathcal{D}(\mathcal{S})$, is defined by the following abstract syntax equation:*

$$\varphi ::= \vartheta \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle a \rangle \varphi$$

Here a ranges over Act, and ϑ ranges over boolean combinations of formulae from $\mathcal{S}$. The set $\mathcal{B}(\mathcal{S})$ of box *formulae over $\mathcal{S}$ is defined in the same way, but we use $[a]$-modality instead of $\langle a \rangle$.*

Theorem 3. *Let $\sim$ be a process equivalence having a closed modal characterization $\mathcal{H}$. Then every formula of $\mathcal{D}(\mathcal{H})$ is preserved by $\sim$-characterizations.*

Proof. Let $\varphi \in \mathcal{D}(\mathcal{H})$. By induction on the structure of φ:

- $\varphi \equiv \vartheta$. It suffices to realize that ϑ is preserved by $\sim$-representations (Theorem 2) and every $\sim$-characterization is also a $\sim$-representation (Lemma 6).
- $\varphi \equiv \varphi_1 \wedge \varphi_2$, or $\varphi \equiv \varphi_1 \vee \varphi_2$ where φ_1, φ_2 are preserved. Immediate.
- $\varphi \equiv \langle a \rangle \varphi_1$ where φ_1 is preserved. Let p be an arbitrary process s.t. $p \models \langle a \rangle \varphi_1$. Then there is $p \xrightarrow{a} p'$ s.t. $p' \models \varphi_1$. By definition of $\sim$-characterization we have $[p] \xmapsto{a} [p']$. Moreover, $[p'] \models \varphi_1$ as φ_1 is preserved. Hence, $[p] \models \langle a \rangle \varphi_1$ as needed. □

In order to prove the corresponding completeness result, we need some additional assumptions about $\sim$ and $\mathcal{H}$.

Definition 13. *Let $\sim$ be a process equivalence. We say that $\sim$ has a* good *modal characterization iff it has a closed modal characterization $\mathcal{H}$ which satisfies the following conditions:*

- *if $\varphi \in \mathcal{H}$, then also $\langle a \rangle \varphi \in \mathcal{H}$ for every $a \in Act$;*
- *if $\varphi \in \mathcal{H}$, then also $r(\varphi) \in \mathcal{H}$ for every renaming r;*
- *if $\langle a \rangle \psi$ is an occurrence of a subformula in φ, then also $\varphi', \varphi'' \in \mathcal{H}$ where φ' and φ'' are the formulae obtained from φ by substituting the occurrence of $\langle a \rangle \psi$ with $\mathtt{tt}$ and $\mathtt{ff}$, respectively;*
- *if $\varphi \in \mathcal{H}$ and $\neg\psi$ is a subformula of φ, then also $\neg\xi \in \mathcal{H}$ for every subformula ξ of ψ;*
- *there are processes s, t such that $Act(s) \cup Act(t)$ is finite and $\mathcal{H}(s) \subset \mathcal{H}(t)$.*

The requirements of Definition 13 look strange at first glance. In fact, the first four of them only eliminate a lot of 'unnatural' process equivalences from our considerations. The last requirement is also no problem, because the majority of 'real' process equivalences are defined as kernels of certain preorders, and one can always find processes s, t such that s is 'strictly less' than t in the preorder.

Now we present a sequence of technical lemmas which are then used to prove the last main theorem of our paper.

Lemma 7. *Let $\mathcal{H}$ be a good modal characterization of a process equivalence $\sim$. For every $n \in \mathbb{N}$ and every finite $\mathcal{A} \subseteq Act$ there are processes $p_1, \cdots, p_n$ such that $Act(p_i)$ is finite, $Act(p_i) \cap \mathcal{A} = \emptyset$, and $\mathcal{H}(p_i) \supset \mathcal{H}(p_{i+1})$ for each $1 \leq i < n$.*

Proof. Let s and t be processes such that $\mathcal{H}(s) \subset \mathcal{H}(t)$. We can safely assume that $(Act(s) \cup Act(t)) \cap \mathcal{A} = \emptyset$, because otherwise we can consider processes $r(s), r(t)$ for an appropriate renaming r (observe that $\mathcal{H}(r(s)) \subset \mathcal{H}(r(t))$ due to Lemma 3 and Definition 13). Let $\xi \in \mathcal{H}$ be a formula such that $t \models \xi$ and $s \not\models \xi$. Let $a_1, \cdots, a_n$ be fresh (unused) actions. The process p_i has (exactly) the following transitions: $p_i \xrightarrow{a_j} s$ for every $1 \leq j < i \leq n$, and $p_i \xrightarrow{a_j} t$ for every $1 \leq i \leq j \leq n$. We prove that $\mathcal{H}(p_i) \supset \mathcal{H}(p_{i+1})$ for each $1 \leq i < n$. First, note that $\langle a_i \rangle \xi \in \mathcal{H}$, $p_i \models \langle a_i \rangle \xi$, and $p_{i+1} \not\models \langle a_i \rangle \xi$. It remains to prove that for every $\varphi \in \mathcal{H}$ such that $p_{i+1} \models \varphi$ we also have $p_i \models \varphi$. The formula φ can be viewed as a boolean combination of formulae of the form $\langle a \rangle \psi$. We show that for each such $\langle a \rangle \psi$ we have that $p_{i+1} \models \langle a \rangle \psi \iff p_i \models \langle a \rangle \psi$, or $p_i \models \langle a \rangle \psi$ and $\langle a \rangle \psi$ is not within the scope of any negation in φ. It clearly suffices to conclude $p_i \models \varphi$. We distinguish two possibilities:

– $p_{i+1} \models \langle a \rangle \psi$. As $\psi \in \mathcal{H}$ and $\mathcal{H}(s) \subset \mathcal{H}(t)$, we also have $p_i \models \langle a \rangle \psi$ (see the construction of p_i above).
– $p_{i+1} \not\models \langle a \rangle \psi$. If $p_i \not\models \langle a \rangle \psi$, we are done immediately. If $p_i \models \langle a \rangle \psi$, then necessarily $a = a_i$; we obtain that $t \models \psi$ and $s \not\models \psi$. If the formula $\langle a \rangle \psi$ is within the scope of some negation in φ, we obtain $\neg\psi \in \mathcal{H}$. As $s \models \neg\psi$ and $t \not\models \neg\psi$, we have a contradiction with $\mathcal{H}(s) \subset \mathcal{H}(t)$. □

Lemma 8. *Let $\sim$ be a process equivalence having a closed modal characterization $\mathcal{H}$. Let s, t be processes such that for every $a \in Act$ we have $\bigcup_{s \xrightarrow{a} s'} \mathcal{H}(s') = \bigcup_{t \xrightarrow{a} t'} \mathcal{H}(t')$. Then $s \sim t$.*

Proof. We show that for every $\varphi \in \mathcal{H}$ we have $s \models \varphi$ iff $t \models \varphi$. By induction on the structure of φ.

– $\varphi \equiv \mathtt{tt}$. Immediate.
– $\varphi \equiv \psi \wedge \xi$. Suppose that $\psi \wedge \xi$ distinguishes between s and t. Then $\psi, \xi \in \mathcal{H}$ and at least one of those formulae must distinguish between s and t; we obtain a contradiction with induction hypotheses.
– $\varphi \equiv \neg\psi$. The same as above.
– $\varphi \equiv \langle a \rangle \psi$. Suppose, e.g., $s \models \langle a \rangle \psi$ and $t \not\models \langle a \rangle \psi$. Then $\psi \in \mathcal{H}$, $\psi \in \bigcup_{s \xrightarrow{a} s'} \mathcal{H}(s')$, and $\psi \notin \bigcup_{t \xrightarrow{a} t'} \mathcal{H}(t')$, a contradiction. □

Lemma 9. *Let $\sim$ be a process equivalence having a good modal characterization $\mathcal{H}$. Let $\mathcal{A}$ be a finite subset of Act, $k \in \mathbb{N}_0$. Let $T_1, T_2 \in Tree(\mathcal{A})_k$ s.t. there is a homomorphism f from T_2 to T_1 which preserves $\sim$. Then the Trees T_1, T_2 can be extended (by adding some new states and transitions) in such a way that the obtained transition systems T_1', T_2' satisfy the following:*

– *the homomorphism f can be extended to a homomorphism f' from T_2' to T_1' which also preserves $\sim$,*
– *for every H.M. formula φ s.t. $Act(\varphi) \subseteq \mathcal{A}$ we have $T_2' \models \varphi$ iff $T_2 \models \varphi$ and $T_1' \models \varphi$ iff $T_1 \models \varphi$,*
– *the 'old' states of T_1' (i.e., the ones which have not been added to T_1 during the extension procedure) are pairwise nonequivalent w.r.t. $\sim$.*

Proof. First we describe the extension of T_1 which yields the system T_1'. This extension is then 'propagated' back to T_2 via the homomorphism f—each state s of T_2 is extended in the same way as the state $f(s)$ of T_1. Finally, we show that the three requirements of our lemma are satisfied.

Let n be the number of states of T_1, and let m be the number of those states t of T_1 for which there is a state s of T_2 such that $f(s) = t$. Let $p_1, \ldots, p_n$ be processes over a finite $\mathcal{A}' \subseteq Act$ such that $\mathcal{H}(p_1) \supset \mathcal{H}(p_2) \supset \cdots \supset \mathcal{H}(p_n)$ and $\mathcal{A} \cap \mathcal{A}' = \emptyset$. Such processes must exist by Lemma 7. Now we take an arbitrary bijection b from the set of states of T_1 to $\{1, \ldots, n\}$ satisfying the following conditions:

– if $t = f(s)$ for some state s of T_2, then $b(t) \leq m$,
– if there is a (nonempty) path from t to t' in T_2, then $b(t) > b(t')$.

Now we add to T_1 all states of $p_1, \ldots, p_n$, and for each state t of T_1 and each transition $p_{b(s)} \xrightarrow{a} q$ we add the transition $t \xrightarrow{a} q$ (i.e., the state t has the same set of a-successors as $p_{b(s)}$ for every $a \in \mathcal{A}'$ after the modification). The described extension of T_1 is now 'propagated' to T_2 in the above indicated way, yielding the system T_2'.

As $\mathcal{A} \cap \mathcal{A}' = \emptyset$, the new transitions which have been added to T_1 and T_2 cannot influence the (in)validity of any H.M. formula φ s.t. $Act(\varphi) \subseteq \mathcal{A}$. Hence, the second requirement of our lemma is satisfied. Moreover, it is easy to see that the third requirement is satisfied as well, because the 'old' states of T_1' now satisfy pairwise different subsets of $\mathcal{H}_{\mathcal{A}'}$. It remains to show that the first requirement is also valid.

The homomorphism f' is defined as a 'natural' extension of f – it agrees with f on the 'old' states of T_1', and behaves like an identity function on the 'new' ones. Observe that if s is a 'new' state of T_2', then the transition systems $T_2'(s)$ and $T_1'(f'(s))$ are the same (isomorphic). Hence, f' trivially preserves $\sim$ on all 'new' states of T_2'. To prove that $s \sim f'(s)$ for every 'old' state s of T_2', we first need to show the following auxiliary lemma: let $s_1, \ldots, s_j$ be 'old' states of T_2', t an 'old' state of T_1' such that

- there is no state s of T_2' such that $f'(s) = t$,
- $\mathcal{H}_{\mathcal{A}}(t) \subseteq \bigcup_{i=1}^{j} \mathcal{H}_{\mathcal{A}}(s_i)$.

Then $\mathcal{H}(t) \subseteq \bigcup_{i=1}^{j} \mathcal{H}(s_i)$.

A proof of the auxiliary lemma: Let $\varphi \in \mathcal{H}$ such that $t \models \varphi$. We show that $s_i \models \varphi$ for some $1 \leq i \leq j$. First we construct a formula $\varphi' \in \mathcal{H}_{\mathcal{A}}$ from φ in the following way (recall the notions introduced in Definition 8): every occurence of a subformula $\langle a \rangle \psi$ in φ, $a \in \mathcal{A}'$, which is not within the scope of any $\langle b \rangle$-modality, where $b \in \mathcal{A}'$, is substituted by

- `tt` if $t \models \langle a \rangle \psi$ or there is some $t' \in \mathcal{R}_t(\langle a \rangle \psi)$ such that $t' \models \langle a \rangle \psi$,
- `ff` otherwise.

Clearly $\varphi' \in \mathcal{H}_{\mathcal{A}}$ (see Definition 13). We prove that $t \models \varphi'$, (i.e., $\varphi' \in \mathcal{H}_{\mathcal{A}}(t)$) by showing that the assumptions of Lemma 4 are satisfied for φ and the above defined substitution. Let $\langle a \rangle \psi$ be a formula whose occurence has been substituted in φ to obtain φ'. First, let us realize that every state of $\mathcal{R}_t(\langle a \rangle \psi)$ is an 'old' one, because $\mathcal{A}_d(\langle a \rangle \psi) \subseteq \mathcal{A}$ (see above). We can distinguish two possibilities:

- the occurence of $\langle a \rangle \psi$ has been substituted by `tt`. Then there are two subcases:
  - $t \models \langle a \rangle \psi$. Remember that each 'old' state q of T_1' has the same set of a-successors as $p_{b(q)}$ for every $a \in \mathcal{A}'$. Hence, $p_{b(t)} \models \langle a \rangle \psi$ because $t \models \langle a \rangle \psi$. Furthermore, for every $t' \in \mathcal{R}_t(\langle a \rangle \psi)$ we have $\mathcal{H}(p_{b(t)}) \subset \mathcal{H}(p_{b(t')})$ (see the definition of b above). Therefore, $p_{b(t')} \models \langle a \rangle \psi$ and thus we get $t' \models \langle a \rangle \psi$. In other words, for every $t' \in \mathcal{R}_t(\langle a \rangle \psi)$ we obtain $t' \models \mathtt{tt} \iff t' \models \langle a \rangle \psi$.
  - there is $t' \in \mathcal{R}_t(\langle a \rangle \psi)$ such that $t' \models \langle a \rangle \psi$. First, if $\langle a \rangle \psi$ is satisfied by *every* state of $\mathcal{R}_t(\langle a \rangle \psi)$, we are done immediately. Otherwise, there is $t'' \in \mathcal{R}_t(\langle a \rangle \psi)$ such that $t'' \not\models \langle a \rangle \psi$. Now it suffices to show that the occurrence of $\langle a \rangle \psi$ in φ cannot be within the scope of any negation (see the second condition of Lemma 4). Suppose the converse. As $\varphi \in \mathcal{H}$ and $\mathcal{H}$ is a good modal characterization, we know that both $\langle a \rangle \psi$ and $\neg \langle a \rangle \psi \in \mathcal{H}$. As the processes t'

and t'' have the same a-successors as the processes $p_{b(t')}$ and $p_{b(t'')}$, respectively, we obtain $p_{b(t')} \models \langle a \rangle \psi$ and $p_{b(t'')} \not\models \langle a \rangle \psi$, hence also $p_{b(t')} \not\models \neg\langle a \rangle \psi$ and $p_{b(t'')} \models \neg\langle a \rangle \psi$. Therefore, it cannot be that $\mathcal{H}(p_{b(t')}) \subset \mathcal{H}(p_{b(t'')})$ or $\mathcal{H}(p_{b(t'')}) \subset \mathcal{H}(p_{b(t')})$, a contradiction.

– the occurence of $\langle a \rangle \psi$ has been substituted by `ff`. Then $t' \not\models \langle a \rangle \psi$ for each $t' \in \mathcal{R}_t(\langle a \rangle \psi)$, and we are done immediately.

Now we know that $\varphi' \in \mathcal{H}_{\mathcal{A}}(t)$, hence there must be some s_i such that $s_i \models \varphi'$. We prove that $s_i \models \varphi$, again by applying Lemma 4 (observe that φ can be obtained from φ' by a substitution which is 'inverse' to the previously considered one). We show that the assumptions of Lemma 4 are satisfied also for φ' and the 'inverse' substitution, distinguishing two possibilities:

– a given occurence of `tt` is substituted 'back' to $\langle a \rangle \psi$. It means that we previously had $t \models \langle a \rangle \psi$ or $t' \models \langle a \rangle \psi$ for some $t' \in \mathcal{R}_t(\langle a \rangle \psi)$. As $\mathcal{H}(p_{b(f'(s))}) \supset \mathcal{H}(p_{b(v)})$ for every 'old' state s of T_2' and every 'old' state v of T_1' which is reachable from t (see the definition of b and the construction of T_2'), we can conclude that $\langle a \rangle \psi$ is satisfied by *each* 'old' state of T_2' (in particular, by all states of $\mathcal{R}_{s_i}(\mathtt{tt})$).
– a given occurence of `ff` is substituted 'back' to $\langle a \rangle \psi$. If $\langle a \rangle \psi$ is not satisfied by any state of $\mathcal{R}_{s_i}(\mathtt{ff})$, we done immediately. We show that if there is some $s' \in \mathcal{R}_{s_i}(\mathtt{ff})$ such that $s' \models \langle a \rangle \psi$, then the occurence of `ff` in φ' cannot be within the scope of any negation. Suppose the converse. Then there is an occurrence of $\langle a \rangle \psi$ in φ which is within the scope of some negation, hence $\neg\langle a \rangle \psi$ belong to $\mathcal{H}$. As $t \models \neg\langle a \rangle \psi$ and $\mathcal{H}(p_{b(t)}) \subset \mathcal{H}(p_{b(f'(s'))})$ (see above), we have $s' \models \neg\langle a \rangle \psi$, a contradiction.

Now we can continue with the main proof. We show that for each 'old' state s of T_2' we have that $s \sim f'(s)$. We proceed by induction on the depth of the subtree which is rooted by s in T_2 (denoted by d).

– $d = 0$. Then s is a leaf in T_2, hence the transition systems $T_2'(s)$ and $T_1'(f'(s))$ are isomorphic. Hence, we trivially have $s \sim f'(s)$.
– **Induction step:** We prove that $\bigcup_{s \xrightarrow{a} s'} \mathcal{H}(s') = \bigcup_{f'(s) \xrightarrow{a} t} \mathcal{H}(t)$ for each $a \in Act$ (hence $s \sim f'(s')$ by Lemma 8). If $a \in \mathcal{A}'$, the equality holds trivially because s and $f'(s)$ have the same set of a-successors. Now let $a \in \mathcal{A}$. By induction hypotheses we know that $\mathcal{H}(s') = \mathcal{H}(f'(s'))$ for each a-successor s' of s. To finish the proof, we need to show that for each a-successor t of $f'(s)$ for which there is no state q of T_2' with $f'(q) = t$ we have that $\mathcal{H}(t) \subseteq \bigcup_{s \xrightarrow{a} s'} \mathcal{H}(s')$. However, it can be easily achieved with a help of the auxiliary lemma which has been proved above; all we need is to show that $\mathcal{H}_{\mathcal{A}}(t) \subseteq \bigcup_{s \xrightarrow{a} s'} \mathcal{H}_{\mathcal{A}}(s')$. Suppose it is not the case, i.e., there is some $\vartheta \in \mathcal{H}_{\mathcal{A}}$ such that $t \models \vartheta$ and $s' \not\models \vartheta$ for each a-successor s' of s. Hence $\langle a \rangle \vartheta \in \mathcal{H}_{\mathcal{A}}$, $s \not\models \langle a \rangle \vartheta$, and $f(s) \models \langle a \rangle \vartheta$; it contradicts the fact that the homomorphism f preserves $\sim$. □

Theorem 4. *Let $\sim$ be a process equivalence having a good modal characterization $\mathcal{H}$. Then every formula which is preserved by $\sim$-characterizations is equivalent to some formula of $\mathcal{D}(\mathcal{H})$.*

Proof. Let φ be a formula preserved by $\sim$-characterizations, $k = depth(\varphi)$, $\mathcal{A} = Act(\varphi)$. For every $T \in Tree(\mathcal{A})_k$ we define the formula ψ_T by induction on the depth of T:

- if the depth of T is 0, then $\psi_T \equiv \mathtt{tt}$,
- if the depth of T is $j+1$, r is the root of T, and $r \xrightarrow{a_1} s_1, \cdots, r \xrightarrow{a_n} s_n$ are the outgoing arcs of r, then

$$\psi_T \equiv \bigwedge_{\substack{\varrho \in \mathcal{H}_{\mathcal{A}}^{j+1} \\ T \models \varrho}} \varrho \ \wedge \bigwedge_{\substack{\varrho \in \mathcal{H}_{\mathcal{A}}^{j+1} \\ T \not\models \varrho}} \neg\varrho \ \wedge \bigwedge_{i=1}^{n} \langle a_i \rangle \psi_{T(s_i)}$$

  where $T(s_i)$ is the sub-Tree of T rooted by s_i.

Let

$$\psi \equiv \bigvee_{\substack{T \in Tree(\mathcal{A})_k \\ T \models \varphi}} \psi_T$$

We prove that φ, ψ are equivalent by showing that they agree on every $T_1 \in Tree(\mathcal{A})_k$.

- Let $T_1 \in Tree(\mathcal{A})_k$ s.t. $T_1 \models \varphi$. As $T_1 \models \psi_{T_1}$, we immediately have $T_1 \models \psi$.
- Let $T_1 \in Tree(\mathcal{A})_k$ s.t. $T_1 \models \psi$. Then there is $T_2 \in Tree(\mathcal{A})_k$ with $T_2 \models \varphi$ and $T_1 \models \psi_{T_2}$. We need to prove that $T_1 \models \varphi$. Suppose the converse, i.e., $T_1 \models \neg\varphi$. Let r_1, r_2 be the roots of T_1, T_2, respectively. First we show that there is a homomorphism f from T_2 to T_1 s.t. for every node s of T_2 we have $f(s) \models \psi_{T(s)}$. The homomorphism f is defined by induction on the distance of s from r_2.
  - $s = r_2$. Then $f(r_2) = r_1$ (remember $T_1 \models \psi_{T_2}$).
  - s is the j^{th} successor of t where $t \xrightarrow{a_1} s_1, \cdots, t \xrightarrow{a_n} s_n$ are the outgoing arcs of t. The formula $\psi_{T(t)}$ looks as follows:

$$\psi_{T(t)} \equiv \bigwedge_{\substack{\varrho \in \mathcal{H}_{\mathcal{A}}^{k-d} \\ T(t) \models \varrho}} \varrho \ \wedge \bigwedge_{\substack{\varrho \in \mathcal{H}_{\mathcal{A}}^{k-d} \\ T(t) \not\models \varrho}} \neg\varrho \ \wedge \bigwedge_{i=1}^{n} \langle a_i \rangle \psi_{T(s_i)}$$

    where d is the distance of t from r_2. Let $f(t) = q$. As $q \models \psi_{T(t)}$ (by induction hypotheses), there is some $q \xrightarrow{a_j} q'$ s.t. $q' \models \psi_{T(s_j)}$. We put $f(s) = q'$.

  Observe that f also preserves $\sim$ because for every node s of T_2 we have that s and $f(s)$ satisfy exactly the same formulae of $\mathcal{H}_{\mathcal{A}}^{k-d}$ (d is the distance of s from r_2). Now we can apply Lemma 9—the Trees T_1, T_2 can be extended to transition systems T_1', T_2' in such a way that the 'old' states of T_1' are pairwise nonequivalent, φ is still valid (invalid) in r_2 (r_1), and the homomorphism f can be extended to a homomorphism f' which still preserves $\sim$. Let us define a transition system $\mathcal{T} = (S, \mathcal{A} \cup \mathcal{A}' \cup \{b\}, \rightarrow)$ where
  - S is a disjoint union of the sets of states of T_1' and T_2',
  - $\mathcal{A}'$ is the set of 'new' actions of T_1', T_2' (cf. the proof of Lemma 9), $b \notin \mathcal{A} \cup \mathcal{A}'$ is a fresh action,

- $\rightarrow$ contains all transitions of T_1' and T_2'; moreover, we also have $r_2 \stackrel{b}{\rightarrow} r_2$, $r_1 \stackrel{b}{\rightarrow} r_1$, and $r_2 \stackrel{b}{\rightarrow} r_1$.

The new b-transitions have been added just to make r_1 reachable from r_2. Observe that we still have $r_1 \sim r_2$, $r_1 \models \neg\varphi$, and $r_2 \models \varphi$. As T_2' can be 'embedded' into T_1' by f', the $\sim$-characterization of the process r_2 of $\mathcal{T}$ is the same (up to isomorphism) as the $\sim$-characterization of the process r_1 of T_1' with one additional arc $r_1 \stackrel{b}{\rightarrow} r_1$. As the 'old' states of T_1' (see Lemma 9) are pairwise non-equivalent w.r.t. $\sim$, and possible identification of the 'new' states of T_1' in the $\sim$-characterization of r_1 cannot influence (in)validity of any H.M. formula whose set of actions is contained in $\mathcal{A}$, we can conclude that φ is not satisfied by the process $[r_1]$ of $T_1'/\!\sim$. Hence, φ is not satisfied by the process $[r_1] = [r_2]$ of $\mathcal{T}/\!\sim$ either. As φ is satisfied by the process r_2 of $\mathcal{T}$, we can conclude that φ is not preserved by $\sim$-characterizations, and we have a contradiction. □

Theorem 3 and 4 together say that a H.M. property P is preserved (reflected) by $\sim$-characterizations, where $\sim$ is a process equivalence having a good modal characterization $\mathcal{H}$, iff P is equivalent to some diamond formula (or box formula – see Lemma 1) over $\mathcal{H}$.

4 Applications

Our abstract results can be applied to many concrete process equivalences which have been deeply studied in concurrency theory. A nice overview and comparison of such equivalences has been presented in [18]; existing equivalences (eleven in total) are ordered w.r.t. their coarseness and a kind of modal characterization is given for each of them (unfortunately, not a good one in the sense of Definition 13). However, those characterizations can be easily modified so that they become good (there are two exceptions – see below). Due to the lack of space, we present a good modal characterization only for *trace* equivalence.

Definition 14. *The set of* traces *of a process s, denoted $Tr(s)$, is defined by*

$$Tr(s) = \{w \in Act^* \mid \exists t \textit{ such that } s \stackrel{w}{\rightarrow} t\}$$

We say that s, t are trace equivalent, *written $s =_t t$, iff $Tr(s) = Tr(t)$.*

A good modal characterization $\mathcal{H}$ for trace equivalence is given by

$$\varphi \ ::= \ \mathtt{tt} \mid \mathtt{ff} \mid \langle a \rangle \varphi$$

where a ranges over Act. Let s, t be processes with transitions $s \stackrel{a}{\rightarrow} s'$, $t \stackrel{a}{\rightarrow} t'$, $t \stackrel{b}{\rightarrow} t''$ (and no other transitions). Obviously $\mathcal{H}(s) \subset \mathcal{H}(t)$.

To see that even an infinite-state process can have a very small $=_t$-representation and $=_t$-characterization, consider the process p of Fig. 1. The process q is a $=_t$-representation of p, and the process r is the $=_t$-characterization of p. According to our results,

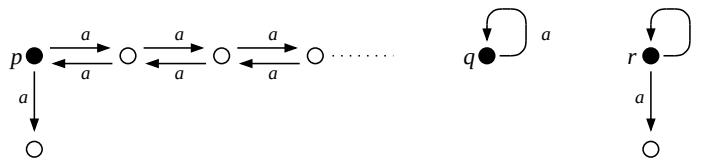


Fig. 1. An infinite-state process having finite $=_t$-representation and $=_t$-characterization

the formula $\langle a\rangle\neg\langle a\rangle\mathtt{tt}$ which is satisfied by p is not generally preserved by $=_t$-representations, but it is preserved by $=_t$-characterizations. Indeed, we have $q \not\models \langle a\rangle\neg\langle a\rangle\mathtt{tt}$, while $r \models \langle a\rangle\neg\langle a\rangle\mathtt{tt}$.

An interesting related problem is whether a given infinite-state state process has for a given $\sim$ any finite $\sim$-representation, and whether its $\sim$-characterization is finite. It is also known as the *regularity* and *strong regularity* problem (see also [13]). Some decidability results for various equivalences and various classes of infinite-state processes have already been established [3,12,9,10,14], but this area still contains a number of open problems.

The only equivalences of [18] which do not have a good modal characterization are bisimilarity [17] and completed trace equivalence. Bisimilarity is not a 'real' problem, in fact (only the last requirement of Definition 13 cannot be satisfied); a modal characterization of bisimilarity is formed by *all* H.M. formulae, and therefore *each* H.M. formula is trivially preserved and reflected by $\sim$-representations and $\sim$-characterizations. As for completed trace equivalence, the problem is that this equivalence requires a simple infinite conjuction, or a generalized $\langle\cdot\rangle$ modality (which can be phrased 'after any action'), which are not at disposal.

5 Related and Future Work

In the context of process theory, modal characterizations were introduced by Hennessy and Milner in their seminal paper [7]. The paper provides characterizations of bisimulation, simulation, and trace equivalence as full, conjunction-free, and negation-free Hennessy-Milner logic, respectively. The result stating that bisimulation equivalence is also characterized by the modal μ-calculus seems to be folklore. In [18], van Glabbeek introduces the equivalences of his hierarchy by means of sets of formulae, in a style close to modal characterizations.

In [11], Kaivola and Valmari determine weakest equivalences preserving certain fragments of linear time temporal logic. In [6], Goltz, Kuiper, and Penczek study the equivalences characterized by various logics in a partial order setting.

An interesting open problem is whether it is possible to give a similar classification for some richer (more expressive) logic. Also, we are not sufficiently acquainted with work on modal logic outside of computer science (or before computer science was born). Work on filtrations [4] or partial isomorphisms [5] should help us to simplify and streamline our proofs.

Acknowledgements

We thank anonymous CSL referees who greatly helped us to improve the presentation of the paper.

References

1. *Proceedings of CONCUR'92*, volume 630 of *LNCS*. Springer, 1992.
2. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 25 April 1994. Fundamental Study.
3. O. Burkart, D. Caucal, and B. Steffen. Bisimulation collapse and the process taxonomy. In *Proceedings of CONCUR'96*, volume 1119 of *LNCS*, pages 247–262. Springer, 1996.
4. B.F. Chellas. *Modal Logic—An Introduction*. Cambridge University Press, 1980.
5. J. Flum. First-order logic and its extensions. In *Proceedings of the International Summer Institute and Logic Colloquium*, volume 499 of *Lecture Notes in Mathematics*. Springer, 1975.
6. U. Goltz, R. Kuiper, and W. Penczek. Propositional temporal logics and equivalences. In *Proceedings of CONCUR'92* [1], pages 222–236.
7. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.
8. T. Henzinger. Hybrid automata with finite bisimulations. In *Proceedings of ICALP'95*, volume 944 of *LNCS*, pages 324–335. Springer, 1995.
9. P. Jančar and J. Esparza. Deciding finiteness of Petri nets up to bisimilarity. In *Proceedings of ICALP'96*, volume 1099 of *LNCS*, pages 478–489. Springer, 1996.
10. P. Jančar and F. Moller. Checking regular properties of Petri nets. In *Proceedings of CONCUR'95*, volume 962 of *LNCS*, pages 348–362. Springer, 1995.
11. R. Kaivola and A. Valmari. The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic. In *Proceedings of CONCUR'92* [1], pages 207–221.
12. A. Kučera. Regularity is decidable for normed PA processes in polynomial time. In *Proceedings of FST&TCS'96*, volume 1180 of *LNCS*, pages 111–122. Springer, 1996.
13. A. Kučera. On finite representations of infinite-state behaviours. *Information Processing Letters*, 70(1):23–30, 1999.
14. A. Kučera and R. Mayr. Simulation preorder on simple process algebras. In *Proceedings of ICALP'99*, LNCS. Springer, 1999. To appear.
15. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
16. R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, December 1987.
17. D.M.R. Park. Concurrency and automata on infinite sequences. In *Proceedings* 5^{th} *GI Conference*, volume 104 of *LNCS*, pages 167–183. Springer, 1981.
18. R.J. van Glabbeek. The linear time—branching time spectrum. In *Proceedings of CONCUR'90*, volume 458 of *LNCS*, pages 278–297. Springer, 1990.

A Truly Concurrent Semantics for a Simple Parallel Programming Language

Paul Gastin[1] and Michael Mislove[2] *

[1] LIAFA, Université Paris 7, 2, place Jussieu, F-75251 Paris Cedex 05, France
Paul.Gastin@liafa.jussieu.fr
[2] Department of Mathematics, Tulane University, New Orleans, LA 70118
mwm@math.tulane.edu

Abstract. This paper represents the beginning of a study aimed at devising semantic models for true concurrency that provide clear distinctions between concurrency, parallelism and choice. We present a simple programming language which includes (weakly) sequential composition, asynchronous and synchronous parallel composition, a restriction operator, and that supports recursion. We develop an operational and a denotational semantics for this language, and we obtain a theorem relating the behavior of a process as described by the transition system to the meaning of the process in the denotational model. This implies that the denotational model is adequate with respect to the operational model. Our denotational model is based on the resource traces of Gastin and Teodesiu, and since a single resource trace represents all possible executions of a concurrent process, we are able to model each term of our concurrent language by a single trace. Therefore we obtain a deterministic semantics for our language and we are able to model parallelism without introducing nondeterminism.

1 Introduction

The basis for building semantic models to support parallel composition of processes was laid out in the seminal work of Hennessy and Plotkin [5]. That work showed how *power domains* could be used to provide such models, but at the expense of introducing nondeterminism in the models, and also into the language. In this paper, we present an alternative approach to modeling parallelism that avoids nondeterminism. Our approach relies instead on *true concurrency* and more specifically on *trace theory* (as the area is called). Trace theory was introduced in the seminal work of Mazurkiewicz [6] in order to devise models for Petri nets, themselves a model of nondeterministic automata [8]. A great deal of research has been carried out in this area (cf. [3]), but programming semantics has not benefited from this research. The reasons are twofold: first, research in trace theory has focused on automata theory, for obvious reasons. The second

* Partial support provided by the National Science Foundation and the US Office of Naval Research

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 515–529, 1999.

reason is perhaps more telling: the traditional models which were developed for the concatenation operator of trace theory do not support a partial order relative to which concatentation is continuous (in the sense of Scott), nor are they metric spaces relative to which concatenation is a contraction. This means that the standard methods for modeling recursion are not available in these models, and so their use in modeling programming language constructs is limited.

Even so, domain-theoretic connections to trace theory abound in the literature, owing mainly to Winskel's insight that certain domains – *prime event structures* – provide models for concurrency [9]. There also is the work of Pratt [7] that introduced *pomsets*, which are models for trace theory. But none of this work has developed a *programming semantics* approach in which there is an abstract language based on the concatenation operator.

Recently Diekert, Gastin and Teodesiu have developed models for trace theory that are cpos relative to which the concatenation operator is Scott continuous [2, 4]. This opens the way for studying a trace-theoretic approach to concurrency, using these structures as the denotational models for such a language. This paper reports the first research results into this area. It presents a truly concurrent language which supports a number of interesting operators: the concatenation operator of trace theory, a restriction operator that confines processes to a certain set of *resources*, a synchronization operator that allows processes to execute independently, synchronizing on those actions which share common channels in the synchronization set, and that includes process variables and recursion.

The basis for our language is the concatenation operator from trace theory. In trace theory, independent actions can occur concurrently while dependent actions must be ordered. Therefore the concatenation of traces is only *weakly sequential*: it allows the beginning of the second process to occur independently of the end of the first process provided they are independent. We think this is a very attractive feature that corresponds to the automatic parallelization of processes. We also can model purely sequential composition by ending a process with a terminating action upon which all other actions depend.

The rest of the paper is organized as follows. In the Section 2 we provide some preliminary background on domain theory, on trace theory generally, and on the resource traces model of Gastin and Teodesiu [4]. These are the main ingredients of the semantic models for our language. The syntax of our language is the subject of the Section 3, and this is followed by Section 4 in which we explore the properties of the resource mapping that assigns to each action its set of resources; the results of this section are needed for both the operational and denotational semantics. Section 5 gives the transition system and the resulting operational semantics of our language, and also shows that the operational transition system is Church-Rosser. Section 6 is devoted to the denotational model, and the seventh section presents the main theorem relating the operational and the denotational semantics.

Due to space limitations, most proofs are omitted and will appear in a forthcoming full version of this paper.

2 Preliminaries

In this section we review some basic results from domain theory, and then some results from trace theory. A standard reference for domain theory is [1], and most of the results we cite can be found there. Similarly, for the theory of traces the reader is referred to [3]; specific results on resource traces can be found in [4].

2.1 Domain Theory

To begin, a *poset* is a partially ordered set, usually denoted P. The least element of P (if it exists) is denoted $\bot$, and a subset $D \subseteq P$ is *directed* if each finite subset $F \subseteq D$ has an upper bound in D. Note that since $F = \emptyset$ is a possibility, a directed subset must be non-empty. A *(directed) complete partial order (dcpo)* is a poset P in which each directed set has a least upper bound. If P also has a least element, then it is called a *cpo*. If P and Q are posets and $f\colon P \to Q$ is a monotone map, then f is *(Scott) continuous* if f preserves sups of directed sets: if $D \subseteq P$ is directed and $x = \vee D \in P$ exists, then $\vee f(D) \in Q$ exists and $f(\vee D) = \vee f(D)$.

If P is a dcpo, the element $k \in P$ is *compact* if, for each directed subset $D \subseteq P$, if $k \sqsubseteq \vee D$, then $(\exists d \in D)\ k \sqsubseteq d$. The set of compact elements of P is denoted $K(P)$, and for an element $x \in P$, $K(x) = K(P) \cap {\downarrow}x$, where ${\downarrow}x = \{y \in P \mid y \sqsubseteq x\}$. P is *algebraic* if $K(x)$ is directed and $x = \vee K(x)$ for each $x \in P$.

2.2 Resource Traces

We start with a finite alphabet Σ, a finite set $\mathcal{R}$ of *resources*, and a mapping $\mathrm{res}\colon \Sigma \to \mathcal{P}(\mathcal{R})$ satisfying $\mathrm{res}(a) \neq \emptyset$ for all $a \in \Sigma$. We can then define a dependence relation on Σ by $(a, b) \in D$ iff $\mathrm{res}(a) \cap \mathrm{res}(b) \neq \emptyset$. The dependence relation is reflexive and symmetric and its complement $I = (\Sigma \times \Sigma) \setminus D$ is called the *independence relation* on Σ.

A *real trace* t over (Σ, D) is the isomorphism class of a labeled, directed graph $t = [V, E, \lambda]$, where V is a countable set of *events*, $E \subseteq V \times V$ is the *synchronization relation* on V, and $\lambda\colon V \to \Sigma$ is a node-labeling satisfying

- $\forall p \in V$, ${\downarrow}p = \{q \in V \mid (q, p) \in E^*\}$ is finite,
- $\forall p, q \in V$, $(\lambda(p), \lambda(q)) \in D \ \Leftrightarrow\ (p, q) \in E \cup E^{-1} \cup \{(p, p) \mid p \in V\}$

The trace t is finite if V is finite and the length of t is $|t| = |V|$. The set of real traces over (Σ, D) is denoted by $\mathbb{R}(\Sigma, D)$, and the set of finite traces by $\mathbb{M}(\Sigma, D)$.

The alphabet of a real trace t is the set $\mathrm{Alph}(t) = \lambda(V)$ of letters which occur in t. We also define the *alphabet at infinity* of t as the set $\mathrm{alphinf}(t)$ of letters which occur infinitely often in t. We extend the resource mapping to real traces by defining $\mathrm{res}(t) = \mathrm{res}(\mathrm{Alph}(t))$. The *resources at infinity* of t is the set $\mathrm{resinf}(t) = \mathrm{res}(\mathrm{alphinf}(t))$. A real trace is finite iff $\mathrm{alphinf}(t) = \mathrm{resinf}(t) = \emptyset$.

A partial concatenation operation is defined on real traces as follows: Let $t_1 = [V_1, E_1, \lambda_1]$ and $t_2 = [V_2, E_2, \lambda_2]$ be real traces such that $\mathrm{resinf}(t_1) \cap \mathrm{res}(t_2) = \emptyset$,

then the concatenation of t_1 and t_2 is the real trace $t_1 \cdot t_2 = [V, E, \lambda]$ obtained by taking the disjoint union of t_1 and t_2 and adding necessary edges from t_1 to t_2, i.e., $V = V_1 \dot\cup V_2$, $\lambda = \lambda_1 \dot\cup \lambda_2$, and $E = E_1 \dot\cup E_2 \dot\cup (V_1 \times V_2 \cap \lambda^{-1}(D))$. In this representation, the *empty trace* $1 = [\emptyset, \emptyset, \emptyset]$ is the identity.

The monoid of finite traces $(\mathbb{M}(\Sigma, D), \cdot)$ is isomorphic to the quotient monoid $\Sigma^*/\equiv$ of the free monoid Σ^* of finite words over Σ, modulo the least congruence generated by $\{(ab, ba) \mid (a, b) \in I\}$.

The prefix ordering is defined on real traces by $r \leq t$ iff there exists a real trace s such that $t = r \cdot s$. When $r \leq t$, then the trace s satisfying $t = r \cdot s$ is unique and is denoted by $r^{-1}t$. $(\mathbb{R}(\Sigma, D), \leq)$ is a dcpo with the empty trace as least element. The compact elements of $(\mathbb{R}(\Sigma, D), \leq)$ are exactly the finite traces.

Just as in the case of the concatenation of words, the concatenation operation on $\mathbb{M}(\Sigma, D)$ is not monotone with respect to the prefix order. It is for this reason that $\mathbb{M}(\Sigma, D)$ cannot be completed into a dcpo on which concatenation is continuous, and so it is not clear how to use traces as a basis for a domain-theoretic model for the concatenation operator of trace theory.

This shortcoming was overcome by the work of Diekert, Gastin and Teodesiu [2,4]. In this paper, we will use the latter work as a basis for the denotational models for our language. The *resource trace domain* over $(\Sigma, \mathcal{R}, \mathrm{res})$ is then defined to be the family

$$\mathbb{F}(\Sigma, D) = \{(r, R) \mid r \in \mathbb{R}(\Sigma, D),\ R \subseteq \mathcal{R} \text{ and } \mathrm{resinf}(r) \subseteq R\}.$$

For a resource trace $x = (r, R) \in \mathbb{F}(\Sigma, D)$, we call $\mathrm{Re}(x) = r$ the *real part* of x and $\mathrm{Im}(x) = R$ the *imaginary part* of x. Most resource traces are meant to describe approximations of actual processes. The real part describes what has already been observed from the process and the imaginary part is the set of resources allocated to the process for its completion. The set of resource traces $\mathbb{F}(\Sigma, D)$ is thus endowed with a partial order called the *approximation order:*

$$(r, R) \sqsubseteq (s, S) \ \Leftrightarrow\ r \leq s \text{ and } R \supseteq S \cup \mathrm{res}(r^{-1}s).$$

We also endow $\mathbb{F}(\Sigma, D)$ with the concatenation operation

$$(r, R) \cdot (s, S) = (r \cdot \mu_R(s), R \cup S \cup \sigma_R(s)),$$

where $\mu_R(s)$ is the largest prefix u of s satisfying $\mathrm{res}(u) \cap R = \emptyset$ and $\sigma_R(s) = \mathrm{res}(\mu_R(s)^{-1}s)$. Intuitively, the product $(r, R) \cdot (s, S)$ is the best approximation we can compute for the composition of two processes if we only know their approximations (r, R) and (s, S).

It turns out that $(\mathbb{F}, \sqsubseteq)$ is a dcpo with least element $(1, \mathcal{R})$, where 1 is the empty trace. Moreover, the concatenation operator defined above is continuous with respect to this order. In other words, $(\mathbb{F}, \sqsubseteq, \cdot)$ is a continuous algebra in the sense of domain theory. The dcpo $(\mathbb{F}, \sqsubseteq)$ is also algebraic and a resource trace $x = (r, R)$ is compact if and only if it is finite, that is, iff its real part r is finite.

We close this section with a simple result about the resource mapping.

Proposition 1. *The resource mapping* $\mathrm{res}\colon \Sigma \to \mathcal{P}(\mathcal{R})$ *extends to a continuous mapping* $\mathrm{res}\colon \mathbb{F}(\Sigma, D) \to (\mathcal{P}(\mathcal{R}), \supseteq)$ *defined by* $\mathrm{res}(r, R) = \mathrm{res}(r) \cup R$. □

2.3 Alphabetic Mappings

The results presented in this section are new. They are useful for the denotational semantics of our parallel composition operator.

Let $\text{res} : \Sigma \to \mathcal{P}(\mathcal{R})$ and $\text{res}' : \Sigma' \to \mathcal{P}(\mathcal{R})$ be two resource maps and let D and D' be the associated dependence relations over the alphabets Σ and Σ'.

Let $\varphi : \Sigma \to \Sigma' \cup \{1\}$ be an alphabetic mapping such that $\text{res}'(\varphi(a)) \subseteq \text{res}(a)$ for all $a \in \Sigma$. We extend φ to real traces: If $r = [V, E, \lambda] \in \mathbb{R}(\Sigma, D)$, then we define $\varphi(r) = [V', E', \lambda']$ by $V' = \{e \in V \mid \varphi \circ \lambda(e) \neq 1\}$, $\lambda' = \varphi \circ \lambda$ and $E' = E \cap {\lambda'}^{-1}(D') = \{(e, f) \in E \mid \lambda'(e)\, D'\, \lambda'(f)\}$. The mapping φ is said to be *non-erasing* if $\varphi(a) \neq 1$ for all $a \in \Sigma$.

Proposition 2.

1. $\varphi : (\mathbb{R}(\Sigma, D), \cdot) \to (\mathbb{R}(\Sigma', D'), \cdot)$ *is a morphism.*
2. $\varphi : (\mathbb{R}(\Sigma, D), \leq) \to (\mathbb{R}(\Sigma', D'), \leq)$ *is continuous.* □

We now extend φ to a mapping over resource traces of $\mathbb{F}(\Sigma, D)$ simply by defining $\varphi(r, R) = (\varphi(r), R)$. Since $\text{res}'(\varphi(a)) \subseteq \text{res}(a)$ for all $a \in \Sigma$, we deduce that $\text{resinf}'(\varphi(r)) \subseteq \text{resinf}(r) \subseteq R$ and so $(\varphi(r), R)$ is a resource trace over Σ'. Hence, $\varphi : \mathbb{F}(\Sigma, D) \to \mathbb{F}(\Sigma', D')$ is well defined.

Proposition 3.

1. $\varphi : (\mathbb{F}(\Sigma, D), \sqsubseteq) \to (\mathbb{F}(\Sigma', D'), \sqsubseteq)$ *is continuous.*
2. *If* $\text{res}'(\varphi(a)) = \text{res}(a)\ (\forall a \in \Sigma)$*, then* $\varphi : (\mathbb{F}(\Sigma, D), \cdot) \to (\mathbb{F}(\Sigma', D'), \cdot)$ *is a non-erasing morphism.* □

3 The Language

In this section we introduce a simple parallel programming language. We begin once again with a finite set Σ of atomic actions, a finite set $\mathcal{R}$ of resources, and a mapping $\text{res}\colon \Sigma \to \mathcal{P}(\mathcal{R})$ which assigns to each $a \in \Sigma$ a *non-empty* set of resources. We view $\text{res}(a)$ as the set of resources – memory, ports, etc. – that the action a needs in order to execute. Two actions $a, b \in \Sigma$ may be executed concurrently if and only if they are independent – i.e. iff they do not share any resource. We define the BNF-like syntax of the language $\mathcal{L}$ we study as

$$p ::= \text{STOP} \mid a \mid p \circ p \mid p|_R \mid p \underset{C}{\|} p \mid x \mid \text{rec}\, x.p$$

where

- STOP is the process capable of no actions but claiming all resources; it is full deadlock.
- $a \in \Sigma$ denotes the process which can execute the action a and then terminate normally.
- $p \circ q$ denotes the weak sequential composition of the two argument processes with the understanding that independent actions commute with one another: $a \circ b = b \circ a$ if $a, b \in I$. We call $\circ$ *weak sequential composition* because it enforces sequential composition of those actions which are dependent, while allowing those which are independent of one another to execute concurrently.

- $p|_R$ denotes the process p with all resources restricted to the subset $R \subseteq \mathcal{R}$. Only those actions a from p can execute for which $\mathrm{res}(a) \subseteq R$; all other actions are disabled.
- $p \underset{C}{\|} q$ denotes the parallel composition of the component processes, synchronizing on all actions a which satisfy $\mathrm{res}(a) \cap C \neq \emptyset$, where $C \subseteq \mathcal{R}$. Those actions from either component which do not have any resources in common with any of the actions in the other component nor any resources lying in C are called *local* and can execute independently. Since our semantics is deterministic, this process can only make progress as long as there are no actions from either component that use resources that some action from the other component also uses, except in the case of synchronization actions. If this condition is violated, the process deadlocks.
- $x \in V$ is a process variable.
- $\mathrm{rec}\, x.p$ denotes recursion of the process body p in the variable x.

One of the principal impetuses for our work is the desire to understand the differences between parallel composition, choice and nondeterminism. Historically, nondeterministic choice arose as a convenient means with which to model parallel composition, namely, as the set of possible interleavings of the actions of each component. We avoid nondeterminism, and in fact our language is deterministic. But we still support parallel composition – that in which the actions of each component are independent.

A parallel composition involves choice whenever there is a competition between conflicting events. Since we use a truly concurrent semantic domain, our events are not necessarily conflicting and we can consider a very natural and important form of cooperative parallel composition which does not require choice or nondeterminism. Each process consists of local events which occur independently of the other process and of synchronization events which are executed in matching pairs. These synchronization events may introduce conflict when the two processes offer non-matching synchronization events. Since nondeterministic choice is unavailable, conflicting events result in deadlock in our parallel composition. Note that this situation does not occur in a cooperative parallel composition, e.g. in a parallel sorting algorithm.

We view the BNF-like syntax given above as the signature, $\Omega = \cup_n \Omega_n$, of a single sorted universal algebra, where the index n denotes the arity of the operators in the subset Ω_n. In our case, we have

Nullary operators: $\Omega_0 = \{STOP\} \cup \Sigma \cup V$,
Unary operators: $\Omega_1 = \{-|_R \mid R \subseteq \mathcal{R}\} \cup \{\mathrm{rec}\, x.- \mid x \in V\}$,
Binary operators: $\Omega_2 = \{\circ\} \cup \{\underset{C}{\|} \mid C \subseteq \mathcal{R}\}$, and
$\Omega_n = \emptyset$ for all other n;

then $\mathcal{L}$ is the *initial Ω-algebra*. This means that, given any Ω-algebra A, there is a unique Ω-algebra homomorphism $\phi_A \colon \mathcal{L} \to A$, i.e., a unique compositional mapping from $\mathcal{L}$ to A.

4 The Resource Mapping

In this section we define the resources which may be used by a process $p \in \mathcal{L}$. This is crucial for defining the operational semantics of weak sequential composition and of parallel composition. We extend the mapping $\mathrm{res}\colon \Sigma \to \mathcal{P}(\mathcal{R})$ to the full language $\mathcal{L}$ with variables and recursion. In order to define the resource set associated with a process with free variables, we use a *resource environment*, a mapping $\sigma : V \to \mathcal{P}(\mathcal{R})$ assigning a resource set to each variable. Any resource environment $\sigma \in \mathcal{P}(\mathcal{R})^V$ can be *locally overridden* in its value at x:

$$\sigma[x \mapsto R](y) = R, \text{ if } y = x, \quad \text{and} \quad \sigma[x \mapsto R](y) = \sigma(y), \text{ otherwise,}$$

where $R \in \mathcal{P}(\mathcal{R})$ is any resource set we wish to assign at x.

Now, we define inductively the resources of a process $p \in \mathcal{L}$ in the resource environment $\sigma \in \mathcal{P}(\mathcal{R})^V$ by:

- $\mathrm{res}(\mathrm{STOP}, \sigma) = \mathcal{R}$,
- $\mathrm{res}(a, \sigma) = \mathrm{res}(a)$ for all $a \in \Sigma$,
- $\mathrm{res}(p|_R, \sigma) = \mathrm{res}(p, \sigma) \cap R$ for all $R \subseteq \mathcal{R}$,
- $\mathrm{res}(p \circ q, \sigma) = \mathrm{res}(p, \sigma) \cup \mathrm{res}(q, \sigma)$,
- $\mathrm{res}(p \underset{C}{\parallel} q, \sigma) = \mathrm{res}(p, \sigma) \cup \mathrm{res}(q, \sigma)$,
- $\mathrm{res}(x, \sigma) = \sigma(x)$ for all $x \in V$,
- $\mathrm{res}(\mathrm{rec}\, x.p, \sigma) = \mathrm{res}(p, \sigma[x \mapsto \emptyset])$.

For instance, we have $\mathrm{res}(\mathrm{STOP}|_R, \sigma) = R$, $\mathrm{res}(\mathrm{rec}\, x.(a \circ x \circ b, \sigma)) = \mathrm{res}(a) \cup \mathrm{res}(b)$ and $\mathrm{res}((\mathrm{rec}\, x.(x \circ a)) \underset{C}{\parallel} (\mathrm{rec}\, y.(b \circ y))) = \mathrm{res}(a) \cup \mathrm{res}(b)$.

It is easy to see that the map $\mathrm{res}(p, -) : (\mathcal{P}(\mathcal{R}), \supseteq)^V \to (\mathcal{P}(\mathcal{R}), \supseteq)$ is continuous for each process $p \in \mathcal{L}$. A crucial result concerning the resource map states that the definition of recursion is actually a fixed point.

Proposition 4. *Let $p \in \mathcal{L}$ be a process and $\sigma \in \mathcal{P}(\mathcal{R})^V$ be a resource environment. Then,* $\mathrm{res}(\mathrm{rec}\, x.p, \sigma)$ *is the greatest fixed point of the mapping*

$$\mathrm{res}(p, \sigma[x \mapsto -]) : (\mathcal{P}(\mathcal{R}), \supseteq) \to (\mathcal{P}(\mathcal{R}), \supseteq). \qquad \square$$

In fact, we can endow the set of continuous maps $[\mathcal{P}(\mathcal{R})^V \to \mathcal{P}(\mathcal{R})]$ with a structure of a continuous Ω-algebra. The constants STOP and a ($a \in \Sigma$) are interpreted as constant maps $\sigma \mapsto \mathcal{R}$ and $\sigma \mapsto \mathrm{res}(a)$, the process x is interpreted as the projection $\sigma \mapsto \sigma(x)$, restriction $|_R$ is intersection with R, the two compositions $\circ$ and $\underset{C}{\parallel}$ are union, and finally, recursion $\mathrm{rec}\, x$ is the greatest fixed point: it maps $f \in [\mathcal{P}(\mathcal{R})^V \to \mathcal{P}(\mathcal{R})]$ to the mapping $\sigma \mapsto (\nu R.f(\sigma[x \mapsto R]))$. With this view, the mapping $p \mapsto \mathrm{res}(p, -)$ is the unique Ω-algebra map from $\mathcal{L}$ to $[\mathcal{P}(\mathcal{R})^V \to \mathcal{P}(\mathcal{R})]$.

We use $p[q/x]$ to denote the result of substituting q for the variable x in p. We now show how to compute the resource map at the process $p[q/x]$ in terms of the resource map at p.

Lemma 1. *Let $p, q \in \mathcal{L}$ be two processes and $\sigma \in \mathcal{P}(\mathcal{R})^V$ be a resource environment. Then*

$$\mathrm{res}(p[q/x], \sigma) = \mathrm{res}(p, \sigma[x \mapsto \mathrm{res}(q, \sigma)]). \qquad \square$$

5 Operational Semantics

In this section we present an operational semantics for all terms $p \in \mathcal{L}$, even those with free variables. This is necessitated by our use of something other than the usual least fixed point semantics of recursion that domain theory offers. The reason for this will be clarified later on – for now, we confine our discussion to presenting the transition rules for our language, and deriving results about the resulting behavior of terms from $\mathcal{L}$ under these rules. The key is to use environments. We rely on the mappings $\sigma\colon V \to \mathcal{P}(\mathcal{R})$ to aid us, and so our transition rules tell us what next steps are possible for a term *in a given environment* σ.

We must make an additional assumption to define our transition rules. We are interested in supporting synchronization over a set $C \subseteq \mathcal{R}$ which we view as the channels over which synchronization can occur. We therefore assume that the alphabet Σ has a synchronization operation $\|\colon \Sigma \times \Sigma \to \Sigma$ that satisfies $\mathrm{res}(a_1 \| a_2) = \mathrm{res}(a_1) \cup \mathrm{res}(a_2)$ for all $(a_1, a_2) \in \Sigma^2$. Moreover, for $p_1, p_2 \in \mathcal{L}$, $\sigma \in \mathcal{P}(\mathcal{R})^V$ and $C \subseteq \mathcal{R}$ we define the set $\mathrm{Sync}_{C,\sigma}(p_1, p_2)$ of pairs $(a_1, a_2) \in \Sigma^2$ such that

$$\mathrm{res}(a_1) \cap \mathrm{res}(p_2, \sigma) = \mathrm{res}(a_2) \cap \mathrm{res}(p_1, \sigma) = \mathrm{res}(a_1) \cap C = \mathrm{res}(a_2) \cap C \neq \emptyset.$$

$\mathrm{Sync}_{C,\sigma}(p_1, p_2)$ consists of all pairs which may be synchronized in $p_1 \underset{C}{\|} p_2$. We present the transition rules which are the basis for the operational semantics for our language $\mathcal{L}$ in Table 1 below. We denote by SKIP the process $\mathrm{STOP}|_\emptyset$.

We need a number of results about the rules in Table 1 before we can define the operational behaviour of a term $p \in \mathcal{L}$. Some of the results presented here are easier to prove once we have defined the denotational semantics of our language in the following section, but we have chosen to state the results now to improve the readability of the presentation.

Proposition 5. *In the following, $p, p', p'' \in \mathcal{L}$ are processes, $\sigma \in \mathcal{P}(\mathcal{R})^V$ is a syntactic environment, $x \in V$, and $s \in \Sigma^*$.*

1. *$p \xrightarrow[\sigma]{s} p'$ implies $\mathrm{res}(p, \sigma) = \mathrm{res}(p', \sigma) \cup \mathrm{res}(s)$.*
2. *$p \xrightarrow[\sigma]{a} p'$ and $p \xrightarrow[\sigma]{a} p''$ imply $p' = p''$.*
3. *If $a \neq b$ then $p \xrightarrow[\sigma]{a} p'$ and $p \xrightarrow[\sigma]{b} p''$ imply $a\, I\, b$ and $\exists p''' \in \mathcal{L}$ with $p' \xrightarrow[\sigma]{b} p'''$ and $p'' \xrightarrow[\sigma]{a} p'''$.*
4. *If $a I b$ then $p \xrightarrow[\sigma]{a} p'$ and $p' \xrightarrow[\sigma]{b} p''$ imply $\exists p''' \in \mathcal{L}$ with $p \xrightarrow[\sigma]{b} p'''$ and $p''' \xrightarrow[\sigma]{a} p''$.*

Proposition 5 (2) means that our transition system is deterministic. Adding Proposition 5 (3), we know that it is strongly locally confluent, whence Church-Rosser. Since we want a truly concurrent semantics, it should be possible for a process to execute independent events concurrently – i.e., independently. This is reflected by Proposition 5 (4) in our transition system. From this we derive by induction

(1) $$\frac{}{a \xrightarrow[\sigma]{a} \text{SKIP}}$$

(2a) $$\frac{p_1 \xrightarrow[\sigma]{a} p_1'}{p_1 \circ p_2 \xrightarrow[\sigma]{a} p_1' \circ p_2}$$

(2b) $$\frac{p_2 \xrightarrow[\sigma]{a} p_2',\ \text{res}(a) \cap \text{res}(p_1, \sigma) = \emptyset}{p_1 \circ p_2 \xrightarrow[\sigma]{a} p_1 \circ p_2'}$$

(3) $$\frac{p \xrightarrow[\sigma]{a} p',\ \text{res}(a) \subseteq R}{p|_R \xrightarrow[\sigma]{a} p'|_R}$$

(4a) $$\frac{p_1 \xrightarrow[\sigma]{a} p_1',\ \text{res}(a) \cap (\text{res}(p_2, \sigma) \cup C) = \emptyset}{p_1 \underset{C}{\|} p_2 \xrightarrow[\sigma]{a} p_1' \underset{C}{\|} p_2}$$

(4b) $$\frac{p_2 \xrightarrow[\sigma]{a} p_2',\ \text{res}(a) \cap (\text{res}(p_1, \sigma) \cup C) = \emptyset}{p_1 \underset{C}{\|} p_2 \xrightarrow[\sigma]{a} p_1 \underset{C}{\|} p_2'}$$

(4c) $$\frac{p_1 \xrightarrow[\sigma]{a_1} p_1',\ p_2 \xrightarrow[\sigma]{a_2} p_2',\ (a_1, a_2) \in \text{Sync}_{C,\sigma}(p_1, p_2)}{p_1 \underset{C}{\|} p_2 \xrightarrow[\sigma]{a_1 \| a_2} p_1' \underset{C}{\|} p_2'}$$

(5) $$\frac{p \xrightarrow[\sigma']{a} p',\ \sigma' = \sigma[x \mapsto \text{res}(\text{rec}\, x.p, \sigma)]}{\text{rec}\, x.p \xrightarrow[\sigma]{a} p'[\text{rec}\, x.p/x]}$$

Table 1. The Transition Rules for $\mathcal{L}$

Corollary 1. *Let $u, v \in \Sigma^*$ with $u \equiv v$. Then $p \xrightarrow[\sigma]{u} q$ iff $p \xrightarrow[\sigma]{v} q$. Hence $p \xrightarrow[\sigma]{s} q$ is well-defined for finite traces $s \in \mathbb{M}(\Sigma, D)$.* □

In an interleaving semantics, the possible operational behaviors of a process p in the environment $\sigma \in \mathcal{P}(\mathcal{R})$ would consist of the set

$$X_{\Sigma^*}(p, \sigma) = \{u \in \Sigma^* \mid \exists q \in \mathcal{L}, p \xrightarrow[\sigma]{u} q\}.$$

Thanks to Corollary 1, we actually can define the possible concurrent behaviors as

$$X_{\mathbb{M}}(p, \sigma) = \{t \in \mathbb{M}(\Sigma, D) \mid \exists q \in \mathcal{L}, p \xrightarrow[\sigma]{t} q\}.$$

But, knowing only a possible real (finite) trace that can be executed does not allow us to know how the process can be continued or composed with another process. Hence we need to bring resources into the picture, and so we define the resource trace behaviors by

$$X_{\mathbb{F}}(p, \sigma) = \{(s, \text{res}(q, \sigma)) \in \mathbb{F} \mid \exists q \in \mathcal{L}, p \xrightarrow[\sigma]{s} q\}.$$

The meaning is that $(s, S) \in X_{\mathbb{F}}(p, \sigma)$ if p can concurrently execute the trace s and then still claim the resources in S.

Actually, we can prove that the set $X_{\mathbb{F}}(p,\sigma)$ is directed. The interpretation is that p has a unique maximal behavior in the environment σ which is the least upper bound of $X_{\mathbb{F}}(p,\sigma)$: $B_{\mathbb{F}}(p,\sigma) = \sqcup X_{\mathbb{F}}(p,\sigma)$. This is exactly what tells us that our semantics of parallelism does not involve nondeterministic choice.

6 Denotational Semantics

The denotational semantics for our language takes its values in the family $[\mathbb{F}^V \to \mathbb{F}]$ of continuous maps from $\mathbb{F}^V$ to the underlying domain $\mathbb{F} = \mathbb{F}(\Sigma, D)$ of resource traces. As was the case with the resources model of Section 4, the semantics of a *finitary process* (i.e., one without variables) $p \in \mathcal{L}$ is a constant map, which means it is a resource trace. More generally, the semantics of any closed process $p \in \mathcal{L}$ is simply a resource trace. But, as in the case of the semantics based on the mapping $\mathrm{res}\colon \Sigma \to \mathcal{P}(\mathcal{R})$, in order to give the semantics of recursion, we also have to consider terms with free variables.

We begin by defining the family of *semantic environments* to be the mappings $\sigma\colon V \to \mathbb{F}$, and we endow this with the domain structure from the target domain $\mathbb{F}$, regarding $\mathbb{F}^V$ as a product on V-copies of $\mathbb{F}$. The semantics of an arbitrary process $p \in \mathcal{L}$ is a continuous map from $\mathbb{F}^V$ to $\mathbb{F}$, and the semantics of a recursive process $\mathrm{rec}\, x.p$ is obtained using a fixed point of the semantic map associated with p.

We obtain a compositional semantics by defining the structure of a continuous Ω-algebra on $[\mathbb{F}^V \to \mathbb{F}]$. We define the interpretations of constants and variables in $[\mathbb{F}^V \to \mathbb{F}]$ directly, but for the other operators except recursion, we instead define their interpretations on $\mathbb{F}$, and then extend them to $[\mathbb{F}^V \to \mathbb{F}]$ in a pointwise fashion. This approach induces on $[\mathbb{F}^V \to \mathbb{F}]$ the structure of a continuous Ω-algebra (cf. [1]). Recursion needs a special treatment since we do not use the classical least fixed point semantics as explained in Section 6.5.

6.1 Constants and Variables

The denotational semantics of constants and of variables are defined by the maps:

$$\begin{aligned} [\![\mathrm{STOP}]\!] \in [\mathbb{F}^V \to \mathbb{F}] \quad &\text{by} \quad [\![\mathrm{STOP}]\!](\sigma) = (1, \mathcal{R}) \\ [\![a]\!] \in [\mathbb{F}^V \to \mathbb{F}] \quad &\text{by} \quad [\![a]\!](\sigma) = (a, \emptyset) \\ [\![x]\!] \in [\mathbb{F}^V \to \mathbb{F}] \quad &\text{by} \quad [\![x]\!](\sigma) = \sigma(x) \end{aligned}$$

The first two clearly are continuous, since they are constant maps. The last mapping amounts to projection of the element $\sigma \in \mathbb{F}^V$ onto its x-component, and since we endow $\mathbb{F}^V$ with the product topology, this mapping also is continuous.

6.2 Weak Sequential Composition

We define the semantics of weak sequential composition using the following result about the concatenation of resource traces.

Proposition 6 ([4]). *Concatenation over resource traces is a continuous operation. Moreover,* $\mathrm{res}(x_1 \cdot x_2) = \mathrm{res}(x_1) \cup \mathrm{res}(x_2)$ *for all* $(x_1, x_2) \in \mathbb{F}^2$. □

The denotational semantics of $\circ$ is then defined by:

$$\circ\colon [\mathbb{F}^V \to \mathbb{F}]^2 \to [\mathbb{F}^V \to \mathbb{F}] \quad \text{by} \quad (f_1 \circ f_2)(\sigma) = f_1(\sigma) \cdot f_2(\sigma).$$

6.3 Restriction

For restriction and parallel composition, we need to define new operations on traces that have not been introduced so far. We start with restriction, which we obtain as the composition of two continuous maps. Let $R \subseteq \mathcal{R}$ be a fixed resource set. We first introduce

$$\mathbb{F}_R = \{x \in \mathbb{F} \mid \mathrm{res}(\mathrm{Re}(x)) \subseteq R\},$$

the set of resource traces whose real parts use resources from R only. Note that if some set $X \subseteq \mathbb{F}_R$ is pairwise consistent in $\mathbb{F}$, then its sup in $\mathbb{F}$ exists and actually belongs to $\mathbb{F}_R$. Therefore, $\mathbb{F}_R$ is also a consistently complete domain. Recall also that $\uparrow x = \{y \in \mathbb{F} \mid x \sqsubseteq y\}$ denotes the upper set of $x \in \mathbb{F}$. Now we define

$$f\colon \mathbb{F} \to \mathbb{F}_R \quad \text{by} \quad x \mapsto \sqcup\{y \in \mathbb{F}_R \mid y \sqsubseteq x\},$$

and

$$g\colon \mathbb{F}_R \to \uparrow(1, R) \subseteq \mathbb{F} \quad \text{by} \quad (s, S) \mapsto (s, S \cap R),$$

and finally,

$$|_R = g \circ f\colon \mathbb{F} \to \mathbb{F}.$$

Note first that all these mappings are well-defined. Indeed, the set $Y = \{y \in \mathbb{F}_R \mid y \sqsubseteq x\}$ is bounded above in $\mathbb{F}$, so its sup exists and belongs to $\mathbb{F}_R$. To show that g is well-defined, one only has to observe that $\mathrm{resinf}(s) \subseteq S \cap R$ when $(s, S) \in \mathbb{F}_R$. Therefore, $|_R$ is well-defined, too.

Proposition 7. *The mapping $|_R : \mathbb{F} \to \mathbb{F}$ defined by $x|_R = g \circ f(x)$ is continuous. Moreover, we have $\mathrm{res}(x|_R) = \mathrm{res}(x) \cap R$ for all $x \in \mathbb{F}$.* □

From this, we obtain the semantics of the restriction operator by

$$|_R\colon [\mathbb{F}^V \to \mathbb{F}] \to [\mathbb{F}^V \to \mathbb{F}] \quad \text{by} \quad (f|_R)(\sigma) = f(\sigma)|_R.$$

6.4 Parallel Composition

We require some preliminary definitions and results before we can define the parallel composition of resource traces. We use the results from Section 2.3 to define the semantics of this operation. Recall first that we assumed the existence of a parallel composition over actions of the alphabet: $\| : \Sigma^2 \to \Sigma$ that satisfies $\mathrm{res}(a_1 \| a_2) = \mathrm{res}(a_1) \cup \mathrm{res}(a_2)$ for all $(a_1, a_2) \in \Sigma^2$. The action $a_1 \| a_2$ represents the result of synchronizing a_1 and a_2 in a parallel composition.

We introduce the alphabet $\Sigma' = (\Sigma \cup \{1\})^2 \setminus \{(1, 1)\}$ with the resource map $\mathrm{res}'(a_1, a_2) = \mathrm{res}(a_1) \cup \mathrm{res}(a_2)$ and the associated dependence relation D'. Then we consider the sets $\mathbb{R}(\Sigma', D')$ and $\mathbb{F}(\Sigma', D')$ of real traces and of resource traces over the resource alphabet $\mathrm{res}' : \Sigma' \to \mathcal{P}(\mathcal{R})$. We define the alphabetic mappings

$$\begin{aligned} \Pi_i &: \Sigma' \to \Sigma \cup \{1\} \quad \text{by} \quad \Pi_i(a_1, a_2) = a_i, i = 1, 2 \text{ and} \\ \Pi &: \Sigma' \to \Sigma \quad\quad\quad\quad \text{by} \quad \Pi(a_1, a_2) = a_1 \| a_2. \end{aligned}$$

where we set $a\|1 = 1\|a = a$. Note that $\mathrm{res}(\Pi_i(a_1, a_2)) \subseteq \mathrm{res}'(a_1, a_2)$, $i = 1, 2$, and $\mathrm{res}(\Pi(a_1, a_2)) = \mathrm{res}'(a_1, a_2)$. Therefore, the three mappings extend to continuous morphisms over real traces (Proposition 2) and to continuous maps over resource traces. Moreover, Π also is a morphism of resource traces (Proposition 3).

We consider a subset $C \subseteq \mathcal{R}$ of resources on which we want to synchronize; we call these resources *channels*. We fix two resource traces $x_1 = (s_1, S_1)$ and $x_2 = (s_2, S_2)$ of $\mathbb{F}(\Sigma, D)$ and we want to define a resource trace $x_1 \underset{C}{\|} x_2$ which represents the parallel composition of x_1 and x_2 with synchronization on the channels of C. We first define a resource trace $\varphi(x_1, x_2) \in \mathbb{F}(\Sigma', D')$ which represents the parallel composition of x_1 and x_2. Then we set $x_1 \underset{C}{\|} x_2 = \Pi(\varphi(x_1, x_2))$. Since the mapping Π is continuous, in order to obtain a continuous semantics for parallel composition, we only need to show that the mapping $\varphi : \mathbb{F}(\Sigma, D)^2 \to \mathbb{F}(\Sigma', D')$ is continuous as well.

In analogy to the set $\mathrm{Sync}_{C,\sigma}(p_1, p_2)$ for terms $p_1, p_2 \in \mathcal{L}$, given resource traces $x_i = (s_i, S_i)$, $i = 1, 2$, we can define the *synchronization set* $\mathrm{Sync}_C(x_1, x_2)$ as the set of pairs $(a_1, a_2) \in \mathrm{Alph}(s_1) \times \mathrm{Alph}(s_2)$ satisfying

$$\mathrm{res}(a_1) \cap C = \mathrm{res}(a_2) \cap C = \mathrm{res}(a_1) \cap \mathrm{res}(x_2) = \mathrm{res}(a_2) \cap \mathrm{res}(x_1) \neq \emptyset.$$

Then the set $\Sigma'_C(x_1, x_2)$ of actions which may occur in $\varphi(x_1, x_2)$ is defined as

$$\begin{aligned}\Sigma'_C(x_1, x_2) = {} & \{(a_1, 1) \in \mathrm{Alph}(s_1) \times \{1\} \mid \mathrm{res}(a_1) \cap (C \cup \mathrm{res}(x_2)) = \emptyset\} \\ & \cup \{(1, a_2) \in \{1\} \times \mathrm{Alph}(s_2) \mid \mathrm{res}(a_2) \cap (C \cup \mathrm{res}(x_1)) = \emptyset\} \\ & \cup \mathrm{Sync}_{C,\sigma}(x_1, x_2).\end{aligned}$$

The first two sets in this union correspond to *local events:* these should not use any channel on which we want to synchronize ($\mathrm{res}(a_1) \cap C = \emptyset$). In addition, the condition $\mathrm{res}(a_1) \cap \mathrm{res}(x_2) = \emptyset$ implies that a local event does not conflict with any event of the other component, which ensures parallel composition does not involve nondeterministic choice. The set $\mathrm{Sync}_{C,\sigma}(x_1, x_2)$ corresponds to synchronization events. In order to synchronize, two events must use exactly the same channels and, in order to assure determinism, neither should conflict with resources of the other component.

Now we introduce the set

$$\begin{aligned}X_C(x_1, x_2) = \{(t, T) \in \mathbb{F}(\Sigma', D') \mid {} & \mathrm{Alph}(t) \subseteq \Sigma'_C(x_1, x_2) \text{ and} \\ & \Pi_i(t, T) \sqsubseteq x_i \text{ for } i = 1, 2\}\end{aligned}$$

Proposition 8. *The set $X_C(x_1, x_2)$ has a least upper bound $x = (r, R)$ given by*

$$\begin{aligned}r &= \sqcup\{r \in \mathbb{R}(\Sigma'_C(x_1, x_2)) \mid \Pi_i(r) \leq s_i \text{ for } i = 1, 2\}, \\ R &= S_1 \cup S_2 \cup \mathrm{res}(r_1^{-1} s_1) \cup \mathrm{res}(r_2^{-1} s_2), \text{ where } r_i = \Pi_i(r).\end{aligned}$$

Moreover, $X_C(x_1, x_2) = {\downarrow}x$ and $\mathrm{res}(x) = \mathrm{res}(x_1) \cup \mathrm{res}(x_2)$. □

Proposition 9. *The mapping* $\varphi\colon \mathbb{F}(\Sigma, D)^2 \to \mathbb{F}(\Sigma', D')$ *defined by* $\varphi(x_1, x_2) = \sqcup X_C(x_1, x_2)$ *is continuous.* □

As announced earlier, we define the semantics of parallel composition by $\underset{C}{\|} = \Pi \circ \varphi$ and the above results imply:

Corollary 2. $\underset{C}{\|} : \mathbb{F}(\Sigma, D)^2 \to \mathbb{F}(\Sigma, D)$ by $x_1 \underset{C}{\|} x_2 = (\Pi \circ \varphi)(x_1, x_2)$ *is continuous. Moreover,* $\mathrm{res}(x_1 \underset{C}{\|} x_2) = \mathrm{res}(x_1) \cup \mathrm{res}(x_2)$ *for all* $(x_1, x_2) \in \mathbb{F}^2$. □

The semantics of parallel composition is defined by

$$\underset{C}{\|} : [\mathbb{F}^V \to \mathbb{F}]^2 \to [\mathbb{F}^V \to \mathbb{F}] \quad \text{by} \quad (f_1 \underset{C}{\|} f_2)(\sigma) = f_1(\sigma) \underset{C}{\|} f_2(\sigma).$$

6.5 Recursion

In order to have a compositional semantics for recursion, we need to define for each variable $x \in V$ an interpretation $\mathrm{rec}\, x : [\mathbb{F}^V \to \mathbb{F}] \to [\mathbb{F}^V \to \mathbb{F}]$, and then we will set $[\![\mathrm{rec}\, x.p]\!] = \mathrm{rec}\, x.[\![p]\!]$. For $f \in [\mathbb{F}^V \to \mathbb{F}]$, $\mathrm{rec}\, x.f$ will be defined as a fixed point of a continuous selfmap from $\mathbb{F}$ to $\mathbb{F}$, but we do not use the classical least fixed point semantics. Indeed, for the process $\mathrm{rec}\, x.(a \circ x)$, the least fixed point semantics would give $(a^\omega, \mathcal{R})$ which claims and blocks unnecessarily all resources. Instead, the semantics should be $(a^\omega, \mathrm{res}(a))$ which claims only the resources needed for its execution. To obtain this semantics, we have to start the fixed point computation with $(1, \mathrm{res}(a))$, which is the least element of the subdomain in which our computation is taking place. We describe our approach in some detail now. Fixing a variable $x \in V$, we first define the maps

$$\varphi\colon [\mathbb{F}^V \to \mathbb{F}] \times \mathbb{F}^V \to [\mathbb{F} \to \mathbb{F}] \quad \text{by} \quad (f, \sigma) \mapsto \varphi_{f,\sigma}$$
$$\psi : [\mathbb{F}^V \to \mathbb{F}] \times \mathbb{F}^V \to [(\mathcal{P}(\mathcal{R}), \supseteq) \to (\mathcal{P}(\mathcal{R}), \supseteq)] \quad \text{by} \quad (f, \sigma) \mapsto \psi_{f,\sigma}$$

where

$$\varphi_{f,\sigma}(y) = f(\sigma[x \mapsto y]),$$
$$\psi_{f,\sigma}(R) = \mathrm{res}(f(\sigma[x \mapsto (1, R)])).$$

Proposition 10. *The two maps* φ *and* ψ *are well-defined and continuous.* □

For $\sigma \in \mathbb{F}^V$, we define $(\mathrm{rec}\, x.f)(\sigma)$ as a fixed point of the continuous map $\varphi_{f,\sigma}$. Instead of using the least fixed point of $\varphi_{f,\sigma}$, we start the iteration yielding the fixed point from a resource trace $\perp_{f,\sigma}$ which depends on f and σ.

We define the mapping $R\colon [\mathbb{F}^V \to \mathbb{F}] \times \mathbb{F}^V \to \mathcal{P}(\mathcal{R})$ by $(f, \sigma) \mapsto R_{f,\sigma} = \mathrm{FIX}(\psi_{f,\sigma})$ which assigns to each pair (f, σ) the *greatest fixed point* of the monotone selfmap $\psi_{f,\sigma}$. The starting point for the iteration is simply the resource trace $\perp_{f,\sigma} = (1, R_{f,\sigma})$. Therefore, we also have a mapping $\perp\colon [\mathbb{F}^V \to \mathbb{F}] \times \mathbb{F}^V \to \mathbb{F}$ by $(f, \sigma) \mapsto \perp_{f,\sigma} = (1, R_{f,\sigma})$.

Lemma 2. *The maps* $R : [\mathbb{F}^V \to \mathbb{F}] \times \mathbb{F}^V \to \mathcal{P}(\mathcal{R})$ *and* $\perp : [\mathbb{F}^V \to \mathbb{F}] \times \mathbb{F}^V \to \mathbb{F}$ *are continuous.* □

We define the semantics of recursion by

$$\operatorname{rec} x\colon [\mathbb{F}^V \to \mathbb{F}] \to [\mathbb{F}^V \to \mathbb{F}] \text{ by } f \mapsto \operatorname{rec} x.f : \sigma \mapsto \bigsqcup_{n\geq 0} \varphi^n_{f,\sigma}(\bot_{f,\sigma}).$$

Proposition 11. *The mapping* $\operatorname{rec} x\colon [\mathbb{F}^V \to \mathbb{F}] \to [\mathbb{F}^V \to \mathbb{F}]$ *is well defined and continuous.* □

6.6 Summary

We have defined our denotational semantics as a compositional mapping $[\![-]\!]\colon \mathcal{L} \to [\mathbb{F}^V \to \mathbb{F}]$; the work in this section has validated that such a mapping exists, since $\mathcal{L}$ is the initial Ω-algebra, and we have given a continuous interpretation in $[\mathbb{F}^V \to \mathbb{F}]$ for each of the operators $\omega \in \Omega$ in the signature of our language. To summarize, the semantics of a process $p \in \mathcal{L}$ is the continuous map $[\![p]\!]$ defined inductively by:

$$\begin{aligned}
[\![\mathrm{STOP}]\!](\sigma) &= (1, \mathcal{R}) & [\![p \circ q]\!](\sigma) &= [\![p]\!](\sigma) \cdot [\![q]\!](\sigma)\\
[\![a]\!](\sigma) &= (a, \emptyset) & [\![p \underset{C}{\|} q]\!](\sigma) &= [\![p]\!](\sigma) \underset{C}{\|} [\![q]\!](\sigma)\\
[\![x]\!](\sigma) &= \sigma(x) & [\![p|_R]\!](\sigma) &= ([\![p]\!](\sigma))|_R\\
[\![\operatorname{rec} x.p]\!](\sigma) &= (\operatorname{rec} x.[\![p]\!])(\sigma) = \bigsqcup_{n\geq 0} \varphi^n_{[\![p]\!],\sigma}(\bot_{[\![p]\!],\sigma}).
\end{aligned}$$

7 The Main Theorem

In this section we complete the picture by showing the relationship between the operational and denotational models for our language.

To begin, we relate the semantic resources of a process to the syntactic resource of the process. The semantic resource set of the process $p \in \mathcal{L}$ in some environment $\sigma \in \mathbb{F}^V$ is given by $\operatorname{res}([\![p]\!](\sigma))$. In order to relate this semantic resource set to the syntactic resource set defined in Section 4, we introduce the map $\operatorname{res}^V : \mathbb{F}^V \to \mathcal{P}(\mathcal{R})^V$ by $\operatorname{res}^V(\sigma)(x) = \operatorname{res}(\sigma(x))$.

Proposition 12. *Let* $p \in \mathcal{L}$ *and* $\sigma \in \mathbb{F}^V$*, then* $\operatorname{res}([\![p]\!](\sigma)) = \operatorname{res}(p, \operatorname{res}^V(\sigma))$. □

The following result is the key lemma for the main theorem. It requires an extended sequence of results to derive.

Proposition 13. *Let* $p, q \in \mathcal{L}$*,* $a \in \Sigma$*,* $\sigma \in \mathbb{F}^V$ *and* $\tau \in \mathcal{P}(\mathcal{R})^V$*. Then*

1. $p \xrightarrow[\operatorname{res}^V(\sigma)]{a} q \;\Rightarrow\; [\![p]\!](\sigma) = a \cdot [\![q]\!](\sigma)$,
2. $[\![p]\!](\tau) = a \cdot [\![q]\!](\tau) \;\Rightarrow\; p \xrightarrow[\tau]{a} q$. □

Using the above proposition, we can show that each possible operational behavior of p in some environment $\sigma \in \mathcal{P}(\mathcal{R})^V$ corresponds to some compact resource trace below $[\![p]\!](\sigma)$, and, conversely, that each compact resource trace below $[\![p]\!](\sigma)$ approximates some operational behavior of p in σ.

More precisely, in order to relate the operational and the denotational semantics, we use the mapping $\chi : \mathbb{F} \to \mathcal{P}(K(\mathbb{F}))$ defined by $\chi(x) = \{(s, S) \in K(\mathbb{F}) \mid s \leq x, S = \mathrm{res}(s^{-1} \cdot x)\}$. Note that for all $x \in \mathbb{F}$ we have $\chi(x) \subseteq K(x) \subseteq {\downarrow}\chi(x)$ and therefore $\chi(x)$ is directed and $\sqcup\chi(x) = x$.

Theorem 1. *For all $p \in \mathcal{L}$ and $\sigma \in \mathcal{P}(\mathcal{R})^V$, we have*

$$X_{\mathbb{F}}(p, \sigma) = \chi([\![p]\!](\sigma))$$

and therefore $B_{\mathbb{F}}(p, \sigma) = [\![p]\!](\sigma)$. It follows directly that the denotational semantics $[\![-]\!]$ is adequate with respect to the operational semantics defined by $X_{\mathbb{M}}$, by $X_{\mathbb{F}}$ or by $B_{\mathbb{F}}$. □

8 Closing Remarks

We have presented a simple language that includes a number of interesting operators: weak sequential composition, deterministic parallel composition, restriction and recursion. We also have presented a theorem relating its operational semantics to its denotational semantics and implying adequacy. The novel feature of our language is that the semantics of parallel composition does not involve nondeterministic choice, as in other approaches. We believe this language will have some interesting applications, among them the analysis of security protocols (where determinism has proved to be an important property) and model checking, where trace theory has been used to avoid the state explosion problem. We are exploring these applications in our ongoing work.

What remains to be done is to expand the language to include some of the missing operators from the usual approach to process algebra. Chief among these are the hiding operator of CSP and a choice operator.

References

1. Abramsky, S and A. Jung, *Domain Theory*, in: "Handbook of Computer Science and Logic," Volume **3**, Clarendon Press, 1995.
2. Diekert, V. and P. Gastin, *Approximating traces*, *Acta Informatica* **35** (1998), pp. 567–593.
3. Diekert, V. and G. Rozenberg, editors, "The Book of Traces," World Scientific, Singapore (1995).
4. Gastin, P. and D. Teodesiu, *Resource traces: a domain for process sharing exclusive resources, Theoretical Computer Science*, to appear.
5. Hennessy, M. and G. D. Plotkin, *Full abstraction for a simple parallel programming language,* Lecture Notes in Computer Science **74** (1979), Springer-Verlag
6. Mazurkiewicz, A., *Trace theory,* Lecture Notes in Computer Science **255** (1987), pp. 279–324.
7. Pratt, V., *On the composition of processes,* Proceedings of the Ninth POPL (1982).
8. Reisig, W., "Petri Nets," EATCS Monographs in Theoretical Computer Science **4** (1985), Springer-Verlag.
9. Winskel, G., "Events in Computation," Ph.D. Thesis, University of Cambridge, 1980.

Specification Refinement with System F

Jo Erskine Hannay

LFCS, Division of Informatics, University of Edinburgh, Scotland, U.K.
joh@dcs.ed.ac.uk

Abstract. Essential concepts of algebraic specification refinement are translated into a type-theoretic setting involving System F and Reynolds' relational parametricity assertion as expressed in Plotkin and Abadi's logic for parametric polymorphism. At first order, the type-theoretic setting provides a canonical picture of algebraic specification refinement. At higher order, the type-theoretic setting allows future generalisation of the principles of algebraic specification refinement to higher order and polymorphism. We show the equivalence of the acquired type-theoretic notion of specification refinement with that from algebraic specification. To do this, a generic algebraic-specification strategy for behavioural refinement proofs is mirrored in the type-theoretic setting.

1 Introduction

This paper aims to express in type theory certain essential concepts of algebraic specification refinement. The benefit to algebraic specification is that inherently first-order concepts are translated into a setting in which they may be generalised through the full force of the chosen type theory. Furthermore, in algebraic specification many concepts have numerous theoretical variants. Here, the setting of type theory may provide a somewhat sobering framework, in that type-theoretic formalisms insist on certain sensibly canonical choices.

On the other hand, the benefit to type theory is to draw from the rich source of formalisms, development methodology and reasoning techniques in algebraic specification. See [7] for a survey and comprehensive bibliography. One of the most appealing and successful endeavours in algebraic specification is that of *stepwise specification refinement*, in which abstract descriptions of processes and data types are methodically refined to concrete executable descriptions, *viz.* programs and program modules. In this paper we base ourselves on the description in [31,30], and we highlight three essential concepts that make this account of specification refinement apt for real-life development. These are so-called *constructor implementations*, *behavioural equivalence* and *stability*. We will express this refinement framework in a type-theoretic environment comprised of System F and the assumption of relational parametricity in Reynolds' sense [27,18], as expressed in Plotkin and Abadi's logic for parametric polymorphism [24]. Abstract data types are expressed in the type theory as existential types.

The above concepts of specification refinement fall out naturally in this setting. In this, relational parametricity plays an essential role. It gives the equivalence at first order of observational equivalence to equality at existential type.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 530–545, 1999.

In algebraic specification there is a generic proof strategy formalised in [6,4,5] for proving observational refinements. This considers axiomatisations of so-called behavioural (partial) congruences. As also observed in [25], Plotkin and Abadi's logic is not sufficient to accommodate this proof strategy. Inspired by [25], we choose the simple solution of adding axioms stating the existence of quotients and sub-objects. This is justified by the soundness of the axioms w.r.t. the parametric PER-model [3] for Plotkin and Abadi's logic. In this paper we import the proof strategy into type theory to show a correspondence between two notions of refinement. But this importation is also interesting in its own right, and our results complement those of [25] in that we consider also partial congruences.

Other work linking algebraic specification and type theory includes [17] encoding constructor implementations in ECC, [26] expressing module-algebra axioms in ECC, [23] encoding behavioural equalities in UTT, [2] treating the specification language ASL+, [35] using Nuprl as a specification language, and [34] promoting dependent types in specification. Only [25] utilises relational parametricity. There are also non-type-theoretic higher-order approaches using higher-order universal algebra [20], and other set-theoretic models [16].

The next section outlines algebraic specification refinement, highlighting the three essential concepts above. Then, the translation of algebraic specification refinement into a System F environment is presented, giving a type-theoretic notion of specification refinement. The main result of this paper is a correspondence at first-order between algebraic specification refinement and the type-theoretic notion of specification refinement. This sets the scene for generalising the refinement concepts now implanted in type theory to higher order and polymorphism.

2 Algebraic Specification Refinement

Let $\Sigma = \langle S, \Omega \rangle$ be a signature, consisting of a set S of sorts, and an $S^* \times S$-sorted set Ω of operator names. We write profiles $f : s_1 \times \cdots \times s_n \to s \in \Omega$, meaning $f \in \Omega_{s_1,\ldots,s_n,s}$. A Σ-algebra $A = \langle (A)_{s \in S}, F \rangle$ consists of an S-sorted set $(A)_{s \in S}$ of non-empty carriers and a set F containing a total function $f^A \in (A_{s_1} \times \cdots \times A_{s_n} \to A_s)$ for every $f : s_1 \times \cdots \times s_n \to s \in \Omega$. The class of Σ-algebras is denoted by $\Sigma\mathbf{Alg}$. Given a countable S-sorted set X of variables, the free Σ-algebra over X is denoted $T_\Sigma(X)$ and for $s \in S$ the carrier $T_\Sigma(X)_s$ contains the terms of sort s. We consider sorted first-order logic with equality. A formula φ is a Σ-formula if all terms in φ are of sorts in S. Let Φ be a set of closed Σ-formulae. Then $SP = \langle \Sigma, \Phi \rangle$ is a basic algebraic specification, and its semantics $[\![SP]\!]$ is $Mod_\Sigma(\Phi)$, the class of Σ-algebras that are models of Φ.

Example 1. The following specification specifies stacks of natural numbers.

spec Stack **is**
  sorts nat, stack
  operators empty : stack, push : nat × stack → stack,
    pop : stack → stack, top : stack → nat
  axioms Φ_{Stack} : pop(push(x, s)) $= s$
    top(push(x, s)) $= x$

We omit universal quantification over free variables in examples. The semantics of a data type (in a program) is an algebra. Wide-spectrum specification languages *e.g.* Extended ML [14], allow specifications and programs to be written in a uniform language, so that specifications are abstract descriptions of a data type or systems of data types, while program modules and programs are concrete executable descriptions of the same. A refinement process seeks to develop in a sound methodical way the latter from the former, and a program is then a *full refinement* or *realisation* of an abstract specification. The basic definition of refinement we adopt here is given by the following refinement relation $\rightsquigarrow$ on specifications of the same signature [30,32]: $SP_j \rightsquigarrow SP_{j+1} \stackrel{def}{\Longleftrightarrow} [\![SP_j]\!] \supseteq [\![SP_{j+1}]\!]$.

There are two indispensable refinements as it were, of the refinement relation. One introduces constructors, the other involves behavioural abstraction.

A refinement process involves making decisions about design and implementation detail. At some point a particular function or module may become completely determined and remain unchanged throughout the remainder of the refinement process. It is convenient to lay aside the fully refined parts and continue development on the remaining unresolved parts only. Let κ be a *parameterised program* [9] with input interface SP_{j+1} and output interface SP_j. Given a program P that is a full refinement of SP_{j+1}, the instantiation $\kappa(P)$ is then a full refinement of SP_j. The semantics of a parameterised program is a function $[\![\kappa]\!] \in (\Sigma_{SP_{j+1}}\mathbf{Alg} \to \Sigma_{SP_j}\mathbf{Alg})$ called a *constructor*. *Constructor implementation* is then defined [30] as $SP_j \stackrel{\cdot}{\rightsquigarrow}_{\kappa} SP_{j+1} \stackrel{def}{\Longleftrightarrow} [\![SP_j]\!] \supseteq [\![\kappa]\!]([\![SP_{j+1}]\!])$. The parameterised program κ is the fully refined part of the system which is set aside, and SP_{j+1} specifies the remaining unresolved part that needs further refinement.

A major point in algebraic specification is that an abstract specification really is abstract enough to give freedom of implementation. The notion of *behavioural abstraction* captures the concept that two programs are considered equivalent if their observable behaviours are equivalent. Algebraically one assumes a designated set $Obs \subseteq S$ of observable sorts, and a designated set $In \subseteq S$ of input sorts. Observable computations are represented by terms in $T_\Sigma(X^{In})_s$, for $s \in Obs$ and where $X^{In}_s = X_s$ for $s \in In$ and $\emptyset$ otherwise. Two Σ-algebras A and B are *observationally equivalent w.r.t.* Obs, In, written $A \equiv_{Obs,In} B$, if every observable computation has equivalent denotations in A and B [29]. However, the semantics $[\![SP]\!]$ is not always closed under behavioural equivalence. For example, the stack-with-pointer implementation of stacks of natural numbers does not satisfy $\mathsf{pop}(\mathsf{push}(x, s)) = s$ and is not in $[\![\mathsf{Stack}]\!]$, but is behaviourally equivalent w.r.t. $Obs = In = \{\mathsf{nat}\}$ to an algebra that is. To capture this, one defines the semantics $[\![SP]\!]_{Obs,In} \stackrel{def}{=} \{B \mid \exists A \in [\![SP]\!] \,.\, B \equiv_{Obs,In} A\}$, and defines *refinement up to behavioural equivalence* [30] as $\langle SP_j, Obs, In\rangle \stackrel{\cdot}{\rightsquigarrow}_{\kappa} \langle SP_{j+1}, Obs', In'\rangle \stackrel{def}{\Longleftrightarrow} [\![SP_j]\!]_{Obs,In} \supseteq [\![\kappa]\!]([\![SP_{j+1}]\!]_{Obs',In'})$. Why do we want designated input sorts? One extremal view would be to say that all observable computations should be ground terms, *i.e.* proclaim $In = \emptyset$. But that would be too strict in a refinement situation where a data type depends on another as yet undeveloped data type. On the other hand, letting all sorts be input sorts would disallow intuitively feasible behavioural refinements as illustrated in the following example from [10].

Example 2. Consider the following specification of sets of natural numbers.

spec Set **is**
sorts nat, set
operators empty : set, add : nat × set → set
in : nat × set → bool, remove : nat × set → set
axioms $\mathsf{add}(x, \mathsf{add}(x, s)) = \mathsf{add}(x, s)$
$\mathsf{add}(x, \mathsf{add}(y, s)) = \mathsf{add}(y, \mathsf{add}(x, s))$
$\mathsf{in}(x, \mathsf{empty}) = \mathsf{false}$
$\mathsf{in}(x, \mathsf{add}(y, s)) = \mathsf{if}\ x =_{\mathsf{nat}} y\ \mathsf{then\ true\ else}\ \mathsf{in}(x, s)$
$\mathsf{in}(x, \mathsf{remove}(x, s)) = \mathsf{false}$

Consider the Σ_{Set}-algebra *ListImpl* (LI) whose carrier LI_{set} is the set of finite lists over the natural numbers; empty^{LI} gives the empty list, add^{LI} appends a given element to the end of a list only if the element does not occur already, in^{LI} is the occurrence function, and remove^{LI} removes the first occurrence of a given element. Being a Σ_{Set}-algebra, LI allows users only to build lists using empty^{LI} and add^{LI}, and on such lists the efficient remove^{LI} gives the intended result. However, $LI \notin [\![\mathsf{Set}]\!]_{Obs,In}$, for $Obs = \{\mathsf{bool}, \mathsf{nat}\}$ and $In = \{\mathsf{set}, \mathsf{bool}, \mathsf{nat}\}$, because the observable computation $\mathsf{in}(x, \mathsf{remove}(x, s))$ might give true, since s ranges over all lists, not only the canonical ones generated by empty^{LI} and add^{LI}. On the other hand, $LI \in [\![\mathsf{Set}]\!]_{Obs,In}$ for $In = Obs = \{\mathsf{bool}, \mathsf{nat}\}$, since now the use of set-variables in observable computations is prohibited. ○

In this example, the correct choice was $In = Obs$. In fact $In = Obs$ is virtually always a sensible choice, and a very reasonable simplifying assumption.

Behavioural refinement steps are in general hard to verify. A helpful concept is *stability* [33]. A constructor $[\![\kappa]\!]$ is stable if $A \equiv_{Obs',In'} B \Rightarrow [\![\kappa]\!](A) \equiv_{Obs,In} [\![\kappa]\!](B)$. Under stability, it suffices for proving $\langle SP_j, Obs, In\rangle \rightsquigarrow_{\kappa} \langle SP_{j+1}, Obs', In'\rangle$, to show that $[\![SP_j]\!]_{Obs,In} \supseteq [\![\kappa]\!]([\![SP_{j+1}]\!])$. The following contrived but short example from [31] illustrates the point. See *e.g.* [33] for a more realistic example.

Example 3. Consider the specification

spec Triv **is**
sorts nat
operators id : nat × nat × nat → nat
axioms Φ_{Triv} : $\mathsf{id}(x, n, z) = x$

Define the constructor $Tr \in (\Sigma_{\mathsf{Stack}}\mathbf{Alg} \to \Sigma_{\mathsf{Triv}}\mathbf{Alg})$ as follows. For $A \in \Sigma_{\mathsf{Stack}}\mathbf{Alg}$, define $multipush_A \in (\mathbb{N} \times \mathbb{N} \times A \to A)$ and $multipop_A \in (\mathbb{N} \times A \to A)$ by

$$multipush_A(n, z, a) = \begin{cases} a, & n = 0 \\ \mathsf{push}^A(z, multipush_A(n-1, z+1, a)), & n > 0 \end{cases}$$

$$multipop_A(n, a) = \begin{cases} a, & n = 0 \\ multipop_A(n-1, \mathsf{pop}^A(a)), & n > 0 \end{cases}$$

Then $Tr(A)$ is the Triv-algebra whose single operator is given by

$$id(x,n,z) = \mathsf{top}^A(\mathit{multipop}_A(n, \mathit{multipush}_A(n,z,\mathsf{push}^A(x,\mathsf{empty}^A)))).$$

We have $[\![\mathsf{Triv}]\!]_{\{\mathsf{nat}\},In} \supseteq Tr([\![\mathsf{Stack}]\!]_{\{\mathsf{nat}\},In})$, but to prove this only assuming membership in $[\![\mathsf{Stack}]\!]_{\{\mathsf{nat}\},In}$ is not straight-forward. However, Tr is in fact stable. So it suffices to show $[\![\mathsf{Triv}]\!]_{\{\mathsf{nat}\},In} \supseteq Tr([\![\mathsf{Stack}]\!])$, and the proof of this is easy [31]. In particular, one may now hold $\mathsf{pop}(\mathsf{push}(x,s)) = s$ among ones assumptions, although this formula is not valid for $[\![\mathsf{Stack}]\!]_{\{\mathsf{nat}\},In}$. ○

One still has to prove the stability of constructors. However, since constructors are given by concrete parameterised programs, this can be done in advance for the language as a whole. A key observation is that stability is intimately related to the effectiveness of encapsulation mechanisms in the language.

Example 4 ([31]). Consider the constructor $Tr' \in (\Sigma_{\mathsf{Stack}}\mathbf{Alg} \to \Sigma_{\mathsf{Triv}}\mathbf{Alg})$ such that $Tr'(A)$ is the Triv-algebra whose single operator is given by

$$id(x,n,z) = \begin{cases} x, & \mathsf{pop}^A(\mathsf{push}^A(z,\mathsf{empty}^A)) = \mathsf{empty}^A \\ z, & \text{otherwise} \end{cases}$$

Then for A the array-with-pointer algebra, we get $Tr'(A) \notin [\![Triv]\!]_{\{\mathsf{nat}\},In}$ and so $[\![\mathsf{Triv}]\!]_{\{\mathsf{nat}\},In} \not\supseteq Tr'([\![\mathsf{Stack}]\!]_{\{\mathsf{nat}\},In})$. Tr' is not stable, and Tr' breaches the abstraction barrier by checking equality on the underlying implementation. ○

Algebraic specifications may be complex, built from basic specifications using specification building operators, *e.g.* [36,32,37]. But as a starting point for the translation into type theory, we only consider basic specifications.

3 The Type Theory

We now sketch the logic in [24,19] for parametric polymorphism on System F. It is this accompanying logic that bears an extension rather than the type theory. See [1] for a more internalised approach. System F has types and terms as follows.

$$T ::= X \mid T \to T \mid \forall X.T \qquad\qquad t ::= x \mid \lambda x{:}T.t \mid tt \mid \Lambda X.t \mid tT$$

where X and x range over type and term variables resp. However, formulae are now built using the usual connectives from equations *and* relation symbols.

$$\phi ::= (t =_A u) \mid R(t,u) \mid \cdots \mid \forall R \subset A \times B.\phi \mid \exists R \subset A \times B.\phi$$

where R ranges over relation symbols. We write $\alpha[R,X,x]$ to indicate possible occurrences of R, X and x in α, and may write $\alpha[\rho,A,t]$ for the result of substitution, following the appropriate rules concerning capture.

Judgements for type and term formation and second-order environments with term environments depending on type environments, are as usual. But formula formation now involves relation symbols, so second-order environments are augmented with relation environments, *viz.* a finite sequence Υ of relational typings $R \subset A \times B$ of relation variables, depending on the type environment, and obeying

standard conventions for environments. The formation rules for atomic formulae consists of the usual one for equations, and now also one for relations:

$$\frac{\Gamma \vdash t{:}A, \quad \Gamma \vdash u{:}B, \quad \Gamma \vdash \Upsilon, \quad \Upsilon \vdash R \subset A \times B}{\Gamma, \Upsilon \vdash R(t,u)\ \mathit{Prop} \qquad (\mathit{also\ written}\ tRu)}$$

The other rules for formulae are as expected. Relation definition is accommodated:

$$\frac{\Gamma, x{:}A, y{:}B \ \vdash \ \phi\ \mathit{Prop}}{\Gamma \ \vdash \ (x{:}A, y{:}B)\ .\ \phi \quad \subset A \times B}$$

For example $\mathsf{eq}_A \stackrel{def}{=} (x{:}A, y{:}A).(x =_A y)$.

If $\rho \subset A \times B$, $\rho' \subset A' \times B'$ and $\rho''[R] \subset A[Y] \times B[Z]$, then complex relations are built by $\rho \to \rho' \subset (A \to A') \times (B \to B')$ where

$$(\rho \to \rho') \stackrel{def}{=} (f{:}A \to A', g{:}B \to B').(\forall x{:}A \forall x'{:}B.(x \rho x' \ \Rightarrow \ (fx)\rho'(gx')))$$

and $\forall(Y, Z, R \subset Y \times Z)\rho''[R] \subset (\forall Y.A[Y]) \times (\forall Z.B[Z])$ where

$$\forall(Y, Z, R \subset Y \times Z)\rho'' \stackrel{def}{=} (y{:}\forall Y.A[Y], z{:}\forall Z.B[Z]).(\forall Y \forall Z \forall R \subset Y \times Z.((yY)\rho''[R](zZ)))$$

One can now acquire further definable relations by substituting definable relations for type variables in types. For $\boldsymbol{X} = X_1, \ldots, X_n$, $\boldsymbol{B} = B_1, \ldots, B_n$, $\boldsymbol{C} = C_1, \ldots, C_n$ and $\boldsymbol{\rho} = \rho_1, \ldots, \rho_n$, where $\rho_i \subset B_i \times C_i$, we get $T[\boldsymbol{\rho}] \subset T[\boldsymbol{B}] \times T[\boldsymbol{C}]$, the action of $T[\boldsymbol{X}]$ on $\boldsymbol{\rho}$, defined by cases on $T[\boldsymbol{X}]$ as follows:

$$\begin{array}{ll} T[\boldsymbol{X}] = X_i : & T[\boldsymbol{\rho}] = \rho_i \\ T[\boldsymbol{X}] = T'[\boldsymbol{X}] \to T''[\boldsymbol{X}] : & T[\boldsymbol{\rho}] = T'[\boldsymbol{\rho}] \to T''[\boldsymbol{\rho}] \\ T[\boldsymbol{X}] = \forall X'.T'[\boldsymbol{X}, X'] : & T[\boldsymbol{\rho}] = \forall(Y, Z, R \subset Y \times Z).T'[\boldsymbol{\rho}, R] \end{array}$$

The proof system is natural deduction over formulae now involving relation symbols, and is augmented with inference rules for relation symbols, for example we have for Φ a finite set of formulae:

$$\frac{\Phi \vdash_{\Gamma, R \subset A \times B} \phi[R]}{\Phi \vdash_\Gamma \forall R \subset A \times B\ .\ \phi[R]} \qquad \frac{\Phi \vdash_\Gamma \forall R \subset A \times B.\phi[R], \quad \Gamma \vdash \rho \subset A \times B}{\Phi \vdash_\Gamma \phi[\rho]}$$

One also has axioms for equational reasoning and $\beta\eta$ equalities. Finally, the following parametricity axiom schema is asserted:

$$\textsc{Param} : \vdash_\emptyset \forall Y_1, \ldots, \forall Y_n \forall u{:}(\forall X.T[X, Y_1, \ldots, Y_n])\ .\ u(\forall X.T[X, \mathsf{eq}_{Y_1}, \ldots, \mathsf{eq}_{Y_n}])u$$

To understand, it helps to ignore the parameters Y_i and expand the definition to get $\forall u{:}(\forall X.T[X])\ .\forall Y \forall Z \forall R \subset Y \times Z\ .\ u(Y)\ T[R]\ u(Z)$ *i.e.* if one instantiates a polymorphic inhabitant at two related types then the results are also related. One gets

Fact 1 (Identity Extension Lemma [24]**).** *For any $T[\boldsymbol{Z}]$, the following sequent is derivable using* Param.

$$\vdash_\emptyset \forall \boldsymbol{Z}.\forall u, v{:}T\ .\ (u\ T[\mathsf{eq}_{\boldsymbol{Z}}]\ v \ \Leftrightarrow \ (u =_T v))$$

Encapsulation is provided by the following encoding of existential types and the following pack and unpack combinators.

$$\exists X.T[X] \stackrel{def}{=} \forall Y.(\forall X.(T[X] \to Y) \to Y)$$

$$\mathsf{pack}_{T[X]}{:}\forall X.(T[X] \to \exists X.T[X])$$
$$\mathsf{pack}_{T[X]}(A)(impl) \stackrel{def}{=} \Lambda Y.\lambda f{:}\forall X.(T[X] \to Y).f(A)(impl)$$

$$\mathsf{unpack}_{T[X]}{:}(\exists X.T[X]) \to \forall Y.(\forall X.(T[X] \to Y) \to Y)$$
$$\mathsf{unpack}_{T[X]}(package)(B)(client) \stackrel{def}{=} package(B)(client)$$

We omit subscripts to pack and unpack as much as possible. Operationally, pack packages a data representation and an implementation of operators on that data representation. The resulting package is a polymorphic functional that given a client and its result domain, instantiates the client with the particular elements of the package. And unpack is the application operator for pack.

Fact 2 (Characterisation by Simulation Relation [24]**).** *The following sequent schema is derivable using* Param.

$$\vdash_{\emptyset} \forall \boldsymbol{Z}.\forall u, v{:}\exists X.T[X, \boldsymbol{Z}] \ .$$
$$u =_{\exists X.T[X,\boldsymbol{Z}]} v \quad \Leftrightarrow \quad \exists A, B.\exists \mathfrak{a}{:}T[A, \boldsymbol{Z}], \mathfrak{b}{:}T[B, \boldsymbol{Z}].\exists R \subset A \times B \ .$$
$$u = (\mathsf{pack} A \mathfrak{a}) \ \wedge \ v = (\mathsf{pack} B \mathfrak{b}) \ \wedge \ \mathfrak{a}(T[R, \mathbf{eq}_{\boldsymbol{Z}}])\mathfrak{b}$$

The sequent in Fact 2 states the equivalence of equality at existential type with the existence of a simulation relation in the sense of [21]. From this we also get

$$\vdash_{\emptyset} \forall \boldsymbol{Z}.\forall u{:}\exists X.T[X, \boldsymbol{Z}].\exists A.\exists \mathfrak{a}{:}T[A, \boldsymbol{Z}] \ . \ u = (\mathsf{pack} A \ \mathfrak{a})$$

Weak versions of standard constructs such as products, initial and final (co-)algebras are encodable in System F [8]. With Param, these constructs are provably universal constructions. We can *e.g.* freely use product types. Given $\rho \subset A \times B$ and $\rho' \subset A' \times B'$, $(\rho \times \rho)$ is defined as the action $(X \times X')[\rho, \rho']$. One derives $\forall u{:}A \times A', v{:}B \times B' \ . \ u(\rho \times \rho')v \ \Leftrightarrow \ (\mathsf{fst}(u) \ \rho \ \mathsf{fst}(v) \wedge \mathsf{snd}(u) \ \rho \ \mathsf{snd}(v))$. We also use the abbreviations $\mathsf{bool} \stackrel{def}{=} \forall X.X \to X \to X$ and $\mathsf{nat} \stackrel{def}{=} \forall X.X \to (X \to X) \to X$; which are provably initial constructs.

Finally, this logic is sound w.r.t. to the parametric PER-model of [3].

4 The Translation

We now define a translation $\mathcal{T}$ giving an interpretation in the type theory and logic outlined in Sect. 3, of the concept of algebraic specification refinement $\langle SP_j, Obs, In\rangle \rightsquigarrow_{\kappa} \langle SP_{j+1}, Obs', In'\rangle$. We will use inhabitants of existential types as analogues to algebras, and then existentially quantified variables will correspond to non-observable (*behavioural*) sorts.

To keep things simple, we will at any one refinement stage assume a single behavioural sort b; methodologically this means focusing on one data type at a

time, and on one thread in a development. Thus we can stick to existential types with one existentially quantified variable. It is straight-forward to generalise to multiple existentially quantified variables [21].

In algebraic specification, there is no restraint on the choice of input sorts and observable sorts within the sorts S of a signature. In the type-theoretic setting, we will see that we have only one choice for the corresponding notion of *input types*, namely the collection of all types. Since a behavioural sort corresponds to an existentially quantified type variable, this automatically caters for situations which in algebraic specification correspond to crucially excluding the behavioural sort from the input sorts (Example 2). In algebraic specification, conforming to this type-theoretic insistence means assuming $In = S \setminus b$, which probably covers all reasonable examples of refinement. Thus, the type-theoretic formalisms inherently select a sensible choice.

For *observable types* on the other hand, we seem to have some choice. Our assumption of at most one behavioural sort means $Obs = S \setminus b$, hence $Obs = In$, in the algebraic specification setting. In type theory we could therefore let all types be observable types, as we must for input types. However, since 'observable' should mean 'printable', we limit the observable types by letting Obs denote also observable types; we assume that for every sort $s \in Obs$ there is an obvious closed type given the name s, for which the Identity Extension Lemma (Fact 1) gives $x\,(s[\rho])\,y \;\Leftrightarrow\; x =_s y$. Examples are bool and nat.

Note that the assumption of $Obs = In$ means that it suffices to write algebraic specification refinement as $\langle SP_j, Obs\rangle \rightsquigarrow_\kappa \langle SP_{j+1}, Obs'\rangle$.

In the following we use record type notation as a notational convenience.

Definition 1 (Translation and Type Theory Specification). *Let $SP = \langle \Sigma, \Phi\rangle$ where $\Sigma = \langle S, \Omega\rangle$. Define the translation $\mathcal{T}$ by*

$$\mathcal{T}\langle SP, Obs\rangle = \langle\langle Sig_{SP}, \Theta_{SP}\rangle, Obs\rangle$$

where $Sig_{SP} = \exists X.Prof_{SP}$,
where $Prof_{SP} = Record(f_1\colon s_{11}\times\cdots\times s_{1n_1} \to s_1,\ \ldots\ , f_k\colon s_{k1}\times\cdots\times s_{kn_k} \to s_k)[X/b]$,
for $f_i\colon s_{i1} \times \cdots \times s_{in_i} \to s_i \in \Omega$,
and where $\Theta_{SP}(u) = \exists X.\exists \mathfrak{x}\colon Prof_{SP}\ .\ u = (\mathsf{pack}X\mathfrak{x}) \ \wedge\ \Phi[X, \mathfrak{x}]$.

Here, $\Phi[X, \mathfrak{x}]$ indicates the conjunction of Φ, where X substitutes b, and every operator symbol in Φ belonging to Ω is prefixed with $\mathfrak{x}$. We call $\mathcal{T}\langle SP, Obs\rangle$ a type theory specification. *If $\Theta_{SP}(u)$ is derivable then u is a* realisation *of $\mathcal{T}\langle SP, Obs\rangle$.*

Example 5. For example, $\mathcal{T}\langle \mathsf{Stack}, \{\mathsf{nat}\}\rangle = \langle\langle Sig_{\mathsf{Stack}}, \Theta_{\mathsf{Stack}}\rangle, \{\mathsf{nat}\}\rangle$, where

$Sig_{\mathsf{Stack}} = \exists X.Prof_{\mathsf{Stack}}$,
$Prof_{\mathsf{Stack}} = Record(\mathsf{empty}\colon X, \mathsf{push}\colon \mathsf{nat} \times X \to X, \mathsf{pop}\colon X \to X, \mathsf{top}\colon X \to \mathsf{nat})$
$\Theta_{\mathsf{Stack}}(u) = \exists X.\exists \mathfrak{x}\colon Prof_{\mathsf{Stack}}\ .\ u = (\mathsf{pack}X\mathfrak{x}) \ \wedge$
$\forall x\colon \mathsf{nat}, s\colon X\ .\ \mathfrak{x}.\mathsf{pop}(\mathfrak{x}.\mathsf{push}(x, s)) = s \ \wedge$
$\forall x\colon \mathsf{nat}, s\colon X\ .\ \mathfrak{x}.\mathsf{top}(\mathfrak{x}.\mathsf{push}(x, s)) = x$ ○

Henceforth, existential types arise from algebraic specifications as in Def. 1. We do not consider free type variables in existential types since this corresponds to parameterised algebraic specifications, which is outside this paper's scope.

The type theory specification of Def. 1 is essentially that of [17]. The important difference is that with parametricity, equality of data type inhabitants is inherently behavioural, so implementation is up to observational equivalence.

In algebraic specification we said that two Σ-algebras A and B are observationally equivalent w.r.t. *Obs* and *In* iff for any observable computation $t \in T_\Sigma(X^{In})_s$, $s \in Obs$ the interpretations t^A and t^B are equivalent. Analogously, and in the style of [21], we give the following definition of type-theoretic observational equivalence.

Definition 2 (Type Theory Observational Equivalence). *For any u, v: $\exists X.T[X]$, we say u and v are* observationally equivalent w.r.t. *Obs iff the following sequent is derivable.*

$$\vdash_\Gamma \exists A, B.\exists \mathfrak{a}{:}T[A], \mathfrak{b}{:}T[B] \;.\; u = (\mathsf{pack} A\,\mathfrak{a}) \;\wedge\; v = (\mathsf{pack} B\,\mathfrak{b}) \;\wedge\; \bigwedge_{C \in Obs} \forall f{:}\forall X.(T[X] \to C) \;.\; (f\,A\,\mathfrak{a}) = (f\,B\,\mathfrak{b})$$

Notice that there is nothing hindering having free variables in an observable computation $f{:}\forall X.(T[X] \to C)$. Importantly, though, these free variables can not be of the existentially bound type.

Example 6. Recalling Example 2, for *ListImpl* to be a behavioural implementation of Set, it was essential that the input sorts did not include set, as then the observable computation $\mathsf{in}(x, \mathsf{remove}(x, s))$ would not have the same denotation in *ListImpl* as in any algebra in $[\![\mathsf{Set}]\!]$. In our type-theoretic setting, the corresponding observable computation is $\Lambda X.\lambda \mathfrak{x}{:}\, Prof_{\mathsf{Set}} \;.\; \mathfrak{x}.\mathsf{in}(x, \mathfrak{x}.\mathsf{remove}(x, g))$. Here g must be a term of the bound type X. The typing rules insist that g can only be of the form $\mathfrak{x}.\mathsf{add}(\cdots \mathfrak{x}.\mathsf{add}(\mathfrak{x}.\mathsf{empty}) \cdots)$ and not a free variable. ○

Our first result is essential to understanding the translation.

Theorem 3. *Suppose $\exists X.T[X] = Sig_{SP}$ in $\mathcal{T}\langle SP, Obs\rangle$ for some basic algebraic specification SP and set of observable sorts Obs. Then, assuming* Param*, equality at existential type is derivably equivalent to observational equivalence, i.e. the following sequent is derivable in the logic.*

$$\vdash_\emptyset \forall u, v{:}\exists X.T[X] \;.\; u =_{\exists X.T[X]} v \;\Leftrightarrow\; \exists A, B.\exists \mathfrak{a}{:}T[A], \mathfrak{b}{:}T[B] \;.\; u = (\mathsf{pack} A\,\mathfrak{a}) \;\wedge\; v = (\mathsf{pack} B\,\mathfrak{b}) \;\wedge\; \bigwedge_{C \in Obs} \forall f{:}\forall X.(T[X] \to C) \;.\; (f\,A\,\mathfrak{a}) = (f\,B\,\mathfrak{b})$$

Proof: This follows from Fact 2 and Lemma 4 below. □

Lemma 4. *Let $\exists X.T[X] = Sig_{SP}$ be as in Theorem 3. Then, assuming* Param*, the existence of a simulation relation is derivably equivalent to observational equivalence, i.e. the following sequent is derivable.*

$$\vdash_\emptyset \forall A, B.\forall \mathfrak{a}{:}T[A], \mathfrak{b}{:}T[B] \;.\; \exists R \subset A \times B \;.\; \mathfrak{a}(T[R])\mathfrak{b} \;\Leftrightarrow\; \bigwedge_{C \in Obs} \forall f{:}\forall X.(T[X] \to C) \;.\; (f\,A\,\mathfrak{a}) = (f\,B\,\mathfrak{b})$$

Proof: $\Rightarrow$: This follows from PARAM.

$\Leftarrow$: We must exhibit an R such that $\mathfrak{a}(T[R])\mathfrak{b}$. Semantically, [22,33] define a relation between elements iff they are denotable by some common term. We mimic this: Give $R \stackrel{def}{=} (a{:}A, b{:}B).(\exists f{:}\forall X.(T[X] \to X).(f\,A\,\mathfrak{a}) = a \wedge (f\,B\,\mathfrak{b}) = b)$. We must now derive $\mathfrak{a}(T[R])\mathfrak{b}$, *i.e.* for every component $(g{:}\,s_1 \times \cdots \times s_n \to s)[X/b]$ in $T[X]$, we must show that

$$\forall v_1{:}\,s_1[A], \ldots, \forall v_n{:}\,s_n[A], \forall w_1{:}\,s_1[B], \ldots, \forall w_n{:}\,s_n[B]\ .\\ v_1\ s_1[R]\ w_1\ \wedge\ \cdots\ \wedge\ v_n\ s_n[R]\ w_n\\ \Rightarrow\ \mathfrak{a}.g(v_1, \ldots, v_n)\ s[R]\ \mathfrak{b}.g(w_1, \ldots, w_n)$$

Under our present assumptions, any s_j in the antecedent is either b or else an observable sort. If s_j is b then the antecedent says $v_j\ R\ w_j$ hence we may assume $\exists f_j{:}\forall X.(T[X] \to X).(f_j A\,\mathfrak{a}) = v_j\ \wedge\ (f_j B\,\mathfrak{b}) = w_j$. If s_j is an observable sort we may by Fact 1 assume $v_j = w_j$. Consider $f \stackrel{def}{=} \Lambda X.\lambda\mathfrak{x}{:}\,T[X]\ .\ \mathfrak{x}.g(u_1, \ldots, u_n)$, where u_j is $(f_j X\mathfrak{x})$ if s_j is b, and $u_j = v_j$ otherwise.

Suppose now the co-domain sort s is an observable sort. Then by assumption we have $(f\,A\,\mathfrak{a}) = (f\,B\,\mathfrak{b})$ and by β-reduction we are done. Suppose the co-domain sort s is b. Then we need to derive $\mathfrak{a}.g(v_1, \ldots, v_n)\ R\ \mathfrak{b}.g(w_1, \ldots, w_n)$, *i.e.* that $\exists f{:}\forall X.(T[X] \to X).(f\,A\,\mathfrak{a}) = \mathfrak{a}.g(v_1, \ldots, v_n)\ \wedge\ (f\,B\,\mathfrak{b}) = \mathfrak{b}.g(w_1, \ldots, w_n)$. But then we exhibit our f above, and we are done. □

This proof does not in general generalise to higher order T. The problem lies in exhibiting a simulation relation R.

Given Theorem 3, $\Theta_{SP}(u)$ of translation $\mathcal{T}$ expresses "u is observationally equivalent to a package $(\mathsf{pack}X\mathfrak{x})$ that satisfies the axioms Φ". Therefore:

Definition 3 (Type Theory Specification Refinement). *A type theory specification $\mathcal{T}\langle SP', Obs'\rangle$ is a* refinement *of a type theory specification $\mathcal{T}\langle SP, Obs\rangle$,* with constructor $F{:}\,Sig_{SP'} \to Sig_{SP}$ *iff* $\vdash_\Gamma \forall u{:}\,Sig_{SP'}\ .\ \Theta_{SP'}(u)\ \Rightarrow\ \Theta_{SP}(Fu)$ *is derivable. We write $\mathcal{T}\langle SP, Obs\rangle \stackrel{\mathcal{T}}{\rightsquigarrow}_F \mathcal{T}\langle SP', Obs'\rangle$ for this fact.*

Any constructor $F{:}\,Sig_{SP'} \to Sig_{SP}$ is by Theorem 3 inherently stable under parametricity: Congruence gives $\forall u, v{:}\,Sig_{SP'}.u =_{Sig_{SP'}} v\ \Rightarrow\ F(u) =_{Sig_{SP}} F(v)$. But equality at existential type is of course observational equivalence.

Example 7. The constructor Tr of Example 3 is expressed in this setting as $\lambda u{:}\,Sig_{\mathsf{Stack}}.\mathsf{unpack}(u)(Sig_{\mathsf{Triv}})(\Lambda X.\lambda\mathfrak{x}{:}\,Prof_{\mathsf{Stack}}\ .\ (\mathsf{pack}X\ record(\mathsf{id} =$
$\lambda x, n, z{:}\,\mathsf{nat}\ .\ \mathfrak{x}.\mathsf{top}(\mathsf{multipop}(n, \mathsf{multipush}(n, z, \mathfrak{x}.\mathsf{push}(x, \mathfrak{x}.\mathsf{empty}))))))))$ ○

Note that we automatically get the proof simplification due to stability that we have in algebraic specification. Since observational equivalence is simply equality in the type theory, it is sound to substitute any package with an observationally equivalent package that satisfies the axioms of the specification literally.

Observe that the non-stable constructor Tr' from Example 4 is not expressible in the type theory, because $x =_X y$ *Prop* is not allowed in System F terms.

5 A Correspondence at First Order

We seek to establish a formal connection between the concept of algebraic specification refinement and its type-theoretic counterpart as defined in Def. 3, *i.e.*

$$\langle SP, Obs\rangle \rightsquigarrow_{\kappa} \langle SP', Obs'\rangle \;\Leftrightarrow\; \mathcal{T}\langle SP, Obs\rangle \overset{\mathcal{T}}{\underset{F_\kappa}{\rightsquigarrow}} \mathcal{T}\langle SP', Obs'\rangle$$

where κ and F_κ are constructors that correspond in a sense given below.

Now, that u is a realisation of a type theory specification $\langle\langle Sig_{SP}, \Theta_{SP}\rangle, Obs\rangle$ can in general only be proven by exhibiting an observationally equivalent package u' that satisfies Φ_{SP}. For any particular closed term $g\colon Sig_{SP}$, one can attempt to construct such a g' perhaps ingeniously, using details of g. But to show that a specification is a refinement of another specification we are asked to consider a term ($\mathsf{pack}A\mathfrak{a}$) where we do not know details of A or $\mathfrak{a}$. We therefore need a universal method for exhibiting suitable observationally equivalent packages. It also defies the point of behavioural abstraction having to construe a literal implementation to justify a behavioural one.

In algebraic specification one proves observational refinements by first considering quotients w.r.t. a possibly partial congruence $\approx_{Obs,In}$ induced by Obs and In [5], and then using an axiomatisation of this quotienting congruence to prove relativised versions of the axioms of the specification to be refined. In the case that this congruence is partial, clauses restricting to the domain of the congruence must also be incorporated [6,4]. The quotients are of the form $dom_A(\approx_{Obs,In})/\!\approx_{Obs,In}$, where $dom_A(\approx_{Obs,In})_s \overset{def}{=} \{a \in A_s \mid a \approx_{Obs,In} a\}$.

This proof method is not available in the type theory and logic of [24]. One remedy would be to augment the type theory by quotient types, *e.g.* [11], and subset types. However, for its simplicity and because it complies to existing proof techniques in algebraic specification, we adapt an idea from [25] where the logic is augmented with an axiom schema postulating the existence of quotients (Def. 4). In addition, we need a schema asserting the existence of sub-objects (Def. 5) for dealing with partial congruences. The justification for these axioms lies in their soundness w.r.t. the parametric PER-model [3] that is one justification for the logic of [24]. These axioms are tailored to suit refinement proof purposes. One could alternatively derive them from more fundamental and general axioms.

Definition 4 (Existence of Quotients (Quot) [25]**).**

$$\begin{aligned}\vdash_\emptyset \forall X.\forall \mathfrak{x}\colon T[X].\forall R \subset X\times X\;.\quad & (\mathfrak{x}\ T[R]\ \mathfrak{x}\ \wedge\ equiv(R))\ \Rightarrow\\ \exists Q.\exists \mathfrak{q}\colon T[Q].\exists epi\colon X\to Q\;.\; \forall x,y\colon X\;.\; xRy\ &\Leftrightarrow\ (epi\,x) =_Q (epi\,y)\ \wedge\\ \forall q\colon Q.\exists x\colon X\;.\; q =_Q (epi\,x)\ &\wedge\\ \mathfrak{x}\ (T[(x\colon X, q\colon Q).((epi\,x) =_Q q)])\ \mathfrak{q}&\end{aligned}$$

where $equiv(R)$ specifies R to be an equivalence relation.

Definition 5 (Existence of Sub-objects (Sub)**).**

$$\begin{aligned}\vdash_\emptyset \forall X.\forall \mathfrak{x}\colon T[X].\forall R\subset X\times X\;.\quad & (\mathfrak{x}\ T[R]\ \mathfrak{x})\ \Rightarrow\\ \exists S.\exists \mathfrak{s}\colon T[S].\exists R'\subset S\times S.\exists mono\colon S\to X\;.\; & (T[(x\colon X, s\colon S).(x = _X(mono\,s))])\ \mathfrak{s}\quad\wedge\\ \forall s.s'\colon S\;.\; s\ R'\ s'\ &\Leftrightarrow\ (mono\,s)\ R\ (mono\,s')\ \wedge\\ \forall s\colon S\;.\; s\ R'\ s&\end{aligned}$$

Algebraic specification uses classical logic, while the logic in [24] is constructive. However, formulae may be interpreted classically in the parametric PER-model, and it is sound w.r.t. this model to assume the axiom of excluded middle [24]. For our comparison with algebraic specification, we shall do this.

We can now show our desired correspondence. We first do this for refinements without constructors. We must assume that specifications are *behaviourally closed w.r.t.* $\approx_{Obs,In}$, *i.e.* $\{dom_A(\approx_{Obs,In})/\approx_{Obs,In} \mid A \in [\![SP]\!]\} \subseteq [\![SP]\!]$. This is methodologically an obvious requirement for behavioural specification [6,5].

Theorem 5. *Let* $SP = \langle \Sigma, \Phi \rangle$ *and* $SP' = \langle \Sigma, \Phi' \rangle$ *be basic algebraic specifications, with* $\Sigma = \langle S, \Omega \rangle$. *Assume one behavioural sort* b, *and assume* $Obs = In = S \setminus b$. *Assume behavioural closedness w.r.t.* $\approx_{Obs,In}$. *Then*

$$\langle SP, Obs \rangle \rightsquigarrow \langle SP', Obs \rangle \;\Leftrightarrow\; \mathcal{T}\langle SP, Obs \rangle \overset{\mathcal{T}}{\rightsquigarrow} \mathcal{T}\langle SP', Obs \rangle$$

Proof: $\Rightarrow$: We must show the derivability of $\vdash_\Gamma \forall u{:}\, Sig_{SP'} \,.\, \Theta_{SP'}(u) \;\Rightarrow\; \Theta_{SP}(u)$. We can obtain proof-theoretical information from $\langle SP, Obs \rangle \rightsquigarrow \langle SP', Obs \rangle$. By behavioural closedness, there exists a sound and complete calculus $\vdash_{\Pi_{\rightsquigarrow}}$ for behavioural refinement, based on a calculus $\vdash_{\Pi_S}$ for structured specifications [6]. By syntax directedness, we must have had $SP'/\approx_{Obs,In} \vdash_{\Pi_S} \Phi$, where the semantics of $SP'/\approx_{Obs,In}$ is $\{dom_A(\approx_{Obs,In})/\approx_{Obs,In} \mid A \in [\![SP']\!]\}$. For our basic specification case, this boils down to the predicate logic statement of

$$\Phi', Ax(\sim) \;\vdash\; \mathcal{L}(\Phi) \qquad (\dagger)$$

Here $\sim$ stands for a new symbol representing $\approx_{Obs,In}$ at the behavioural sort b, and $\mathcal{L}(\Phi) \stackrel{def}{=} \{\mathcal{L}(\phi) \mid \phi \in \Phi\}$, for $\mathcal{L}(\phi) = (\;\wedge_{\,y \in FV_b(\phi)}\; y \sim y) \;\Rightarrow\; \phi^*$ where $FV_b(\phi)$ is the set of free variables of sort b in ϕ, and where inductively

(a) $(u =_b v)^* \stackrel{def}{=} u \sim v$,
(b) $(\neg\phi)^* \stackrel{def}{=} \neg(\phi^*)$ and $(\phi \;\wedge\; \psi)^* \stackrel{def}{=} \phi^* \;\wedge\; \psi^*$,
(c) $(\forall x{:}\, b.\phi)^* \stackrel{def}{=} \forall x{:}\, b.(x \sim x \;\Rightarrow\; \phi^*)$,
(d) $\phi^* \stackrel{def}{=} \phi$, otherwise.

and $Ax(\sim) \stackrel{def}{=} \forall x, y{:}\, b.(x \sim y \;\Leftrightarrow\; Beh_b(x, y))$, where $Beh_b(x, y)$ is an axiomatisation of $\approx_{Obs,In}$ at b [4]. (At $s \in Obs = In$, $\approx_{Obs,In}$ is just equality.)

Using this we derive our goal as follows. Let $u{:}\, Sig_{SP'}$ be arbitrary. Let T denote $Prof_{SP'}(= Prof_{SP})$. We must derive $\exists B.\exists \mathfrak{b}{:}\, T[B].(\mathsf{pack}B\mathfrak{b}) = u \wedge \Phi[B, \mathfrak{b}]$ assuming $\exists A.\exists \mathfrak{a}{:}\, T[A].(\mathsf{pack}A\mathfrak{a}) = u \wedge \Phi'[A, \mathfrak{a}]$. Let $\mathfrak{a}$ and A denote the witnesses projected out from that assumption.

Now, *Beh* is in general infinitary. However, with higher-order logic one gets a finitary Beh^* equivalent to *Beh* [12]. Thus we form $\sim$ type-theoretically by $\sim \;\stackrel{def}{=}\; (a{:}\, A, a'{:}\, A).(Beh^*_A(a, a'))$. Since $\sim$ is an axiomatisation of a partial congruence, we have $\mathfrak{a}\; T[\sim]\; \mathfrak{a}$. We use SUB to get S_A, $\mathfrak{s}_\mathfrak{a}$ and $\sim' \subset S_A \times S_A$ and $mono{:}\, S_A \to A$ s.t. we can derive

($s1$) $\mathfrak{a}\; (T[(a{:}\, A, s{:}\, S_A).(a =_A (mono\, s))])\; \mathfrak{s}_\mathfrak{a}$
($s2$) $\forall s.s'{:}\, S_A \;.\; s \sim' s' \;\Leftrightarrow\; (mono\, s) \sim (mono\, s')$
($s3$) $\forall s{:}\, S_A \;.\; s \;\sim'\; s$

By $(s2)$ we get $\mathfrak{s}_\mathfrak{a}\ T[\sim']\ \mathfrak{s}_\mathfrak{a}$. We also get $equiv(\sim')$ by $(s3)$. We now use QUOT to get Q and $\mathfrak{q}{:}T[Q]$ and $epi{:}S_A \to Q$ s.t.

$(q1)\ \forall s, s'{:}S_A\ .\ s \sim' s' \ \Leftrightarrow\ (epi\ s) =_Q (epi\ s')$
$(q2)\ \forall q{:}Q.\exists s{:}S_A\ .\ q =_Q (epi\ s)$
$(q3)\ \mathfrak{s}_\mathfrak{a}\ (T[(s{:}S_A, q{:}Q).((epi\ s) =_Q q)])\ \mathfrak{q}$

We exhibit Q for B, and $\mathfrak{q}$ for $\mathfrak{b}$; it remains to derive 1. $(\mathsf{pack}Q\mathfrak{q}) = (\mathsf{pack}A\mathfrak{a})$ and 2. $\Phi[Q, \mathfrak{q}]$. To show the derivability of (1), it suffices to observe that, through Fact 2, $(s1)$ and $(q3)$ give $(\mathsf{pack}A\mathfrak{a}) = (\mathsf{pack}S_A\mathfrak{s}_\mathfrak{a}) = (\mathsf{pack}Q\mathfrak{q})$. For (2) we must show the derivability of $\phi[Q, \mathfrak{q}]$ for every $\phi \in \Phi$. We induce on the structure of ϕ.

(a) ϕ is $u =_b v$. We must derive $u[\mathfrak{q}] =_Q v[\mathfrak{q}]$. For any variable $q_i{:}Q$ in $u[\mathfrak{q}]$ or $v[\mathfrak{q}]$, we may by $(q2)$ assume an $s_{q_i}{:}S_A$ s.t. $(epi\ s_{q_i}) = q_i$. From (†) we can derive $\wedge_i((mono\ s_{q_i}) \sim (mono\ s_{q_i})) \Rightarrow u[\mathfrak{a}][\cdots(mono\ s_{q_i})\cdots] \sim v[\mathfrak{a}][\cdots(mono\ s_{q_i})\cdots]$, but by $(s2)$ and $(s1)$ this is equivalent to $\wedge_i(s_{q_i} \sim' s_{q_i}) \ \Rightarrow\ u[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i}\cdots] \sim' v[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i}\cdots]$, which by $(s3)$ is equivalent to $u[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i}\cdots] \sim' v[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i}\cdots]$.

Then from $(q1)$ we can derive $(epi\ u[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i}\cdots]) =_Q (epi\ v[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i}\cdots])$. By $(q3)$ we then get $(epi\ u[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i}\cdots]) = u[\mathfrak{q}]$ and $(epi\ v[\mathfrak{s}_\mathfrak{a}][\cdots s_{q_i}\cdots]) = v[\mathfrak{q}]$.

(b) Suppose $\phi = \neg\phi'$. By negation n.f. convertibility it suffices to consider ϕ' an atomic formula. The case for ϕ' as (a) warrants a proof for $\neg\phi'$ similar to that of (a). Suppose $\phi = \phi' \wedge \phi''$. This is dealt with by i.h. on ϕ' and ϕ''.

(c) $\phi = \forall x{:}B.\phi'$. This is dealt with by i.h. on ϕ'.

(d) This covers the remaining cases. Proofs are similar to those above.

$\Leftarrow$: Observe that to show $\Phi[Q, \mathfrak{q}]$ we must either use $\Phi'[Q, \mathfrak{q}]$ and the definition of $\sim$, or else $\Phi[Q, \mathfrak{q}]$ was a tautology; in both cases we get (†). □

We can easily extend Theorem 5 to deal with constructors. Dealing with constructors in full generality, requires specification building operators, which is outside the scope of this paper. However, consider *simple* type theory constructors $F{:}\,Sig_{SP'} \to Sig_{SP}$ of the form

$$\lambda u{:}\,Sig_{SP'}.\mathsf{unpack}(u)(Sig_{SP})(\Lambda X.\lambda\mathfrak{x}{:}\,Prof_{SP'}[X]\ .\ (\mathsf{pack}X\mathfrak{x}'))$$

for some $\mathfrak{x}' : Prof_{SP}[X]$. The concept of algebraic specification of algebras is extended in [28] to algebraic specifications of constructors. In the simple case, we can extend our translation in Def. 1 to this framework.

Example 8. An algebraic specification of Example 3's Tr, can be given by $\Pi S{:}\mathsf{Stack.Triv}'[S]$ where $\mathsf{Triv}'[S]$ is

hide multipush, multipop **in**
operators multipush: nat × nat × S.stack → S.stack,
multipop: nat × S.stack → S.stack, id: nat × nat × nat → nat
axioms Φ_{Tr} : multipop$(n,$ multipush$(n, z, s)) = s$
id$(x, n, z) = S$.top(multipop$(n,$ multipush$(n, z, S$.push$(x, S$.empty$))))$

We can give a corresponding type theory specification $\mathcal{T}(\Pi S{:}\,\mathsf{Stack.Triv}')$ by $\langle Sig_{\Pi S:\mathsf{Stack.Triv}'}, \Theta_{\Pi S:\mathsf{Stack.Triv}'}\rangle$, where $Sig_{\Pi S:\mathsf{Stack.Triv}'} \stackrel{def}{=} Sig_{\mathsf{Stack}} \to Sig_{\mathsf{Triv}}$ and

$$\Theta_{\Pi S:\mathsf{Stack.Triv'}}(u,v) \stackrel{def}{=} \exists X.\exists \mathfrak{x}\colon Prof_{\mathsf{Stack}}.\exists Y.\exists \mathfrak{y}\colon Prof_{\mathsf{Triv}}\ .\ (\mathsf{pack}X\mathfrak{x}) = u \ \wedge\ (\mathsf{pack}Y\mathfrak{y}) = v \ \wedge\ \ldots \wedge \forall x,n,z\colon\mathsf{nat}\ .\ \mathfrak{y}.\mathsf{id}(x,n,z) = \mathfrak{x}.\mathsf{top}(\mathsf{multipop}(n,\mathsf{multipush}(n,z,\mathfrak{x}.\mathsf{push}(x,\mathfrak{x}.\mathsf{empty}))))$$

whereby F is a realisation of, or satisfies, $\mathcal{T}(\Pi S\colon \mathsf{Stack.Triv'})$ if one can derive $\forall u\colon Sig_{\mathsf{Stack}}\ .\ \Theta_{\mathsf{Stack}}(u) \ \Rightarrow\ \Theta_{\Pi S:\mathsf{Stack.Triv'}}(u, Fu)$. ○

We now want to show $\langle SP, Obs\rangle \underset{\kappa_F}{\rightsquigarrow} \langle SP', Obs'\rangle \ \Leftrightarrow\ \mathcal{T}\langle SP, Obs\rangle \overset{\mathcal{T}}{\underset{F}{\rightsquigarrow}} \mathcal{T}\langle SP', Obs'\rangle$ where κ_F is a realisation of a specification SP_F that maps to a specification $\mathcal{T}(SP_F)$ for which F, a simple constructor, is a realisation, and where the axioms of SP_F and $\mathcal{T}(SP_F)$ are given by Φ_F. We have to show the derivability of $\vdash_\Gamma\ \forall u\colon Sig_{SP'}\ .\ \Theta_{SP'}(u) \ \Rightarrow\ \Theta_{SP}(Fu)$, supposing $\langle SP, Obs\rangle \underset{\kappa_F}{\rightsquigarrow} \langle SP', Obs'\rangle$. Similarly to the proof of Theorem 5, we get $\Phi', Ax(\sim), \Phi_F \vdash \mathcal{L}(\Phi)$ (‡) We need to exhibit a B and $\mathfrak{b}$ s.t. 1. $\mathsf{pack}B\mathfrak{b} = F(\mathsf{pack}A\mathfrak{a})$ and 2. $\Phi[B,\mathfrak{b}]$. We construct Q and $\mathfrak{q}$ from $F(\mathsf{pack}A\mathfrak{a}) = \mathsf{pack}A\mathfrak{a}'$, for $\mathfrak{a}'\colon Prof_{SP}[A]$, as in the proof of Theorem 5, and (1) follows as before. Then for (2), to show $\Phi[Q,\mathfrak{q}]$, and also for the converse direction, use (‡) in place of (†).

Finally and importantly, the '⇒' direction of the proof of Theorem 5 displays a reasoning technique in its own right for type theory specification refinement. This extends the discussion in [25] to deal also with partial congruences.

6 Final Remarks

In this paper we have expressed an account of algebraic specification refinement in System F and the logic for parametric polymorphism of [24]. We have seen in Sect. 4 how the concepts of behavioural (observational) refinement, and stable constructors are inherent in this type-theoretic setting, because at first order, equality at existential type is exactly observational equivalence (Theorem 3). We have shown a correspondence (Theorem 5) between refinement in the algebraic specification sense, and a notion of type theory specification refinement (Def. 3). We have seen how a proof technique from algebraic specification can be mirrored in type theory by extending the logic soundly with axioms QUOT and SUB, the latter also extending the discussion in [25].

The stage is now set for type-theoretic development in at least two directions. First, algebraic specification has much more to it than presented here. An obvious extension would be to express specification building operators in System F. This would also allow a full account of specifications of parameterised programs and also parameterised specifications [28].

Secondly, we can use our notion of type theory specification refinement and start looking at specification refinement for higher-order polymorphic functionals. In this context one must resolve what observational equivalence means, since the higher-order version of Theorem 3 is an open question. However, there are grounds to consider an alternative notion of simulation relation that would re-establish a higher-order version of Lemma 4. Operationally, the only way two concrete data types $(\mathsf{pack}A\mathfrak{a})\colon T[A]$ and $(\mathsf{pack}B\mathfrak{b})\colon T[B]$ can be utilised, is in clients of the form $\Lambda X.\lambda\mathfrak{x}\colon T[X]\ .\ \mathfrak{x}.t$. Such a client cannot incite the application

of functionals $\mathfrak{a}.f$ and $\mathfrak{b}.f$ whose domain types involve the instantiations A and B of the existential type variable, to arbitrary terms of appropriate instantiated types, but only to terms definable in some sense by items in the respective implementations $\mathfrak{a}$ and $\mathfrak{b}$. However, the usual notion of simulation relation considers in fact arbitrary terms. At first order, this does not matter, because one can exhibit a relation that explicitly restricts arguments to be definable, namely the relation R in the proof of Lemma 4. At higher-order, one could try altering the relational proof criteria by incorporating explicit definability clauses. This is reminiscent of recent approaches on the semantical level [15,13].

Acknowledgements Thanks are due to Martin Hofmann, Don Sannella, and the referees for helpful comments and suggestions. This research has been supported by EPSRC grant GR/K63795, and NFR (Norwegian Research Council) grant 110904/41.

References

1. M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121:9–58, 1993.
2. D. Aspinall. *Type Systems for Modular Programs and Specifications*. PhD thesis, University of Edinburgh, 1998.
3. E.S. Bainbridge, P.J. Freyd, A. Scedrov, and P.J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.
4. M. Bidoit and R. Hennicker. Behavioural theories and the proof of behavioural properties. *Theoretical Computer Science*, 165:3–55, 1996.
5. M. Bidoit, R. Hennicker, and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming*, 25:149–186, 1995.
6. M. Bidoit, R. Hennicker, and M. Wirsing. Proof systems for structured specifications with observability operators. *Theoretical Computer Sci.*, 173:393–443, 1997.
7. M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella (eds.). *Algebraic System Specification and Development: A Survey and Annotated Bibliography*, volume 501 of *LNCS*. Springer, 1991.
8. C. Böhm and A. Beraducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.
9. J.A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, 1984.
10. R. Hennicker. Structured specifications with behavioural operators: Semantics, proof methods and applications. Habilitationsschrift, LMU, München, 1997.
11. M. Hofmann. A simple model for quotient types. In *Proc. TLCA'95*, volume 902 of *LNCS*, pages 216–234. Springer, 1995.
12. M. Hofmann and D. Sannella. On behavioural abstraction and behavioural satisfaction in higher-order logic. *Theoretical Computer Science*, 167:3–45, 1996.
13. F. Honsell and D. Sannella. Pre-logical relations. In *Proc. CSL'99*, *LNCS*, 1999.
14. S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.
15. Y. Kinoshita, P.W. O'Hearn, A.J. Power, M. Takeyama, and R.D. Tennent. An axiomatic approach to binary logical relations with applications to data refinement. In *Proceedings of TACS'97*, volume 1281 of *LNCS*, pages 191–212. Springer, 1997.

16. H. Kirchner and P.D. Mosses. Algebraic specifications, higher-order types, and set-theoretic models. In *Proc. AMAST'98*, volume 1548 of *LNCS*, pages 378–388. Springer, 1998.
17. Z. Luo. Program specification and data type refinement in type theory. *Math. Struct. in Comp. Sci.*, 3:333–363, 1993.
18. Q. Ma and J.C. Reynolds. Types, abstraction and parametric polymorphism, part 2. In *Proc. 7th MFPS*, volume 598 of *LNCS*, pages 1–40. Springer, 1991.
19. H. Mairson. Outline of a proof theory of parametricity. In *ACM Symposium on Functional Programming and Computer Architecture*, volume 523 of *LNCS*, pages 313–327. Springer, 1991.
20. K. Meinke. Universal algebra in higher types. *Theoretical Computer Science*, 100:385–417, 1992.
21. J.C. Mitchell. On the equivalence of data representations. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 305–330. Academic Press, 1991.
22. J.C. Mitchell. *Foundations for Programming Languages.* Foundations of Computing Series. MIT Press, 1996.
23. N. Mylonakis. Behavioural specifications in type theory. In *Recent Trends in Data Type Spec., 11th WADT*, volume 1130 of *LNCS*, pages 394–408. Springer, 1995.
24. G. Plotkin and M. Abadi. A logic for parametric polymorphism. In *Proc. of TLCA 93*, volume 664 of *LNCS*, pages 361–375. Springer, 1993.
25. E. Poll and J. Zwanenburg. A logic for abstract data types as existential types. In *Proc. TLCA'99*, volume 1581 of *LNCS*, pages 310–324, 1999.
26. B. Reus and T. Streicher. Verifying properties of module construction in type theory. In *Proc. MFCS'93*, volume 711 of *LNCS*, pages 660–670, 1993.
27. J.C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, 83:513–523, 1983.
28. D. Sannella, S. Sokołowski, and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Inform.*, 29:689–736, 1992.
29. D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34:150–178, 1987.
30. D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Inform.*, 25(3):233–281, 1988.
31. D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.
32. D. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. In *Proc. 1983 Intl. Conf. on Foundations of Computation Theory*, volume 158 of *LNCS*, pages 413–427. Springer, 1983.
33. O. Schoett. *Data Abstraction and the Correctness of Modular Programming.* PhD thesis, University of Edinburgh, 1986.
34. T. Streicher and M. Wirsing. Dependent types considered necessary for specification languages. In *Recent Trends in Data Type Spec.*, volume 534 of *LNCS*, pages 323–339. Springer, 1990.
35. J. Underwood. Typing abstract data types. In *Recent Trends in Data Type Spec., Proc. 10th WADT*, volume 906 of *LNCS*, pages 437–452. Springer, 1994.
36. M. Wirsing. Structured specifications: Syntax, semantics and proof calculus. In *Logic and Algebra of Specification*, pages 411–442. Springer, 1993.
37. M. Wirsing. Algebraic specification languages: An overview. In *Recent Trends in Data Type Specification*, volume 906 of *LNCS*, pages 81–115. Springer, 1994.

Pre-logical Relations*

Furio Honsell[1,2] and Donald Sannella[1]

[1] Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ; furio@dcs.ed.ac.uk and dts@dcs.ed.ac.uk
[2] Dipartimento di Matematica e Informatica, Università di Udine

Abstract. We study a weakening of the notion of logical relations, called *pre-logical relations*, that has many of the features that make logical relations so useful as well as further algebraic properties including composability. The basic idea is simply to require the reverse implication in the definition of logical relations to hold only for pairs of functions that are expressible by the same lambda term. Pre-logical relations are the minimal weakening of logical relations that gives composability for extensional structures and simultaneously the most liberal definition that gives the Basic Lemma. The use of pre-logical relations in place of logical relations gives an improved version of Mitchell's representation independence theorem which characterizes observational equivalence for all signatures rather than just for first-order signatures. Pre-logical relations can be used in place of logical relations to give an account of data refinement where the fact that pre-logical relations compose explains why stepwise refinement is sound.

1 Introduction

Logical relations are structure-preserving relations between models of typed lambda calculus.

Definition 1.1. *Let $\mathcal{A}$ and $\mathcal{B}$ be Σ-applicative structures. A* logical relation $\mathcal{R}$ over $\mathcal{A}$ and $\mathcal{B}$ *is a family of relations* $\{R^\sigma \subseteq [\![\sigma]\!]^{\mathcal{A}} \times [\![\sigma]\!]^{\mathcal{B}}\}_{\sigma \in Types(B)}$ *such that:*

- $R^{\sigma\to\tau}(f,g)$ *iff* $\forall a \in [\![\sigma]\!]^{\mathcal{A}}.\forall b \in [\![\sigma]\!]^{\mathcal{B}}.R^\sigma(a,b) \Rightarrow R^\tau(App_{\mathcal{A}}\ f\ a, App_{\mathcal{B}}\ g\ b)$.
- $R^\sigma([\![c]\!]^{\mathcal{A}}, [\![c]\!]^{\mathcal{B}})$ *for every term constant* $c:\sigma$ *in* Σ.

Logical relations are used extensively in the study of typed lambda calculus and have applications outside lambda calculus, for example to abstract interpretation [Abr90] and data refinement [Ten94]. A good reference for logical relations is [Mit96]. An important but more difficult reference is [Sta85].

The Basic Lemma is the key to many of the applications of logical relations. It says that any logical relation over $\mathcal{A}$ and $\mathcal{B}$ relates the interpretation of each lambda term in $\mathcal{A}$ to its interpretation in $\mathcal{B}$.

* An extended version of this paper, which includes proofs, is Report ECS-LFCS-99-405, Univ. of Edinburgh (1999).

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 546–561, 1999.

Lemma 1.2 (Basic Lemma). *Let $\mathcal{R}$ be a logical relation over Henkin models $\mathcal{A}$ and $\mathcal{B}$. Then for all Γ-environments $\eta_{\mathcal{A}}, \eta_{\mathcal{B}}$ such that $R^{\Gamma}(\eta_{\mathcal{A}}, \eta_{\mathcal{B}})$ and every term $\Gamma \rhd M : \sigma$, $R^{\sigma}([\![\Gamma \rhd M : \sigma]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}}, [\![\Gamma \rhd M : \sigma]\!]^{\mathcal{B}}_{\eta_{\mathcal{B}}})$.* □

($R^{\Gamma}(\eta_{\mathcal{A}}, \eta_{\mathcal{B}})$ refers to the obvious extension of $\mathcal{R}$ to environments, see page 550.)

As structure-preserving relations, logical relations resemble familiar algebraic concepts like homomorphisms and congruence relations but they lack some of the convenient properties of such concepts. In particular, the composition of two logical relations is not in general a logical relation. This calls into question their application to data refinement at least, where one would expect composition to provide an account of stepwise refinement.

We propose a weakening of the notion of logical relations called *pre-logical relations* (Sect. 3) that has many of the features that make logical relations so useful — in particular, the Basic Lemma still holds for pre-logical relations (Lemma 4.1) — but having further algebraic properties including composability (Prop. 5.5). The basic idea is simply to require the reverse implication in the definition of logical relations to hold only for pairs of functions that are expressible by the same lambda term. Pre-logical relations turns out to be the minimal weakening of logical relations that gives composability for extensional structures (Corollary 7.2) and simultaneously the most liberal definition that gives the Basic Lemma. Pre-logical predicates (the unary case of pre-logical relations) coincide with sets that are invariant under Kripke logical relations with varying arity as introduced by Jung and Tiuryn [JT93] (Prop. 6.2). The use of pre-logical relations in place of logical relations gives an improved version of Mitchell's representation independence theorem (Corollaries 8.5 and 8.6 to Theorem 8.4) which characterizes observational equivalence for all signatures rather than just for first-order signatures. Pre-logical relations can be used in place of logical relations in Tennent's account of data refinement in [Ten94] and the fact that pre-logical relations compose explains why stepwise refinement is sound.

Many applications of logical relations follow a standard pattern where the result comes directly from the Basic Lemma once an appropriate logical relation has been defined. Some results in the literature follow similar lines in the sense that a type-indexed family of relations is defined by induction on types and a proof like that of the Basic Lemma is part of the construction, but the family of relations defined is not logical. Examples can be found in Plotkin's and Jung and Tiuryn's lambda-definability results using I-relations [Plo80] and Kripke logical relations with varying arity [JT93] respectively, and Gandy's proof of strong normalization using hereditarily strict monotonic functionals [Gan80]. In each of these cases, the family of relations involved turns out to be a pre-logical relation (Example 3.8, Sect. 6 and Example 3.9) which allows the common pattern to be extended to these cases as well. Since pre-logical relations are more general than logical relations and variants like I-relations, they provide a framework within which these different classes can be compared. Here we begin by studying and comparing their closure properties (Prop. 5.6) with special attention to closure under composition.

The definition of pre-logical relations is not new. In [Sch87], Schoett uses a first-order version of algebraic relations which he calls *correspondences*, and he conjectures (p. 281) that for Henkin models, what we have called pre-logical relations (formulated as in Prop. 3.3) would be closed under composition and yield the Basic Lemma. In [Mit90], Mitchell makes the same suggestion, referring to Schoett and also crediting Abramsky and Plotkin, but as an assertion rather than a conjecture. The idea is not developed any further. An independent but apparently equivalent definition of pre-logical relations over cartesian closed categories is given in [PPS98] where they are called *lax logical relations*. It is shown that these compose and that the Basic Lemma holds, and an axiomatic account is provided. Earlier, a closely related notion called *L-relations* was defined in [KOPTT97] and shown to compose. Another related paper is [Rob96]. There appears to be no previous work on pre-logical relations that goes beyond observing that they compose and that the Basic Lemma holds. Another difference to [PPS98] and [KOPTT97] is that our treatment is elementary rather than categorical, and covers also combinatory logics.

2 Syntax and Semantics

We begin with $\lambda^{\to}$, the simply-typed lambda calculus having $\to$ as the only type constructor. Other type constructors will be considered in Sect. 10. We follow the terminology in [Mit96] for the most part, with slightly different notation.

Definition 2.1. *The set* $Types(B)$ *of* types *over a set* B *of* base types *(or* type constants*) is given by the grammar* $\sigma ::= b \mid \sigma \to \sigma$ *where* b *ranges over* B*. A* signature Σ *consists of a set* B *of type constants and a collection* C *of typed* term constants $c : \sigma$*.*

Let $\Sigma = \langle B, C\rangle$ be a signature. We assume familiarity with the usual notions of *context* $\Gamma = x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$ and Σ*-term* M of type σ over a context Γ, written $\Gamma \rhd M : \sigma$, with the meta-variable t reserved for lambda-free Σ-terms. If Γ is empty then we write simply $M : \sigma$. Capture-avoiding substitution $[N/x]M$ is as usual.

Definition 2.2. *A* Σ-applicative structure $\mathcal{A}$ *consists of:*

- *a* carrier *set* $[\![\sigma]\!]^{\mathcal{A}}$ *for each* $\sigma \in Types(B)$*;*
- *a function* $App^{\sigma,\tau}_{\mathcal{A}} : [\![\sigma \to \tau]\!]^{\mathcal{A}} \to [\![\sigma]\!]^{\mathcal{A}} \to [\![\tau]\!]^{\mathcal{A}}$ *for each* $\sigma, \tau \in Types(B)$*;*
- *an element* $[\![c]\!]^{\mathcal{A}} \in [\![\sigma]\!]^{\mathcal{A}}$ *for each term constant* $c : \sigma$ *in* Σ*.*

We drop the subscripts and superscripts when they are determined by the context. Two elements $f, g \in [\![\sigma \to \tau]\!]^{\mathcal{A}}$ *are said to be* extensionally equal *if* $App^{\sigma,\tau}_{\mathcal{A}}\, f\, x = App^{\sigma,\tau}_{\mathcal{A}}\, g\, x$ *for every* $x \in [\![\sigma]\!]^{\mathcal{A}}$*. A* Σ*-applicative structure is* extensional *when extensional equality coincides with identity.*

A Σ-combinatory algebra *is a* Σ*-applicative structure* $\mathcal{A}$ *that has elements* $K^{\sigma,\tau}_{\mathcal{A}} \in [\![\sigma \to (\tau \to \sigma)]\!]^{\mathcal{A}}$ *and* $S^{\rho,\sigma,\tau}_{\mathcal{A}} \in [\![(\rho \to \sigma \to \tau) \to (\rho \to \sigma) \to \rho \to \tau]\!]^{\mathcal{A}}$ *for each* $\rho, \sigma, \tau \in Types(B)$ *satisfying* $K^{\sigma,\tau}_{\mathcal{A}}\, x\, y = x$ *and* $S^{\rho,\sigma,\tau}_{\mathcal{A}}\, x\, y\, z = (x\, z)(y\, z)$*.*

An extensional combinatory algebra is called a Henkin model. *An applicative structure $\mathcal{A}$ is a* full type hierarchy *when $[\![\sigma \to \tau]\!]^{\mathcal{A}} = [\![\sigma]\!]^{\mathcal{A}} \to [\![\tau]\!]^{\mathcal{A}}$ for every $\sigma, \tau \in Types(B)$ and then it is obviously a Henkin model.*

In a combinatory algebra, we can extend the definition of lambda-free Σ-terms by allowing them to contain S and K; we call these *combinatory Σ-terms.*

A *Γ-environment* $\eta_{\mathcal{A}}$ assigns elements of an applicative structure $\mathcal{A}$ to variables, with $\eta_{\mathcal{A}}(x) \in [\![\sigma]\!]^{\mathcal{A}}$ for $x : \sigma$ in Γ. A lambda-free Σ-term $\Gamma \rhd t : \sigma$ is interpreted in a Σ-applicative structure $\mathcal{A}$ under a Γ-environment $\eta_{\mathcal{A}}$ in the obvious way, written $[\![\Gamma \rhd t : \sigma]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}}$, and this extends immediately to an interpretation of combinatory Σ-terms in combinatory algebras by interpreting K and S as $K_{\mathcal{A}}$ and $S_{\mathcal{A}}$. If t is closed then we write simply $[\![t : \sigma]\!]^{\mathcal{A}}$.

There are various ways of interpreting terms containing lambda abstraction in a combinatory algebra by "compiling" them to combinatory terms so that outermost β holds (see Prop. 2.4 below for what we mean by "outermost β"). In Henkin models, all these compilations yield the same result.

An axiomatic approach to interpreting lambda abstraction requires an applicative structure equipped with an interpretation function that satisfies certain minimal requirements — cf. the notion of *acceptable meaning function* in [Mit96].

Definition 2.3. *A* lambda Σ-applicative structure *consists of a Σ-applicative structure $\mathcal{A}$ together with a function $[\![\cdot]\!]^{\mathcal{A}}$ that maps any term $\Gamma \rhd M : \sigma$ and Γ-environment $\eta_{\mathcal{A}}$ over $\mathcal{A}$ to an element of $[\![\sigma]\!]^{\mathcal{A}}$, such that:*

- $[\![\Gamma \rhd x : \sigma]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}} = \eta_{\mathcal{A}}(x)$
- $[\![\Gamma \rhd c : \sigma]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}} = [\![c]\!]^{\mathcal{A}}$
- $[\![\Gamma \rhd M\ N : \tau]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}} = App_{\mathcal{A}}\ [\![\Gamma \rhd M : \sigma \to \tau]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}}\ [\![\Gamma \rhd N : \sigma]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}}$
- $[\![\Gamma \rhd \lambda x{:}\sigma.M : \sigma \to \tau]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}} = [\![\Gamma \rhd \lambda y{:}\sigma.[y/x]M : \sigma \to \tau]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}}$ *provided $y \notin \Gamma$*
- $[\![\Gamma \rhd M : \sigma]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}} = [\![\Gamma \rhd M : \sigma]\!]^{\mathcal{A}}_{\eta'_{\mathcal{A}}}$ *provided $\eta'_{\mathcal{A}}$ is a Γ-environment such that $\eta_{\mathcal{A}}(x) = \eta'_{\mathcal{A}}(x)$ for all $x \in \Gamma$*
- $[\![\Gamma, x{:}\sigma \rhd M : \tau]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}} = [\![\Gamma \rhd M : \tau]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}}$
- $[\![\Gamma, x{:}\sigma \rhd M : \tau]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}[x \mapsto [\![\Gamma \rhd N : \sigma]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}}]} = [\![\Gamma \rhd [N/x]M : \tau]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}}$

Proposition 2.4. *A lambda applicative structure $\mathcal{A}$ with $App_{\mathcal{A}}\ [\![\Gamma \rhd \lambda x{:}\sigma.M : \sigma \to \tau]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}}\ a = [\![\Gamma, x{:}\sigma \rhd M : \tau]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}[x \mapsto a]}$ amounts to a combinatory algebra, and vice versa.* □

Viewing a combinatory algebra as a lambda applicative structure involves interpreting lambda terms via compilation to combinatory terms.

3 Algebraic and Pre-logical Relations

We propose a weakening of the definition of logical relations which is closed under composition and which has most of the attractive properties of logical relations. First we change the two-way implication in the condition on functions to a one-way implication which requires preservation of the relation under application.

Definition 3.1. *Let $\mathcal{A}$ and $\mathcal{B}$ be Σ-applicative structures. An* algebraic relation *$\mathcal{R}$ over $\mathcal{A}$ and $\mathcal{B}$ is a family of relations $\{R^\sigma \subseteq [\![\sigma]\!]^\mathcal{A} \times [\![\sigma]\!]^\mathcal{B}\}_{\sigma \in Types(B)}$ such that:*

- *If $R^{\sigma\to\tau}(f,g)$ then $\forall a \in [\![\sigma]\!]^\mathcal{A}.\forall b \in [\![\sigma]\!]^\mathcal{B}.R^\sigma(a,b) \Rightarrow R^\tau(App_\mathcal{A}\, f\, a, App_\mathcal{B}\, g\, b)$.*
- *$R^\sigma([\![c]\!]^\mathcal{A}, [\![c]\!]^\mathcal{B})$ for every term constant $c : \sigma$ in Σ.*

In lambda applicative structures, we additionally require the relation to preserve lambda abstraction in a sense that is analogous to the definition of *admissible relation* in [Mit96]. First, we extend a family of relations to a relation on Γ-environments: $R^\Gamma(\eta_\mathcal{A}, \eta_\mathcal{B})$ if $R^\sigma(\eta_\mathcal{A}(x), \eta_\mathcal{B}(x))$ for every $x{:}\sigma$ in Γ.

Definition 3.2. *Let $\mathcal{A}$ and $\mathcal{B}$ be lambda Σ-applicative structures. A* pre-logical relation over $\mathcal{A}$ and $\mathcal{B}$ *is an algebraic relation $\mathcal{R}$ such that given Γ-environments $\eta_\mathcal{A}$ and $\eta_\mathcal{B}$ such that $R^\Gamma(\eta_\mathcal{A}, \eta_\mathcal{B})$, and a term $\Gamma, x : \sigma \rhd M : \tau$, if $R^\sigma(a,b)$ implies $R^\tau([\![\Gamma, x : \sigma \rhd M : \tau]\!]^\mathcal{A}_{\eta_\mathcal{A}[x\mapsto a]}, [\![\Gamma, x : \sigma \rhd M : \tau]\!]^\mathcal{B}_{\eta_\mathcal{B}[x\mapsto b]})$ for all $a \in [\![\sigma]\!]^\mathcal{A}$ and $b \in [\![\sigma]\!]^\mathcal{B}$, then $R^{\sigma\to\tau}([\![\Gamma \rhd \lambda x{:}\sigma.M : \sigma \to \tau]\!]^\mathcal{A}_{\eta_\mathcal{A}}, [\![\Gamma \rhd \lambda x{:}\sigma.M : \sigma \to \tau]\!]^\mathcal{B}_{\eta_\mathcal{A}})$.*

This amounts to defining pre-logical relations as simply the class of relations that make the Basic Lemma hold, as we shall see in Lemma 4.1 below. (Indeed, since the Basic Lemma for pre-logical relations is an equivalence rather than a one-way implication, an alternative at this point would be to take the conclusion of the Basic Lemma itself as the definition of pre-logical relations.)

A more appealing definition is obtained if we consider combinatory algebras, where the requirement above boils down to preservation of S and K:

Proposition 3.3. *Let $\mathcal{A}$ and $\mathcal{B}$ be Σ-combinatory algebras. An algebraic relation $\mathcal{R}$ over $\mathcal{A}$ and $\mathcal{B}$ is pre-logical iff $R(S^{\rho,\sigma,\tau}_\mathcal{A}, S^{\rho,\sigma,\tau}_\mathcal{B})$ and $R(K^{\sigma,\tau}_\mathcal{A}, K^{\sigma,\tau}_\mathcal{B})$ for all $\rho, \sigma, \tau \in Types(B)$.* □

If we incorporate S and K into the signature Σ, then pre-logical relations are just algebraic relations on combinatory algebras. One way of understanding the definition of pre-logical relations is that the reverse implication in the definition of logical relations is required to hold only for pairs of functions that are expressible by the same lambda term. For combinatory algebras these are exactly the pairs of functions that are denoted by the same combinatory term, and thus this requirement is captured by requiring the relation to contain S and K.

The use of the combinators S and K in the above proposition is in some sense arbitrary: the same result would be achieved by taking any other combinatory basis and changing the definition of combinatory algebra and the interpretation function accordingly. It would be straightforward to modify the definitions to accommodate other variants of lambda calculus, for instance λ_I for which a combinatory basis is B, C, I, S, or linear lambda calculi. For languages that include recursion, such as PCF, one would add a Y combinator.

As usual, the binary case of algebraic resp. pre-logical relations over $\mathcal{A}$ and $\mathcal{B}$ is derived from the unary case of *algebraic* resp. *pre-logical predicates* for the product structure $\mathcal{A}\times\mathcal{B}$. We omit the obvious definitions. For most results about pre-logical relations below there are corresponding results about pre-logical predicates and about algebraic relations and predicates over applicative structures. Similar comments apply to n-ary relations for $n > 2$.

The fact that pre-logicality is strictly weaker than logicality is demonstrated by the following examples which also provide a number of general methods for defining pre-logical relations.

Example 3.4. For any signature Σ and lambda Σ-applicative structure, the predicate $\mathcal{P}$ defined by

$$P^\sigma(v) \Leftrightarrow v \text{ is the value of a closed } \Sigma\text{-term } M : \sigma$$

is a pre-logical predicate over $\mathcal{A}$. (In fact, $\mathcal{P}$ is the *least* such — see Prop. 5.7 below.) Now, consider the signature Σ containing the type constant *nat* and term constants $0 : nat$ and $succ : nat \to nat$ and let $\mathcal{A}$ be the full type hierarchy over $\mathbb{N}$ where 0 and *succ* are interpreted as usual. $\mathcal{P}$ is not a logical predicate over $\mathcal{A}$: any function $f \in [\![nat \to nat]\!]^{\mathcal{A}}$, including functions that are not lambda definable, takes values in P to values in P and so must itself be in P. □

Example 3.5. The identity relation on a lambda applicative structure is a pre-logical relation but it is logical iff the structure is extensional. □

Example 3.6. A *Σ-homomorphism* between lambda Σ-applicative structures $\mathcal{A}$ and $\mathcal{B}$ is a type-indexed family of functions $\{h^\sigma : [\![\sigma]\!]^{\mathcal{A}} \to [\![\sigma]\!]^{\mathcal{B}}\}_{\sigma \in Types(B)}$ such that for any term constant $c : \sigma$ in Σ, $h^\sigma([\![c]\!]^{\mathcal{A}}) = [\![c]\!]^{\mathcal{B}}$, $h^\tau(App_{\mathcal{A}}^{\sigma,\tau}\ f\ a) = App_{\mathcal{B}}^{\sigma,\tau}\, h^{\sigma\to\tau}(f)\ h^\sigma(a)$ and $h^{\sigma\to\tau}([\![\Gamma \rhd \lambda x{:}\sigma.M : \sigma \to \tau]\!]_{\eta_{\mathcal{A}}}^{\mathcal{A}}) = [\![\Gamma \rhd \lambda x{:}\sigma.M : \sigma \to \tau]\!]_{h(\eta_{\mathcal{A}})}^{\mathcal{B}}$ where $h(\eta_{\mathcal{A}}) = \{x \mapsto h^\sigma(\eta_{\mathcal{A}}(x))\}$ for all $x{:}\sigma$ in Γ. Any Σ-homomorphism is a pre-logical relation. In particular, interpretation of terms in a lambda applicative structure with respect to an environment, viewed as a relation from the lambda applicative structure of terms, is a pre-logical relation but is not in general a logical relation. □

Example 3.7. Let $\mathcal{A}$ and $\mathcal{B}$ be lambda applicative structures and define $R^\sigma \subseteq [\![\sigma]\!]^{\mathcal{A}} \times [\![\sigma]\!]^{\mathcal{B}}$ by $R^\sigma(a, b)$ for $a \in [\![\sigma]\!]^{\mathcal{A}}, b \in [\![\sigma]\!]^{\mathcal{B}}$ iff there is a closed term $M : \sigma$ such that $[\![M : \sigma]\!]^{\mathcal{A}} = a$ and $[\![M : \sigma]\!]^{\mathcal{B}} = b$. This is a pre-logical relation but it is not in general a logical relation. Generalizing: the inverse of any pre-logical relation is obviously pre-logical and according to Prop. 5.5 below the composition of any two pre-logical relations is pre-logical. Then observe that the above relation is just the composition of closed term interpretation in $\mathcal{B}$ (which is pre-logical according to Example 3.6) and the inverse of closed term interpretation in $\mathcal{A}$. □

Example 3.8. Plotkin's *I-relations* [Plo80] give rise to pre-logical relations. The family of relations on the full type hierarchy consisting of the tuples which are in a given I-relation at a given world (alternatively, at all worlds) is a pre-logical relation which is not in general a logical relation. □

A related example concerning Kripke logical relations with varying arity [JT93] is postponed to Sect. 6.

Example 3.9. Let $\mathcal{A}$ be an applicative structure. Given order relations R^b on $[\![b]\!]^{\mathcal{A}}$ for each base type b, we can define Gandy's *hereditarily strict monotonic functionals* [Gan80] as the equivalence classes of those elements of $\mathcal{A}$ which are

self-related with respect to the following inductively defined family of relations on $\mathcal{A} \times \mathcal{A}$:

$$\begin{aligned} R^{\sigma\to\tau}(f,g) \text{ iff } \forall a \in [\![\sigma]\!]^{\mathcal{A}}.\forall b \in [\![\sigma]\!]^{\mathcal{A}}.& \\ R^{\sigma}(a,b) \Rightarrow (f \neq g \Rightarrow (R^{\tau} \setminus \Delta^{\tau})(App_{\mathcal{A}}\ f\ a, App_{\mathcal{A}}\ g\ a)& \\ \wedge& \\ a \neq b \Rightarrow ((R^{\tau} \setminus \Delta^{\tau})(App_{\mathcal{A}}\ f\ a, App_{\mathcal{A}}\ f\ b)& \\ \wedge (R^{\tau} \setminus \Delta^{\tau})(App_{\mathcal{A}}\ g\ a, App_{\mathcal{A}}\ g\ b)))& \end{aligned}$$

(This defines simultaneously at each type both the class of functionals we are interested in and the order relation itself.) This method defines a pre-logical relation (with respect to λ_I) which is not in general a logical relation. □

4 The Basic Lemma

We will now consider the extension of the Basic Lemma to pre-logical relations. In contrast to Lemma 1.2, we get a two-way implication which says that the requirements on pre-logical relations are exactly strong enough to ensure that the Basic Lemma holds. The reverse implication fails for logical relations as Example 3.4 shows (for logical predicates).

Lemma 4.1 (Basic Lemma for pre-logical relations). *Let $\mathcal{R} = \{R^{\sigma} \subseteq [\![\sigma]\!]^{\mathcal{A}} \times [\![\sigma]\!]^{\mathcal{B}}\}_{\sigma \in Types(B)}$ be a family of relations over lambda Σ-applicative structures $\mathcal{A}$ and $\mathcal{B}$. Then $\mathcal{R}$ is a pre-logical relation iff for all Γ-environments $\eta_{\mathcal{A}}, \eta_{\mathcal{B}}$ such that $R^{\Gamma}(\eta_{\mathcal{A}}, \eta_{\mathcal{B}})$ and every Σ-term $\Gamma \rhd M : \sigma$, $R^{\sigma}([\![\Gamma \rhd M : \sigma]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}}, [\![\Gamma \rhd M : \sigma]\!]^{\mathcal{B}}_{\eta_{\mathcal{B}}})$.* □

The "only if" direction of this result is the analogue in our setting of the general version of the Basic Lemma in [Mit96], cf. Lemma 1.2 above for the case of Henkin models, but where $\mathcal{R}$ is only required to be pre-logical.

The Basic Lemma is intimately connected with the concept of lambda definability. This is most apparent in the unary case:

Lemma 4.2 (Basic Lemma for pre-logical predicates). *Let $\mathcal{P} = \{P^{\sigma} \subseteq [\![\sigma]\!]^{\mathcal{A}}\}_{\sigma \in Types(B)}$ be a family of predicates over a lambda Σ-applicative structure $\mathcal{A}$. Then $\mathcal{P}$ is a pre-logical predicate iff it is closed under lambda definability: $P^{\Gamma}(\eta)$ and $\Gamma \rhd M : \sigma$ implies $P^{\sigma}([\![\Gamma \rhd M : \sigma]\!]^{\mathcal{A}}_{\eta})$.* □

5 Properties of Pre-logical Relations

A logical relation on lambda applicative structures is pre-logical provided it is admissible in the following sense.

Definition 5.1 ([Mit96]). *A logical relation $\mathcal{R}$ on lambda applicative structures $\mathcal{A}$ and $\mathcal{B}$ is* admissible *if given Γ-environments $\eta_{\mathcal{A}}$ and $\eta_{\mathcal{B}}$ such that $R^{\Gamma}(\eta_{\mathcal{A}}, \eta_{\mathcal{B}})$, and terms $\Gamma, x{:}\sigma \rhd M, N : \tau$,*

$$\forall a \in [\![\sigma]\!]^{\mathcal{A}}, b \in [\![\sigma]\!]^{\mathcal{B}}. R^{\sigma}(a,b) \Rightarrow R^{\tau}([\![\Gamma, x{:}\sigma \rhd M : \tau]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}[x\mapsto a]}, [\![\Gamma, x{:}\sigma \rhd N : \tau]\!]^{\mathcal{B}}_{\eta_{\mathcal{B}}[x\mapsto b]})$$

implies

$$\forall a \in [\![\sigma]\!]^{\mathcal{A}}, b \in [\![\sigma]\!]^{\mathcal{B}}.R^{\sigma}(a,b) \Rightarrow R^{\tau}(App_{\mathcal{A}}\ [\![\Gamma \rhd \lambda x{:}\sigma.M : \sigma \to \tau]\!]^{\mathcal{A}}_{\eta_{\mathcal{A}}}\ a, \\ App_{\mathcal{B}}\ [\![\Gamma \rhd \lambda x{:}\sigma.N : \sigma \to \tau]\!]^{\mathcal{B}}_{\eta_{\mathcal{B}}}\ b)$$

Proposition 5.2. *Any admissible logical relation on lambda applicative structures is a pre-logical relation.* □

Corollary 5.3. *Any logical relation on combinatory algebras is a pre-logical relation.* □

To understand why the composition of logical relations $\mathcal{R}$ over $\mathcal{A}$ and $\mathcal{B}$ and $\mathcal{S}$ over $\mathcal{B}$ and $\mathcal{C}$ might not be a logical relation, it is instructive to look at examples. When composition fails, the problem is often that the interpretation of some function type in $\mathcal{B}$ has "too few values". But even if we take logical relations over full type hierarchies, where all possible values of function types are present, composition can fail:

Example 5.4. Let Σ contain just two type constants, b and b'. Consider three full type hierarchies $\mathcal{A}, \mathcal{B}, \mathcal{C}$ which interpret b and b' as follows: $[\![b]\!]^{\mathcal{A}} = \{*\} = [\![b']\!]^{\mathcal{A}}$; $[\![b]\!]^{\mathcal{B}} = \{*\}$ and $[\![b']\!]^{\mathcal{B}} = \{\circ, \bullet\}$; $[\![b]\!]^{\mathcal{C}} = \{\circ, \bullet\} = [\![b']\!]^{\mathcal{C}}$. Let $\mathcal{R}$ be the logical relation over $\mathcal{A}$ and $\mathcal{B}$ induced by $R^b = \{\langle *, *\rangle\}$ and $R^{b'} = \{\langle *, \circ\rangle, \langle *, \bullet\rangle\}$ and let $\mathcal{S}$ be the logical relation over $\mathcal{B}$ and $\mathcal{C}$ induced by $S^b = \{\langle *, \circ\rangle, \langle *, \bullet\rangle\}$ and $S^{b'} = \{\langle \circ, \circ\rangle, \langle \bullet, \bullet\rangle\}$. $\mathcal{S} \circ \mathcal{R}$ is not a logical relation because it does not relate the identity function in $[\![b]\!]^{\mathcal{A}} \to [\![b']\!]^{\mathcal{A}}$ to the identity function in $[\![b]\!]^{\mathcal{C}} \to [\![b']\!]^{\mathcal{C}}$. The problem is that the only two functions in $[\![b]\!]^{\mathcal{B}} \to [\![b']\!]^{\mathcal{B}}$ are $\{* \mapsto \circ\}$ and $\{* \mapsto \bullet\}$, and $\mathcal{S}$ does not relate these to the identity in $\mathcal{C}$. □

Proposition 5.5. *The composition $\mathcal{S} \circ \mathcal{R}$ of pre-logical relations $\mathcal{R}$ over $\mathcal{A}, \mathcal{B}$ and $\mathcal{S}$ over $\mathcal{B}, \mathcal{C}$ is a pre-logical relation over $\mathcal{A}, \mathcal{C}$.* □

Composition is definable in terms of product, intersection and projection:

$$\mathcal{S} \circ \mathcal{R} = \pi_{1,3}(\mathcal{A} \times \mathcal{S}\ \cap\ \mathcal{R} \times \mathcal{C})$$

Closure of pre-logical relations under these operations is a more basic property than closure under composition, and is not specific to binary relations. We have:

Proposition 5.6. *Pre-logical relations are closed under intersection, product, projection, restriction to a substructure, permutation and $\forall$. (Here, if $\mathcal{R} \subseteq \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$ then $\forall\mathcal{R} \subseteq \mathcal{A}_2 \times \cdots \times \mathcal{A}_n$ is defined by $(\forall\mathcal{R})^{\sigma} = \{\langle a_2, \ldots, a_n\rangle \mid \forall a_1 \in [\![\sigma]\!]^{\mathcal{A}_1}.\langle a_1, a_2, \ldots, a_n\rangle \in \mathcal{R}^{\sigma}\}$.) Logical relations are closed under product, permutation and $\forall$ but not under intersection, projection or restriction to a substructure.* □

A consequence of closure under intersection is that given a property P of relations that is preserved under intersection, there is always a *least* pre-logical relation satisfying P. We then have the following lambda-definability result (recall Example 3.4 above):

Proposition 5.7. *The least pre-logical predicate over a lambda Σ-applicative structure contains exactly those elements that are the values of closed Σ-terms.* □

In a signature with no term constants, a logical relation may be constructed by defining a relation R on base types and using the definition to *"lift" R* inductively to higher types. The situation is different for pre-logical relations: there are in general many pre-logical liftings of a given R, one being its lifting to a logical relation (provided this gives an admissible relation). But since the property of lifting a given R is preserved under intersection, the least pre-logical lifting of R is also a well-defined relation. Similarly for the least pre-logical *extension* of a given family of relations, for any signature. Lifting/extending a given family of relations to a logical relation is problematic for signatures containing higher-order term constants.

It is easy to see that pre-logical relations are not closed under union. And even in a signature with no term constants, the class of pre-logical relations that lift a given relation R on base types cannot be endowed with a lattice structure in general. But the only logical relation in this class is one of its maximal elements under inclusion.

6 Kripke Logical Relations with Varying Arity

Definition 6.1 ([JT93]). *Let $\mathcal{C}$ be a small category of sets and let $\mathcal{A}$ be a Henkin model. A* Kripke logical relation with varying arity *(*KLRwVA *for short) over $\mathcal{A}$ is a family of relations R^w_σ indexed by objects w of $\mathcal{C}$ and types σ of Types(B), where the elements of R^w_σ are tuples of elements from $[\![\sigma]\!]^{\mathcal{A}}$ indexed by the elements of w, such that:*

- *If $f : v \to w$ is a map in $\mathcal{C}$ and $R^w_\sigma\langle a_j\rangle_{j\in w}$ then $R^v_\sigma\langle a_{f(i)}\rangle_{i\in v}$.*
- *$R^w_{\sigma\to\tau}\langle g_j\rangle_{j\in w}$ iff $\forall f : v \to w.\forall\langle a_i\rangle_{i\in v}.R^v_\sigma\langle a_i\rangle_{i\in v} \Rightarrow R^v_\tau\langle App_{\mathcal{A}}\, g_{f(i)}\, a_i\rangle_{i\in v}$*
- *$R^w_\sigma\langle[\![c]\!]^{\mathcal{A}}\rangle_{j\in w}$ for every term constant $c : \sigma$ in Σ.*

(This extends Jung and Tiuryn's definition to take term constants into account.)

KLRwVAs give rise to pre-logical relations in a similar way to I-relations, see Example 3.8: the family of relations consisting of the w-indexed tuples which are in a given KLRwVA at world w is a pre-logical relation which is not in general a logical relation, and those elements which are invariant under a given KLRwVA (i.e. $a \in [\![\sigma]\!]^{\mathcal{A}}$ such that $R^w_\sigma\langle a\rangle_{j\in w}$ for all w) also form a pre-logical predicate. More interesting is the fact that every pre-logical relation can be obtained in this way. We give the unary case; the binary and n-ary cases are obtained by instantiating to product structures.

Proposition 6.2. *Let $\mathcal{P} = \{P^\sigma \subseteq [\![\sigma]\!]^{\mathcal{A}}\}_{\sigma\in Types(B)}$ be a family of predicates over a Henkin structure $\mathcal{A}$. $\mathcal{P}$ is a pre-logical predicate iff it is the set of elements of $\mathcal{A}$ which are invariant under some KLRwVA.* □

7 Pre-logical Relations via Composition of Logical Relations

Our weakening of the definition of logical relations may appear to be *ad hoc*, but for extensional structures it turns out to be the minimal weakening that is closed under composition. There are variants of this result for several different classes of models. We give the version for Henkin models.

Proposition 7.1. *Let $\mathcal{A}$ and $\mathcal{B}$ be Henkin models and let $\mathcal{R}$ be a pre-logical relation over $\mathcal{A}$ and $\mathcal{B}$. Then $\mathcal{R}$ factors into a composition of three logical relations over Henkin models.*

Proof idea. Let $\mathcal{A}[X]$ and $\mathcal{B}[X]$ be obtained by adding indeterminates to $\mathcal{A}$ and $\mathcal{B}$ respectively. $\mathcal{R}$ is the composition of: the embedding of $\mathcal{A}$ in $\mathcal{A}[X]$; a logical relation $\mathcal{R}[X]$ on $\mathcal{A}[X]$ and $\mathcal{B}[X]$; and the inverse of the embedding of $\mathcal{B}$ in $\mathcal{B}[X]$. □

Corollary 7.2. *The class of pre-logical relations on Henkin models is the closure under composition of the class of logical relations on such structures.* □

This gives the following lambda-definability result:

Corollary 7.3. *Let $\mathcal{A}$ be a Henkin model and $a \in [\![\sigma]\!]^{\mathcal{A}}$. Then $\langle a, a\rangle$ belongs to all relations over $\mathcal{A} \times \mathcal{A}$ obtained by composing logical relations iff $a = [\![M : \sigma]\!]^{\mathcal{A}}$ for some closed Σ-term $M : \sigma$.* □

For non-extensional structures the notion of pre-logical relations is not the minimal weakening that gives closure under composition. The following variant is the minimal weakening for this case.

Definition 7.4. *An algebraic relation is* extensional *if whenever $R^{\sigma\to\tau}(f, g)$, f is extensionally equal to f' and g is extensionally equal to g', we have $R^{\sigma\to\tau}(f', g')$.*

All pre-logical relations over extensional structures are automatically extensional, and all logical relations over applicative structures (even non-extensional ones) are automatically extensional as well.

Proposition 7.5. *Let $\mathcal{A}$ and $\mathcal{B}$ be combinatory algebras and let $\mathcal{R}$ be an extensional pre-logical relation over $\mathcal{A}$ and $\mathcal{B}$. Then $\mathcal{R}$ factors into a composition of three logical relations.* □

Corollary 7.6. *The class of extensional pre-logical relations on combinatory algebras is the closure under composition of the class of logical relations on such structures.* □

These results may suggest that our definition of pre-logical relations on non-extensional structures should be strengthened by requiring the relation to be extensional, but this would make the reverse implication of the Basic Lemma fail. So although the notion of extensional pre-logical relations is the minimal weakening that gives closure under composition, these are stronger than necessary to give the Basic Lemma.

8 Representation Independence and Data Refinement

Logical relations have been applied to explain the fact that the behaviour of programs does not depend on the way that data types are represented, but only on what can be observed about them using the operations that are provided. "Behaviour of programs" is captured by the notion of observational equivalence.

Definition 8.1. *Let $\mathcal{A}$ and $\mathcal{B}$ be lambda Σ-applicative structures and let OBS, the* observable types*, be a subset of Types(B). Then $\mathcal{A}$ is* observationally finer *than $\mathcal{B}$ with respect to OBS, written $\mathcal{A} \leq_{OBS} \mathcal{B}$, if for any two closed terms $M, N : \sigma$ for $\sigma \in OBS$ such that $[\![M : \sigma]\!]^{\mathcal{A}} = [\![N : \sigma]\!]^{\mathcal{A}}$ we have $[\![M : \sigma]\!]^{\mathcal{B}} = [\![N : \sigma]\!]^{\mathcal{B}}$. $\mathcal{A}$ and $\mathcal{B}$ are* observationally equivalent *with respect to OBS, written $\mathcal{A} \equiv_{OBS} \mathcal{B}$, if $\mathcal{A} \leq_{OBS} \mathcal{B}$ and $\mathcal{B} \leq_{OBS} \mathcal{A}$.*

Usually *OBS* are the "built-in" types for which equality is decidable, for instance *bool* and/or *nat*. Then $\mathcal{A}$ and $\mathcal{B}$ are observationally equivalent iff it is not possible to distinguish between them by performing computational experiments.

Mitchell gives the following representation independence result:

Theorem 8.2 ([Mit96]). *Let Σ be a signature that includes a type constant nat, and let $\mathcal{A}$ and $\mathcal{B}$ be Henkin models, with $[\![nat]\!]^{\mathcal{A}} = [\![nat]\!]^{\mathcal{B}} = \mathbb{N}$. If there is a logical relation $\mathcal{R}$ over $\mathcal{A}$ and $\mathcal{B}$ with R^{nat} the identity relation on natural numbers, then $\mathcal{A} \equiv_{\{nat\}} \mathcal{B}$. Conversely, if $\mathcal{A} \equiv_{\{nat\}} \mathcal{B}$, Σ provides a closed term for each element of $\mathbb{N}$, and Σ only contains first-order functions, then there is a logical relation $\mathcal{R}$ over $\mathcal{A}$ and $\mathcal{B}$ with R^{nat} the identity relation.* □

The following example (Exercise 8.5.6 in [Mit96]) shows that the requirement that Σ contains only first-order functions is necessary.

Example 8.3. Let Σ have type constant *nat* and term constants $0, 1, 2, \ldots : nat$ and $f : (nat \to nat) \to nat$. Let $\mathcal{A}$ be the full type hierarchy over $[\![nat]\!]^{\mathcal{A}} = \mathbb{N}$ with $0, 1, 2, \ldots$ interpreted as usual and $[\![f]\!]^{\mathcal{A}}(g) = 0$ for all $g : \mathbb{N} \to \mathbb{N}$. Let $\mathcal{B}$ be like $\mathcal{A}$ but with $[\![f]\!]^{\mathcal{B}}(g) = 0$ if g is computable and $[\![f]\!]^{\mathcal{B}}(g) = 1$ otherwise. Since the difference between $\mathcal{A}$ and $\mathcal{B}$ cannot be detected by evaluating terms, $\mathcal{A} \equiv_{\{nat\}} \mathcal{B}$. But there is no logical relation over $\mathcal{A}$ and $\mathcal{B}$ which is the identity relation on *nat*: if $\mathcal{R}$ is logical then $R^{nat \to nat}(g, g)$ for any $g : \mathbb{N} \to \mathbb{N}$, and then $R^{nat}(App_{\mathcal{A}}\ [\![f]\!]^{\mathcal{A}}\ g, App_{\mathcal{B}}\ [\![f]\!]^{\mathcal{B}}\ g)$, which gives a contradiction if g is non-computable. □

We will strengthen this result by showing that pre-logical relations characterize observational equivalence for *all* signatures. We also generalize to arbitrary sets of observable types but this is much less significant. This characterization is obtained as a corollary of the following theorem which is a strengthening of Lemma 8.2.17 in [Mit96], again made possible by using pre-logical relations in place of logical relations.

Theorem 8.4. *Let $\mathcal{A}$ and $\mathcal{B}$ be lambda Σ-applicative structures and let $OBS \subseteq Types(B)$. Then $\mathcal{A} \leq_{OBS} \mathcal{B}$ iff there exists a pre-logical relation over $\mathcal{A}$ and $\mathcal{B}$ which is a partial function on OBS.* □

(Mitchell's Lemma 8.2.17 is the "if" direction for Henkin models where $OBS = Types(B)$ but $\mathcal{R}$ is required to be logical rather than just pre-logical.)

Corollary 8.5. *Let $\mathcal{A}$ and $\mathcal{B}$ be lambda Σ-applicative structures and let $OBS \subseteq Types(B)$. Then $\mathcal{A} \equiv_{OBS} \mathcal{B}$ iff there exists a pre-logical relation over $\mathcal{A}$ and $\mathcal{B}$ which is a partial function on OBS in both directions.* □

Corollary 8.6. *Let Σ include a type constant nat and let $\mathcal{A}$ and $\mathcal{B}$ be lambda Σ-applicative structures with $[\![nat]\!]^{\mathcal{A}} = [\![nat]\!]^{\mathcal{B}} = \mathbb{N}$ such that Σ provides a closed term for each element of $\mathbb{N}$. There is a pre-logical relation $\mathcal{R}$ over $\mathcal{A}$ and $\mathcal{B}$ with R^{nat} the identity relation on natural numbers iff $\mathcal{A} \equiv_{\{nat\}} \mathcal{B}$.* □

Example 8.7. Revisiting Example 8.3, the pre-logical relation constructed in Example 3.7 has the required property, and it does not relate non-computable functions since they are not lambda definable. □

In accounts of data refinement in terms of logical relations such as Sect. 2 of [Ten94], the fact that logical relations do not compose conflicts with the experience that data refinements *do* compose in real life. Example 5.4 can be embellished to give refinements between data structures like lists and sets for which the logical relations underlying the refinement steps do not compose to give a logical relation, yet the data refinements involved do compose at an intuitive level. This failure to justify the soundness of *stepwise* refinement is a serious flaw. If pre-logical relations are used in place of logical relations, then the fact that the composition of pre-logical relations is again a pre-logical relation (Prop. 5.5) explains why stepwise refinement is sound. This opens the way to further development of the foundations of data refinement along the lines of [ST88], but we leave this to a separate future paper, see Sect. 11.

9 Other Applications

There are many other applications of logical relations. Take for instance the proof of strong normalization of $\lambda^{\rightarrow}$ in [Mit96]: one defines an admissible logical predicate on a lambda applicative structure of terms by lifting the predicate on base types consisting of the strongly normalizing terms to higher types, proves that the predicate implies strong normalization, and then applies the general version of the Basic Lemma to give the result. The pattern for proofs of confluence, completeness of leftmost reduction, etc., is the same, sometimes with logical relations in place of logical predicates. There are also constructions that do not involve the Basic Lemma because the relations defined are not logical relations, but that include proofs following the same lines as the proof of the Basic Lemma. Examples include Gandy's proof that the hereditarily strict monotonic functionals model λ_I terms [Gan80], Plotkin's proof that lambda terms satisfy any I-relation [Plo80], and Jung and Tiuryn's proof that lambda terms satisfy any KLRwVA at each arity (Theorem 3 of [JT93]).

All of these can be cast into a common mould by using pre-logical relations. If a relation or predicate on a lambda applicative structure is logical and admissible,

then it is pre-logical, and then the Basic Lemma for pre-logical relations gives the result. Plotkin's, Jung and Tiuryn's, and Gandy's relations can be shown to be pre-logical (in Gandy's case with respect to λ_I), see Example 3.8, Sect. 6 and Example 3.9 respectively, and then the application of the Basic Lemma for pre-logical relations gives the result in these cases as well. In each case, however, the interesting part of the proof is not the application of the Basic Lemma (or the argument that replaces its application in the case of Gandy, Plotkin, and Jung and Tiuryn) but rather the construction of the relation and the proof of its properties. The point of the analysis is not to say that this view makes the job easier but rather to bring forward the common pattern in all of these proofs, which is suggestive of a possible methodology for such proofs.

Definition 9.1. *A family of binary relations* $\{R^\sigma \subseteq [\![\sigma]\!]^{\mathcal{A}} \times [\![\sigma]\!]^{\mathcal{A}}\}_{\sigma \in Types(B)}$ *over a* Σ*-applicative structure* $\mathcal{A}$ *is a* partial equivalence relation *(abbreviated* PER*) if it is symmetric and transitive for each type.*

Proposition 9.2. *Let* $\mathcal{R}$ *be a PER on a* Σ*-applicative structure* $\mathcal{A}$ *which is algebraic. Define the* quotient *of* $\mathcal{A}$ *by* $\mathcal{R}$*, written* $\mathcal{A}/\mathcal{R}$*, as follows:*

- $[\![\sigma]\!]^{\mathcal{A}/\mathcal{R}} = [\![\sigma]\!]^{\mathcal{A}}/R^\sigma$*, i.e. the set of* $\mathcal{R}$*-equivalence classes of objects* $a \in [\![\sigma]\!]^{\mathcal{A}}$ *such that* $R^\sigma(a, a)$*.*
- $App^{\sigma,\tau}_{\mathcal{A}/\mathcal{R}}\,[f]_{\mathcal{A}/\mathcal{R}}\,[a]_{\mathcal{A}/\mathcal{R}} = [App^{\sigma,\tau}_{\mathcal{A}}\,f\,a]_{\mathcal{A}/\mathcal{R}}$
- $[\![c]\!]^{\mathcal{A}/\mathcal{R}} = [c]_{\mathcal{A}/\mathcal{R}}$ *for each term constant* $c : \sigma$ *in* Σ*.*

Then:

1. *Let* $\mathcal{A}$ *be a lambda applicative structure. Then* $\mathcal{A}/\mathcal{R}$ *is a lambda applicative structure iff* $\mathcal{R}$ *is pre-logical.*
2. *Let* $\mathcal{A}$ *be a combinatory algebra. Then* $\mathcal{A}/\mathcal{R}$ *is a combinatory algebra iff* $\mathcal{R}$ *is pre-logical.*
3. $\mathcal{A}/\mathcal{R}$ *is an extensional applicative structure iff its restriction to the substructure of* $\mathcal{A}$ *consisting of the elements in* $Dom(\mathcal{R})$ *is a logical relation.* □

The last part of the above proposition says that one application of logical relations, that is their use in obtaining extensional structures by quotienting non-extensional structures — the so-called *extensional collapse* — requires a relation that is logical (on a substructure) rather than merely pre-logical.

The above proposition allows us to prove completeness for different classes of structures using the traditional technique of quotienting an applicative structure of terms by a suitable relation defined by provability in a calculus. For non-extensional structures, this is not possible using logical relations because the relation defined by provability is pre-logical or algebraic rather than logical.

10 Beyond $\lambda^\to$ and Applicative Structures

Up to now we have been working in $\lambda^\to$, the simplest version of typed lambda calculus. We will now briefly indicate how other type constructors could be treated so as to obtain corresponding results for extended languages.

As a template, we shall discuss the case of product types. The syntax of types is extended by adding the type form $\sigma \times \tau$ and the syntax of terms is extended by adding pairing $\langle M, N \rangle$ and projections $\pi_1\ M$ and $\pi_2\ M$. If we regard these as additional term constants in the signature, e.g. $\langle \cdot, \cdot \rangle : \sigma \to \tau \to \sigma \times \tau$ for all σ, τ, rather than as new term forms, then the definition of pre-logical relations remains the same: the condition on term constants says that e.g. $R^{\sigma \to \tau \to \sigma \times \tau}(\llbracket \langle \cdot, \cdot \rangle \rrbracket^{\mathcal{A}}, \llbracket \langle \cdot, \cdot \rangle \rrbracket^{\mathcal{B}})$ and this is all that is required. For models that satisfy surjective pairing, this implies the corresponding condition on logical relations, namely

– $R^{\sigma \times \tau}(a, b)$ iff $R^{\sigma}(\pi_1\ a, \pi_1\ b)$ and $R^{\tau}(\pi_2\ a, \pi_2\ b)$.

The treatment of sum types $\sigma + \tau$ is analogous.

A type constructor that has received less attention in the literature is (finite) powerset, $\mathcal{P}(\sigma)$. For lack of space we do not propose a specific language of terms to which one could apply the paradigm suggested above, but we claim that the notion of pre-logical relations over full type hierarchies would be extended to powersets by the addition of the following condition:

– $R^{\mathcal{P}(\sigma)}(\alpha, \beta)$ iff $\forall a \in \alpha. \exists b \in \beta. R^{\sigma}(a, b)$ and $\forall b \in \beta. \exists a \in \alpha. R^{\sigma}(a, b)$.

Note that this is the same pattern used in defining bisimulations. The extension for other kinds of models remains a topic for future work.

Various other kinds of types can be considered, including inductive and coinductive data types, universally and existentially quantified types, and various flavours of dependent types. We have not yet considered these in any detail, but we are confident that for any of them, one could take any existing treatment of logical relations and modify it by weakening the condition on functions as above without sacrificing the Basic Lemma. We expect that this would even yield improved results as it has above, but this is just speculation.

A different dimension of generalization is to consider models having additional structure — e.g. Kripke applicative structures [MM91], pre-sheaf models or cartesian closed categories — for which logical relations have been studied. We have not yet examined the details of this generalization but it appears that a corresponding weakening of the definition would lead to analogues of the results above, cf. [PPS98].

11 Conclusions and Directions for Future Work

Our feeling is that by introducing the notion of pre-logical relation we have, metaphorically and a little immodestly, removed a "blind spot" in the existing intuition of the use and scope of logical relations and related techniques. This is not to say that some specialists in the field have not previously contemplated generalizations similar to ours, but they have not carried the investigation far enough. We believe that in this paper we have exposed very clearly the fact that in many situations the use of logical relations is unnecessarily restrictive. Using pre-logical relations instead, we get improved statements of some results (e.g.

Theorem 8.4 and its corollaries), we encompass constructions that had previously escaped the logical paradigm (e.g. Example 3.9), and we isolate the necessary and sufficient hypotheses for many arguments to go through (e.g. Lemma 4.1). We have given several characterizations of pre-logical relations, summarized in the following theorem (for the unary case):

Theorem 11.1. *Let $\mathcal{P} = \{P^\sigma \subseteq [\![\sigma]\!]^{\mathcal{A}}\}_{\sigma \in Types(B)}$ be a family of predicates over a Henkin structure $\mathcal{A}$. The following are equivalent.*

1. *$\mathcal{P}$ is a pre-logical predicate.*
2. *$\mathcal{P}$ is closed under lambda definability.*
3. *$\mathcal{P}$ is the set of elements of $\mathcal{A}$ which are invariant under some KLRwVA.*
4. *$\mathcal{P}$ is the set of elements of $\mathcal{A}$ that are invariant under the composition of (three) logical relations.* □

The fact that there are so many conceptually independent ways of defining the same class of relations suggests that it is a truly intrinsic notion. Notice that Thm. 11.1(3) gives an inductive flavour to this concept which is not explicit in the definition of pre-logical relation; this apparent lack has been regarded as a weakness of the concept, see e.g. p. 428-429 of [Mit90].

Throughout the paper we have indicated possible directions of future investigation, e.g. with respect to richer type theories. It is plausible that sharper characterizations of representation independence will appear in many different type contexts.

But probably the area where the most benefits will be achieved will be that of the foundations of data refinement. Here we think that a more comprehensive explanation of data refinement would be obtained by combining an account in terms of pre-logical relations with the first-order algebraic treatment in [ST88] which we would expect to extend smoothly to higher-order. Among other improvements, this would result in a non-symmetric refinement relation, giving a better fit with the real-life phenomenon being modelled.

There is a vast literature on logical relations in connection with areas like parametricity, abstract interpretation, etc. A treatment of these topics in terms of pre-logical relations is likely to be as fruitful and illuminating as it has proved to be for the classical example of simply-typed lambda calculus presented here.

Acknowledgements: Thanks to Samson Abramsky, Jo Hannay, Martin Hofmann, Andrew Kennedy, Yoshiki Kinoshita, John Mitchell, Peter O'Hearn, Gordon Plotkin, John Power and Ian Stark for helpful comments. This work has been partially supported by EPSRC grant GR/K63795, an SOEID/RSE Support Research Fellowship, the ESPRIT-funded CoFI and TYPES working groups, and a MURST'97 grant.

References

[Abr90] S. Abramsky. Abstract interpretation, logical relations, and Kan extensions. *Journal of Logic and Computation* 1:5–40 (1990).

[Gan80] R. Gandy. Proofs of strong normalization. In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 457–477. Academic Press (1980).

[JT93] A. Jung and J. Tiuryn. A new characterization of lambda definability. *Proc. TLCA'93*. Springer LNCS 664, 245–257 (1993).

[KOPTT97] Y. Kinoshita, P. O'Hearn, J. Power, M. Takeyama and R. Tennent. An axiomatic approach to binary logical relations with applications to data refinement. *Proc. TACS'97*, Springer LNCS 1281, 191–212 (1997).

[Mit90] J. Mitchell. Type Systems for Programming Languages. Chapter 8 of *Handbook of Theoretical Computer Science, Vol B.* Elsevier (1990).

[Mit96] J. Mitchell. *Foundations for Programming Languages*. MIT Press (1996).

[MM91] J. Mitchell and E. Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure And Applied Logic* 51:99–124 (1991).

[Plo80] G. Plotkin. Lambda-definability in the full type hierarchy. In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 363–373. Academic Press (1980).

[PPS98] G. Plotkin, J. Power and D. Sannella. A compositional generalisation of logical relations. Draft report, `http://www.dcs.ed.ac.uk/home/dts/pub/laxlogrel.ps` (1998).

[Rob96] E. Robinson. Logical relations and data abstraction. Report 730, Queen Mary and Westfield College (1996).

[ST88] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica* 25:233–281 (1988).

[Sch87] O. Schoett. Data Abstraction and the Correctness of Modular Programming. Ph.D. thesis CST-42-87, Univ. of Edinburgh (1987).

[Sta85] R. Statman. Logical relations and the typed lambda calculus. *Information and Control* 65:85–97 (1985).

[Ten94] R. Tennent. Correctness of data representations in Algol-like languages. In: *A Classical Mind: Essays in Honour of C.A.R. Hoare*. Prentice Hall (1994).

Data Refinement for Call-By-Value Programming Languages* **

Yoshiki Kinoshita[1] and John Power[2]

[1] Electrotechnical Laboratory, 1–1–4 Umezono, Tsukuba-shi, Ibaraki, 305 Japan
[2] Laboratory for the Foundations of Computer Science, University of Edinburgh, King's Buildings, Edinburgh EH9 3JZ, Scotland

Abstract. We give a category theoretic framework for data-refinement in call-by-value programming languages. One approach to data refinement for the simply typed λ-calculus is given by generalising the notion of logical relation to one of lax logical relation, so that binary lax logical relations compose. So here, we generalise the notion of lax logical relation, defined in category theoretic terms, from the simply typed λ-calculus to the computational λ-calculus as a model of data refinement.

1 Introduction

A fundamental tenet of data refinement is that data refinements compose, i.e., if M refines N, and N refines P, then M refines P. This fact has meant that, although in principle, one would expect logical relations to model data refinement, that has not been possible because binary logical relations do not compose (see [20]). In response to that problem, there have been attempts to extend the notion of logical relation to a notion of lax logical relation, retaining the fundamental features of logical relations but allowing composition. From a category theoretic perspective, these include [10] and [20], generalising Hermida's approach to logical relations in [3].

For the simply typed λ-calculus generated by a signature Σ, (we recall with detail in Section 2 that) to give a logical relation is equivalent to giving a strict cartesian closed functor from the cartesian closed category L determined by the term model for Σ, to Rel_2, the cartesian closed category for which an object is a pair of sets X and Y together with a binary relation R from X to Y. A lax logical relation is exactly the same except that the functor from L to Rel_2 although still required to preserve finite products strictly, equivalently, to respect contexts, need not preserve exponentials. There is a syntactic counterpart to this (see Section 2), but the above is the most compact definition.

* This work has been done with the support of the STA's COE project "Global Information Processing Technology" and British Council Grant 747FCS R34807. The second author also acknowledges the support of EPSRC grant GR/J84205: Frameworks for programming language semantics and logic and EPSRC grant R34723.

** **Keywords:** category theory, categorical models and logic, denotational semantics, lambda and combinatory calculi

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, pp. 562–576, 1999.

More generally, the work of [3,10,20] (see also [7]) has addressed data refinement, or lax logical relations, where the term model is a category with algebraic structure, as is common in purely logical situations (see [16]) such as the simply typed λ-calculus. But in a call-by-value programming language such as ML, one needs to distinguish carefully between computational expressions and values, leading to consideration of a pair of categories, one generated by arbitrary terms, the other generated by values, with a functor from the second to the first. So, in order to generalise the above notion of lax logical relation to call-by-value programming languages, we require a careful exploration of the distinction between expressions and values, and exactly how finite products, modelling contexts, generalise to account for the distinction. So this paper is devoted to a category theoretic account of data refinement for call-by-value languages, the primary point being the careful distinction between arbitrary terms and values.

For concreteness, we consider the call-by-value language given by the computational λ-calculus, or λ_c-calculus, as introduced by Moggi in [17]. This provides a natural fragment of a range of call-by-value languages such as ML. The λ_c-calculus has a sound and complete class of models, each of which is given by a category C with finite products and a strong monad T on C, such that T has Kleisli exponentials (see Section 3).

In order to generalise the account of data-refinement in [20] from the λ-calculus to the λ_c-calculus, we need to characterise this class of models in terms of some mild generalisation of the notion of category with finite products, subject to a closedness condition like that in cartesian closedness. So in Section 3, we characterise the models of the λ_c-calculus as closed $Freyd$-categories, where the notion of $Freyd$-category (cf [22]) generalises that of category with finite products and models contexts, while the notion of closedness generalises that in the definition of cartesian closed category (cf [21]). A lax logical relation for the λ_c-calculus is then a $Freyd$-functor from the term model of a given signature into an appropriate $Freyd$-category generalising the category Rel_2 above. The definition of a $Freyd$-category appears in Section 4.

In order to give a Basic Lemma for lax logical relations and hence for data refinement, we require several axioms all of similar form, such as the axiom that if $f\, R_{(X\times Y)\Rightarrow Z}\, g$, then $Curry(f)\, R_{X\Rightarrow Y\to Z}\, Curry(g)$. In principle, we need one axiom for each term-constructor of the language, together with axioms for unCurrying and to ensure that values are respected. These axioms weaken the usual logical relation axiom, which says that two functions are related if and only if they send related arguments to related results. So in Section 4, we give a Basic Lemma and explain its use in data refinement.

There are several generalisations of the above analysis, most of which may be made using current techniques. We can generalise from set-theoretic models of a language to models in an arbitrary closed $Freyd$-category. We thus take our relations in an arbitrary $Freyd$-category too, with a little extra data and conditions, this $Freyd$-category generalising the category of binary relations, cf [15]. Also, we can extend from the λ_c-calculus to other languages, for instance incorporating coproducts. In general, our notion of lax logical relation extends

to any language extending the λ_c-calculus and given by algebraic structure in a sense we can make precise (see [19]). We can also account for representation independence following [10]. For space reasons, we must defer this.

There has been an enormous amount of work on data refinement. Much of it stems from Hoare's original paper on data representation [4]. Later, Hoare [5], then Hoare and He Jifeng [6], developed a category theoretic account of data refinement (see [11] for a recent account in standard category theoretic terms and see [13] for application of these ideas in practice). In that account, data refinements do compose, but as originally described, it is limited for higher order structure such as that of the λ_c-calculus, as explained by Tennent in [23]. In fact, the construction of this paper generalises that of Hoare and He by treating functions as single-valued relations. Tennent in turn advocated the use of logical relations, for which a general text is [16] and for which a category theoretic account is given in [3]. As we have explained, binary logical relations do not compose, so one seeks a mild generalisation of logical relations (see [10,20] and the work herein) in order to have the advantages of both Hoare's formulation, which admits composability, and logical relations, which admit an account of higher order structure. An alternative is to consider call-by-value languages with data refinement modelled by predicate transformers, as David Naumann has done in [18]. There has been plenty of other work on data refinement too, but not from a category theoretic perspective, for instance [2].

The most closely related work other than that detailed in Section 2 is that of [10]. The key differences are that here, we address call-by-value languages; we also insist that contexts be preserved, motivating our key definitions of Section 3; but we do not address representation independence, which was the central computational issue addressed in that paper. Later, we plan to extend this work, using the insights of that paper, to account for representation independence.

The paper is organised as follows. In Section 2, we recall the definition of lax logical relation for the simply typed λ-calculus and we set notation. In Section 3, we introduce the notion of *Freyd*-category and characterise the models of the λ_c-calculus as closed *Freyd*-categories. Finally, in Section 4, we show how our definitions may be used to model data refinement. A more substantial use of our whole body of work on data refinement appears in [13].

2 Lax Logical Relations for the Simply Typed λ-Calculus

In this section, we review the work of [20], generalising logical relations to lax logical relations: the latter, unlike the former, compose, in the sense that if R is a lax logical relation from M to N and S is a lax logical relation from N to P, then the pointwise composite of relations $R \circ S$ is a lax logical relation from M to P. That makes lax logical relations, unlike logical relations, apposite for data refinement (see also [10]). Accompanying the definition of lax logical relation is a generalisation of the Basic Lemma of logical relations, allowing one to check a data refinement by checking that basic operations are respected: that is the other central tenet of data refinement, in addition to composability, as advocated, for

instance, by Hoare [5]. Each of these conditions separately is easy to achieve; the combination is more difficult in the presence of higher order structure, and it is that combination that is the topic of this paper.

We give a detailed exposition of the work in [20] as we need the various definitions later.

Let Σ be a signature of basic types and terms for the simply typed λ-calculus with products, generating a language L. Let σ and τ be any types in L. We denote the set of functions from a set X to a set Y by $[X, Y]$.

Definition 1. *A* model *M of L in Set consists of*

- *for each σ, a set M_σ, such that $M_{\sigma\to\tau} = [M_\sigma, M_\tau]$, $M_{\sigma\times\tau} = M_\sigma \times M_\tau$, and $M_1 = 1$*
- *for each base term c of Σ of type σ, an element $M(c)$ of M_σ.*

A model extends inductively to send each judgement $\Gamma \vdash t : \sigma$ of L to a function from M_Γ to M_σ, where M_Γ is the evident finite product in *Set*. Given a signature Σ and two interpretations, M and N, of the language L generated by Σ, we say

Definition 2. *A* binary logical relation *from M to N consists of, for each type σ of L, a relation*

$$R_\sigma \subseteq M_\sigma \times N_\sigma \tag{1}$$

such that

- *for all $f \epsilon M_{\sigma\to\tau}$ and $g \epsilon N_{\sigma\to\tau}$, we have $f R_{\sigma\to\tau} g$ if and only if for all $(x, y) \epsilon M_\sigma \times N_\sigma$, if $x\, R_\sigma\, y$, then $f(x)\, R_\tau\, g(y)$*
- *for all $(x_0, x_1) \epsilon M_\sigma \times M_\tau$ and $(y_0, y_1) \epsilon N_\sigma \times N_\tau$, we have $(x_0, x_1)\, R_{\sigma\times\tau}\, (y_0, y_1)$ if and only if $x_0\, R_\sigma\, y_0$ and $x_1\, R_\tau\, y_1$*
- *$1\, R_1\, 1$, where 1 is the unique element of $M_1 = N_1 = 1$*
- *$M(c)\, R_\sigma\, N(c)$ for every base term c in Σ of type σ.*

The data for a binary logical relation is completely determined by its behaviour on base types. The fundamental result about logical relations is as follows.

Definition 3. (The Basic Lemma) *Let R be a binary logical relation. Then for any term $t : \sigma$ of L in context Γ, if $x\, R_\Gamma\, y$, then $M(\Gamma \vdash t) x\, R_\sigma\, N(\Gamma \vdash t) y$.*

We now outline a category theoretic formulation of logical relations [3]. The language L generates a cartesian closed category, which we also denote by L, such that a model M of the language L in any cartesian closed category such as *Set* extends uniquely to a functor $M : L \longrightarrow Set$ that preserves cartesian closed structure strictly [14]. We may therefore identify the notion of model of the language L with that of a functor strictly preserving cartesian closed structure from L into *Set*.

Definition 4. *The category Rel_2 is defined as follows: an object consists of a pair (X, Y) of sets and a binary relation R from X to Y; a map from (X, R, Y) to (X', R', Y') is a pair of functions $(f : X \longrightarrow X', g : Y \longrightarrow Y')$ such that $x\,R\,y$ implies $f(x)\,R'\,g(y)$; composition is given by ordinary composition of functions. We denote the forgetful functor from Rel_2 to $Set \times Set$ sending (X, R, Y) to (X, Y) by $(\delta_0, \delta_1) : Rel_2 \longrightarrow Set \times Set$.*

The category Rel_2 is cartesian closed, and the cartesian closed structure is preserved by (δ_0, δ_1). We typically abbreviate Rel_2 by Rel when the context is clear.

Proposition 1. *To give a binary logical relation from M to N is equivalent to giving a functor $R : L \longrightarrow Rel$ strictly preserving cartesian closed structure, such that $(\delta_0, \delta_1)R = (M, N)$.*

This proposition is developed and extended in [3], giving a category theoretic treatment of logical relations in terms of fibrations with structure.

It is more conceptual and compact to express the notion of lax logical relation in category theoretic terms, then characterise the definition in more syntactic terms, extending but reversing Proposition 1.

Given a signature Σ and the language L generated by Σ, and two models M and N of L in Set, we say

Definition 5. *A binary* lax logical relation *from M to N is a functor $R : L \longrightarrow Rel$ strictly preserving finite products such that $(\delta_0, \delta_1)R = (M, N)$.*

From the perspective of this definition, the Basic Lemma for lax logical relations is a triviality because the definition amounts to the assertion that for any term t of L of type σ in context Γ, if $x\,R_\Gamma\,y$, then $M(\Gamma \vdash t)x\,R_\sigma\,N(\Gamma \vdash t)y$. So we may express the Basic Lemma as giving an equivalence between the above definition and one in more syntactic terms as follows.

Theorem 1. (The Basic Lemma) *To give a lax logical relation from M to N is to give for each type σ of L, a relation*

$$R_\sigma \subseteq M_\sigma \times N_\sigma \tag{2}$$

such that

1. *if $f_0\,R_{\sigma\to\tau}\,g_0$ and $f_1\,R_{\sigma\to\rho}\,g_1$, then $(f_0, f_1)\,R_{\sigma\to(\tau\times\rho)}\,(g_0, g_1)$*
2. *$\pi_0\,R_{\sigma\times\tau\to\sigma}\pi_0$ and $\pi_1\,R_{\sigma\times\tau\to\tau}\,\pi_1$*
3. *if $f\,R_{(\sigma\times\tau)\to\rho}\,g$, then $Curry(f)\,R_{\sigma\to\tau\to\rho}\,Curry(g)$*
4. *$ev\,R_{(\sigma\times(\sigma\to\tau))\to\tau}\,ev$*
5. *if $f\,R_{\sigma\to\tau}\,g$ and $f'\,R_{\tau\to\rho}\,g'$, then $f'f\,R_{\sigma\to\rho}\,g'g$*
6. *$id\,R_{\sigma\to\sigma}\,id$*
7. *$x\,R_\sigma\,y$ if and only if $x\,R_{1\to\sigma}\,y$*
8. *$M(c)\,R_\sigma\,N(c)$ for every base term c in Σ of type σ.*

The key point of the proof is that every map in the category L is generated by the terms appearing in the above axioms. In these syntactic terms, the difference between logical relations and lax logical relations is that for the former, $f\,Rg$ if and only if $fx\,R\,gy$ whenever $x\,Ry$, but the reverse direction of that equivalence does not hold for lax logical relations: it is replaced by a more complex set of rules.

Finally, in justifying our definition for the purposes of data refinement, we have

Proposition 2. *If R and S are binary lax logical relations, then so is the composite of relations $R \circ S$.*

To see an example of a lax logical relation that is not a logical relation, simply take a pair of non-trivial logical relations and compose them: by the proposition, their composite will be a lax logical relation, but the composite is almost never logical.

3 Models of the Computational λ-Calculus

In this section, we give a version of the computational λ-calculus, or λ_c-calculus, and analyse its models. For the simply typed λ-calculus, it was essential, in defining the notion of lax logical relation, to understand its models as cartesian categories with a closedness property. So in generalising the notion of lax logical relation to the λ_c-calculus, we seek to characterise its models as a generalisation of the notion of category with finite products, subject to a closedness condition like that for cartesian closed categories. In order to do that, we characterise λ_c-models as closed $Freyd$-categories.

There are several equivalent formulations of the λ_c-calculus. We shall not use the original formulation but one of the equivalent versions. The λ_c-calculus has type constructors given by

$$X ::= B \mid X_1 \times X_2 \mid 1 \mid X \Rightarrow Y \tag{3}$$

where B is a base type. We do not assert the existence of a type constructor TX: this formulation is equivalent to the original one because TX may be defined to be $1 \Rightarrow X$.

The terms of the λ_c-calculus are given by

$$e ::= x \mid b \mid e'e \mid \lambda x.e \mid * \mid (e, e') \mid \pi_i(e) \tag{4}$$

where x is a variable, b is a base term of arbitrary type, $*$ is of type 1, with π_i existing for $i = 1$ or 2, all subject to the evident typing. Again, this differs from the original formulation in that we do not explicitly have a *let* constructor or constructions $[e]$ or $\mu(e)$. Again, the two formulations are equivalent as we may consider $let\ x = e\ in\ e'$ as syntactic sugar for $(\lambda x.e')e$, and $[e]$ as syntactic sugar for $\lambda x.e$ where x is of type 1, and $\mu(e)$ as syntactic sugar for $e(*)$.

The λ_c-calculus has two predicates, existence, denoted by $\downarrow$, and equivalence, denoted by $\equiv$. The $\downarrow$ rules may be expressed as saying $* \downarrow$, $x \downarrow$, $\lambda x.e \downarrow$ for all e, if $e \downarrow$ then $\pi_i(e) \downarrow$, and similarly for (e, e'). A value is a term e such that $e \downarrow$. The rules for $\equiv$ say $\equiv$ is a congruence, with variables allowed to range over values, and give rules for the basic constructions and for unit, product and functional types. It follows from the rules that types together with equivalence classes of terms form a category, with a subcategory determined by values.

It is straightforward, using the original formulation of the λ_c-calculus in [17], to spell out the inference rules required to make this formulation agree with the original one: one just bears in mind that the models are the same, and we use syntactic sugar as detailed above. Space does not allow a list of the rules here.

The λ_c-calculus represents a fragment of a call by value programming language. In particular, it was designed to model fragments of ML, but is also a fragment of other languages such as FPC (see [1]). For category theoretic models, the key feature is that there are two entities, expressions and values, so the most direct way to model the language as we have formulated it is in terms of a pair of categories V and E, together with an identity on objects inclusion functor $J : V \longrightarrow E$, subject to some generalisation of the notion of finite product in order to model contexts, further subject to a closedness condition to model $X \Rightarrow Y$. This train of thought leads directly to the notion of closed $Freyd$-category, which we shall compare with the original formulation of the class of models.

A sound and complete class of models for the λ_c-calculus was given by Moggi in [17]: a model consists of a category C with finite products, together with a strong monad T on C, such that T has Kleisli exponentials, i.e., for each pair of objects X and Y, there exists an object $X \Rightarrow Y$ such that $C(Z \times X, TY)$ is isomorphic to $C(Z, X \Rightarrow Y)$ for all Z, naturally in Z.

We recall the definitions of premonoidal category and strict premonoidal functor, and symmetries for them, as introduced in [21] and further studied in [19]. We use them to define the notion of $Freyd$-category. A premonoidal category is a generalisation of the concept of monoidal category: it is essentially a monoidal category except that the tensor need only be a functor of two variables and not necessarily be bifunctorial, i.e., given maps $f : X \longrightarrow Y$ and $f' : X' \longrightarrow Y'$, the evident two maps from $X \otimes X'$ to $Y \otimes Y'$ may differ.

In order to make precise the notion of a premonoidal category, we need some auxiliary definitions.

Definition 6. *A* binoidal category *is a category K together with, for each object X of K, functors $h_X : K \longrightarrow K$ and $k_X : K \longrightarrow K$ such that for each pair (X, Y) of objects of K, $h_X Y = k_Y X$. The joint value is denoted $X \otimes Y$.*

Definition 7. *An arrow $f : X \longrightarrow X'$ in a binoidal category K is* central *if for every arrow $g : Y \longrightarrow Y'$, the following diagrams commute*

$$\begin{array}{ccc} X \otimes Y & \xrightarrow{X \otimes g} & X \otimes Y' \\ \downarrow{\scriptstyle f \otimes Y} & & \downarrow{\scriptstyle f \otimes Y'} \\ X' \otimes Y & \xrightarrow[X' \otimes g]{} & X' \otimes Y' \end{array} \qquad \begin{array}{ccc} Y \otimes X & \xrightarrow{g \otimes X} & Y' \otimes X \\ \downarrow{\scriptstyle Y \otimes f} & & \downarrow{\scriptstyle Y' \otimes f} \\ X \otimes X' & \xrightarrow[g \otimes X']{} & Y' \otimes X' \end{array}$$

A natural transformation $\alpha : G \Longrightarrow H : C \longrightarrow K$ is called *central* if every component of α is central.

Definition 8. *A* premonoidal category *is a binoidal category K together with an object I of K, and central natural isomorphisms a with components $(X \otimes Y) \otimes Z \longrightarrow X \otimes (Y \otimes Z)$, l with components $X \longrightarrow X \otimes I$, and r with components $X \longrightarrow I \otimes X$, subject to two equations: the pentagon expressing coherence of a, and the triangle expressing coherence of l and r with respect to a (see [9] for an explicit depiction of the diagrams).*

Proposition 3. *Given a strong monad T on a symmetric monoidal category C, the Kleisli category $Kl(T)$ for T is a premonoidal category, with the functor $J : C \longrightarrow Kl(T)$ preserving premonoidal structure strictly: a monoidal category such as C is trivially a premonoidal category.*

So a good source of examples of premonoidal categories in general is provided by Moggi's work on monads as notions of computation [17].

Definition 9. *Given a premonoidal category K, the* centre *of K, denoted $Z(K)$, is the subcategory of K consisting of all the objects of K and the central morphisms.*

Given a strong monad on a symmetric monoidal category, the base category C need not be the centre of $Kl(T)$. But, modulo the condition that $J : C \longrightarrow Kl(T)$ be faithful, or equivalently, the mono requirement [17,21], i.e., the condition that the unit of the adjunction be pointwise monomorphic, it must be a subcategory of the centre.

The functors h_A and k_A preserve central maps. So we have

Proposition 4. *The centre of a premonoidal category is a monoidal category.*

This proposition allows us to prove a coherence result for premonoidal categories, directly generalising the usual coherence result for monoidal categories. Details appear in [21].

Definition 10. *A* symmetry *for a premonoidal category is a central natural isomorphism with components $c : X \otimes Y \longrightarrow Y \otimes X$, satisfying the two conditions $c^2 = 1$ and equality of the evident two maps from $(X \otimes Y) \otimes Z$ to $Z \otimes (X \otimes Y)$. A* symmetric *premonoidal category is a premonoidal category together with a symmetry.*

Symmetric premonoidal categories are those of primary interest to us, and seem to be those of primary interest in denotational semantics in general.

Definition 11. *A* strict premonoidal functor *is a functor that preserves all the structure and sends central maps to central maps.*

One may similarly generalise the definition of strict symmetric monoidal functor to strict symmetric premonoidal functor.

We say a functor is the identity on objects when the object part of the functor is the identity function, and therefore both sets of objects are the same.

Definition 12. *A Freyd-category consists of a category C with finite products, a symmetric premonoidal category K, and an identity on objects strict symmetric premonoidal functor $J : C \longrightarrow K$. A strict Freyd-functor consists of a pair of functors that preserve all the Freyd-structure strictly.*

Definition 13. *A Freyd-category $J : C \longrightarrow K$ is* closed *if for every object X, the functor $J(X \otimes -) : C \longrightarrow K$ has a right adjoint. A strict* closed *Freyd-functor is a Freyd-functor that preserves all the closed structure strictly.*

Observe that it follows that the functor $J : C \longrightarrow K$ has a right adjoint, and so K is the Kleisli category for a monad on C. We sometimes write K for a *Freyd*-category, as the rest of the structure is usually implicit: often, it is given by $Z(K)$ and the inclusion.

A variant of one of the main theorems of [19] is

Theorem 2. *To give a closed Freyd-category is to give a category C with finite products together with a strong monad T on C together with assigned Kleisli exponentials. To give a strict closed Freyd-functor is to give a strict map of strong monads that strictly preserves Kleisli exponentials.*

Observe that given a category C with finite products and a strong monad T on it, $Kl(T)$ is a *Freyd*-category. A functor preserving the strong monad and the finite products strictly yields a strict *Freyd*-functor, but the converse is not true.

It follows from Moggi's result, but may also be proved directly, that closed *Freyd*-categories provide a sound and complete class of models for the λ_c-calculus.

4 Data Refinement for the λ_c-Calculus

In this section, we use our analysis of the computational λ-calculus of Section 3 to generalise our account of data refinement, or lax logical relations, for the simply typed λ-calculus as in Section 2.

For concreteness, we shall consider *Set*-based models of the λ_c-calculus. So assume we are given a monad T on *Set*. Every monad on *Set* has a unique

strength, and Kleisli exponentials always exist. So if we denote the Kleisli category by Set_T, then Set_T is a closed $Freyd$-category (leaving Set and the canonical functor $J : Set \longrightarrow Set_T$ implicit by the convention we adopted in the preceding section). Given a signature Σ for the λ_c-calculus, let L denote the closed $Freyd$-category, equivalently the λ_c-model, generated by Σ. Extending our convention for the λ-calculus, and following Hoare's convention in his modelling of data refinement [5,11], we identify the language generated by Σ with the closed $Freyd$-category L. We denote the subcategory of values by L_v with inclusion $J : L_v \longrightarrow L$. Extending one of the equivalent formulations of Section 2 for the simply typed λ-calculus, we say

Definition 14. *A* model *M of L_v in Set_T is a strict closed $Freyd$-functor from L_v to Set_T.*

We shall give an example of this and later definitions at the end of the section. We now need to generalise the construction Rel_2 to an appropriate $Freyd$-category. Recall that Rel_2 has finite products, and that they are preserved by the two projections to Set.

Proposition 5. *Given a monad T on Set, the following data forms a $Freyd$-category Rel_{2T} together with a pair of strict $Freyd$-functors from Rel_{2T} to Set_T:*

- *the category Rel_2 as defined in Section 2*
- *the category Rel_{2T} with the same objects as Rel_2 but with an arrow from (X, R, Y) to (X', R', Y') given by maps $f : X \longrightarrow TX'$ and $g : Y \longrightarrow TY'$ such that there exists a map $h : R \longrightarrow TR'$ commuting with the projections, with the evident composition*
- *the canonical functor $J : Rel_2 \longrightarrow Rel_{2T}$*
- *the projections $\delta_0, \delta_1 : Rel_{2T} \longrightarrow Set_T$.*

The functor $J : Rel_2 \longrightarrow Rel_{2T}$ has a right adjoint given by sending a relation (X, R, Y) to the pair (TX, TY) together with the subobject of $TX \times TY$ determined by the projections from TR. It follows that Rel_{2T} is closed and is therefore a λ_c-model. Axiomatically, that is because Rel_2 is cartesian closed and Set has epi-mono factorisations. We avoid an assumption that T preserves jointly monic pairs because it is not true of powerdomains: a powerdomain is a construct for modelling nondeterminism, a slightly simplified version of one being the endofunctor on Set that sends a set X to its set of finite subsets, $P_f(X)$, with the operation of the endofunctor on maps given by taking the image of each finite subset. A jointly monic pair in Set amounts to a pair of sets (X, Y) together with a subset R of $X \times Y$. Our point here is that the set of finite subsets $P_f(R)$ of R need not be exhibited by the functor P_f as a subset of $P_f(X) \times P_f(Y)$, as for instance can be seen by taking X and Y both to be two element sets with R their product. For notational simplicity, we abbreviate Rel_{2T} by Rel_T where the context is clear.

Observe that one could extend logical relations from the λ-calculus to the λ_c-calculus by treating a logical relation for the λ_c-calculus as a strict closed

Freyd-functor, or equivalently a map of λ_c-models, from L to Rel_T commuting with the projections.

We now extend our definition of lax logical relation from the simply typed λ-calculus to the λ_c-calculus. The central idea is to relax preservation of all structure to preservation of that structure required to model contexts, i.e., to *Freyd*-structure.

Definition 15. *A binary* lax logical relation *from M to N is a strict Freyd-functor $R : L \longrightarrow Rel_T$ such that $(\delta_0, \delta_1)R = (M, N)$.*

It is not automatically the case that a pointwise composite of binary lax logical relations is again a binary lax logical relation. That requires an extra condition on the monad T on *Set*. The central point is that we must consider when the composite of two binary relations extends from Rel_2 to Rel_{2T}; the condition we need is that T weakly preserves pullbacks, i.e., that if

$$\begin{array}{ccc} P & \xrightarrow{h} & X \\ {\scriptstyle k}\downarrow & & \downarrow{\scriptstyle k'} \\ Y & \xrightarrow[h']{} & Z \end{array}$$

is a pullback, then the diagram

$$\begin{array}{ccc} TP & \xrightarrow{Th} & TX \\ {\scriptstyle Tk}\downarrow & & \downarrow{\scriptstyle Tk'} \\ TY & \xrightarrow[Th']{} & TZ \end{array}$$

satisfies the existence part of the definition of pullback. This condition is the central condition used to analyse functional bisimulation in [8] with several of the same examples. Examples of such monads are powerdomains, $S \Rightarrow (S \times -)$ for a set S, as used for modelling side-effects, and similarly for monads used for modelling partiality, or exceptions, or combinations of the above. It does not seem to hold of the monad $(- \Rightarrow R) \Rightarrow R$ as has been used to model continuations; but that does not concern us greatly, as data refinement for continuations seems likely to follow a different paradigm to that adopted here anyway.

Theorem 3. *Let T be a monad on Set that weakly preserves pullbacks. Then for any lax logical relations $R : L \longrightarrow Rel_T$ and $S : L \longrightarrow Rel_T$ such that $\delta_1 R = \delta_0 S$, the pointwise composite of relations yields a lax logical relation $R \circ S$.*

The proof requires one use of the fact that strong epimorphisms in *Set* are retracts. One can avoid it by a more delicate use of epi-mono factorisations,

allowing the result to extend from *Set* to a regular cartesian closed category. There would be more difficulty if we demanded that a lax logical relation preserve not merely *Freyd*-structure but also the monad, as one would need a condition such as T preserving strong epimorphisms, contradicting examples such as $T = S \Rightarrow (S \times -)$.

There are two central assertions in our definition: every map in L is sent to a map in Rel_T, and every map in L_v is sent to a map in Rel. Given models M and N of L, the first condition says that for every expression in context, $\Gamma \vdash e : \sigma$, if $x \, R_\Gamma \, y$, then $M(\Gamma \vdash e : \sigma)x$ is related to $N(\Gamma \vdash e : \sigma)y$ by the relation generated by TR_σ. For instance, if T was a powerdomain, then for any nondeterministic program, if two inputs are related, for every possible output of either, there is a related possible output of the other, as in bisimulation. The second condition says that if one has a value, then if $x \, R_\Gamma \, y$, one must have the stronger result that $M(\Gamma \vdash e : \sigma)x \, R_\sigma \, N(\Gamma \vdash e : \sigma)y$ So for instance, λ-terms are related in the usual way. So our *definition* of lax logical relation amounts to the *conclusion* of a statement of a Basic Lemma for lax logical relations.

Using our definition as a conclusion, we now give a generalised Basic Lemma for lax logical relations for the λ_c-calculus, generalising our result in Section 2. Thus we seek syntactic conditions that imply that every map in L is sent to a map in Rel_T and that every map in L_v is sent to a map in Rel. Owing to the presence of λ-abstraction, every map in L is the unCurrying of a map in L_v with domain 1, so except for one condition saying that unCurrying preserves relations, we need only give conditions about maps in L_v. The list of axioms may seem long, but it is not much longer than that in Section 2, and the axioms correspond closely to the type and term constructors of the language. The key point is that there is no converse to the final axiom: a converse would give a notion of logical relation and would disallow composability.

Theorem 4. (The Basic Lemma) *To give a lax logical relation from M to N is to give for each type σ of L, a relation*

$$R_\sigma \subseteq M_\sigma \times N_\sigma \tag{5}$$

such that

1. *if $f_0 \, R_{\sigma\Rightarrow\tau} \, g_0$ and $f_1 \, R_{\sigma\Rightarrow\rho} \, g_1$, then $(f_0, f_1) \, R_{\sigma\Rightarrow(\tau\times\rho)} \, (g_0, g_1)$*
2. *$\pi_0 \, R_{\sigma\times\tau\Rightarrow\sigma} \pi_0$ and $\pi_1 \, R_{\sigma\times\tau\Rightarrow\tau} \, \pi_1$*
3. *if $f \, R_{(\sigma\times\tau)\Rightarrow\rho} \, g$, then $Curry(f) \, R_{\sigma\Rightarrow\tau\Rightarrow\rho} \, Curry(g)$*
4. *$ev \, R_{(\sigma\times(\sigma\Rightarrow\tau))\Rightarrow\tau} \, ev$*
5. *if $f \, R_{\sigma\Rightarrow\tau} \, g$ and $f' \, R_{\tau\Rightarrow\rho} \, g'$, then $f'f \, R_{\sigma\Rightarrow\rho} \, g'g$*
6. *$id \, R_{\sigma\Rightarrow\sigma} \, id$*
7. *$\eta(M(c)) \, R_{1\Rightarrow\sigma} \, \eta(N(c))$ for every base term c in Σ of type σ*
8. *if $x \, R_\sigma \, y$ and $f \, R_{(\sigma\times\tau)\Rightarrow\rho} \, g$, then $fx \, R_{\tau\Rightarrow\rho} \, gy$*
9. *$(x_0, x_1) \, R_{\sigma\times\tau} \, (y_0, y_1)$ if and only if $x_0 \, R_\sigma \, y_0$ and $x_1 \, R_\tau \, y_1$*
10. *$* \, R_1 \, *$*
11. *if $x \, R_{1\Rightarrow\sigma} \, y$, then $T(R_\sigma)$ induces a relation between $x(*)$ and $y(*)$.*

Proof. For the forward direction, the relations R_σ are given by the object part of the strict $Freyd$-functor. The first ten conditions follow from the fact that R_v has an action on all maps, and from the fact that it strictly preserves finite products. For instance, there is a map in L_v from $(\sigma \Rightarrow \tau) \times (\sigma \Rightarrow \rho)$ to $\sigma \Rightarrow (\tau \times \rho)$, so that map is sent by R_v to a map in Rel, and R_v strictly preserves finite products, yielding the first condition. So using the definition of a map in Rel, and the fact that $(\delta_0, \delta_1)R = (M, N)$, and the fact that M and N are strict structure preserving functors, we have the result. The final condition holds because R is a functor.

For the converse, the family of relations gives the object part of the strict $Freyd$-functor R. The ninth and tenth axioms imply that R_v strictly preserves finite products if it forms a functor. The data for M and N and the coherence condition $(\delta_0, \delta_1)R = (M, N)$ on the putative strict $Freyd$-functor determine its behaviour on maps. It remains to check that the image of every map in L lies in Rel_T and the image of every map in L_v lies in Rel. The first seven conditions inductively define the Currying of every map in L, so unCurrying by using the sixth, eighth, fifth and eleventh conditions, it follows that R is a functor. The eighth, ninth, and tenth conditions ensure that R restricts to a functor from L_v to Rel. It is routine to verify that these constructions are mutually inverse.

One might wonder why we require the ninth and tenth conditions here while they do not appear in Theorem 1. The reason is that the seventh condition of Theorem 1 mirrors an equivalence that holds for the simply typed λ-calculus but does not hold for the computational λ-calculus, the equivalence being that σ is equivalent to $1 \to \sigma$. That condition allows one to deduce ordinary λ-calculus versions of our ninth and tenth conditions here. The failure of that equivalence for the computational λ-calculus is also why our eleventh condition here corresponds to only one direction of the seventh condition of Theorem 1 too: for the eleventh condition, we use the expression *induce* because $T(R_\sigma)$ might not be a subobject of $T(\sigma) \times T(\tau)$ but it does have an epi-jointly monic factorisation, and that is what we intend.

Finally, we shall consider an example to see how this all works in practice.

Example 1. Consider the computational λ-calculus L_{Stack} generated by the data for a stack. We have base types $Stack$ and Nat, and we have base terms including *pop* and *push*. The intended semantics of the unCurrying of *pop* is a partial function from $M(Stack)$ to $M(Stack)$, with $M(Stack)$ being the usual set of stacks. The partiality of the intended semantics for *pop* is the reason it is convenient to consider the computational λ-calculus here rather than the ordinary λ-calculus. Let M be the intended semantics for stacks in $Set_\perp$, where $\perp$ is the usual lifting monad on Set. Recall, or note, that $\perp$ weakly preserves pullbacks, so our composability result holds. Let N be a model of L_{Stack} in $Set_\perp$ generated by modelling stacks in terms of trees, so $N(Stack)$ is the set of non-empty finite trees. Define a logical relation from M to N by defining it on base types as the identity on Nat and on $Stack$, by the usual relationship between stacks and trees. This respects base terms, so it automatically lifts to higher types. We might further define a model P of L_{Stack} in $Set_\perp$ by modelling stacks by lists of

natural numbers. We then have a logical relation S from N to P generated by the identity on Nat and on $Stack$, by relating finite trees with lists. Now taking the pointwise composite $R \circ S$, we have a lax logical relation from M to P.

5 Conclusions and Further Work

We have defined lax logical relations for data refinement in call-by-value languages represented by the computational λ_c-calculus. Binary lax logical relations compose, and have a basic lemma, thus satisfying two key criteria for data refinement.

The λ_c-calculus has a narrow range of type and term constructors. But the techniques herein apply in the considerably greater generality of call-by-value calculi with models in $[\rightarrow, Set]$-categories with algebraic structure [19]. So we could include an account of finite coproducts for instance.

We have also not addressed representation independence, the topic of [10], but the techniques of [10], based on the sketches in [12], extend to the setting of this paper. So we plan to make that extension. We hope for a converse to the leading result therein too.

Finally, for logical relations for the λ-calculus, if one asserts that the functor (δ_0, δ_1) be a fibration, then the logical structure is given by the internal language of the category theoretic structure [3]. It is not clear how to extend our analysis to such a result, but the concept of fibration may be the key construct. We may need to restrict attention to each fibre being a poset.

References

1. Fiore, M., Plotkin, G.D.: An axiomatisation of computationally adequate domain-theoretic models of FPC. Proc LICS **94**. IEEE Press (1994) 92–102.
2. Gardiner, P.H.B., Morgan, C.:A single complete rule for data refinement. Formal Aspects of Computing **5** (1993) 367–382.
3. Hermida, C.A.: Fibrations, Logical Predicates and Indeterminates. Ph.D. thesis. Edinburgh (1993) available as ECS-LFCS-**93-277**.
4. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica **1** (1972) 271–281.
5. Hoare, C.A.R.: Data refinement in a categorical setting. (1987) (draft).
6. Hoare, C.A.R., Jifeng, H.: Data refinement in a categorical setting. (1990) (draft).
7. Honsell, F., Sannella, D.T.: Pre-logical relations. Proc. CSL **99** (in this volume).
8. Johnstone, P.T., Power, A.J., Tsujishita, T., Watanabe, H., Worrell, J.: An Axiomatics for Categories of Transition Systems as Coalgebras. Proc LICS **98**. IEEE Press (1998) 207–213.
9. Kelly, G.M.: Basic concepts of enriched category theory. Cambridge University Press (1982).
10. Kinoshita, Y., O'Hearn, P., Power, A.J., Takeyama, M., Tennent, R.D.: An Axiomatic Approach to Binary Logical Relations with Applications to Data Refinement. Proc TACS **97**. LNCS **1281**. Abadi and Ito (eds.) Springer (1997) 191–212.

11. Kinoshita, Y., Power, A.J.: Data refinement and algebraic structure. Acta Informatica (to appear).
12. Kinoshita, Y., Power, A.J., Takeyama, M.: Sketches. J. Pure Appl. Algebra (to appear).
13. Kinoshita, Y., Power, A.J., Watanabe, H.: A general completeness result in refinement (submitted).
14. Lambek, J., Scott, P.J.: Introduction to Higher-Order Categorical Logic, Cambridge Studies in Advanced Mathematics **7**. CUP (1986).
15. Ma, Q., Reynolds, J.C.: Types, abstraction and parametric polymorphism **2**. Math. Found. of Prog. Lang. Sem. Lecture Notes in Computer Science. Springer (1991).
16. Mitchell, J.: Foundations for programming languages. Foundations of Computing Series. MIT Press (1996).
17. Moggi, E.: Computational Lambda-calculus and Monads. Proc LICS **89**. IEEE Press (1989) 14–23.
18. Naumann, D.A.: Data refinement, call by value, and higher order programs. Formal Aspects of Computing **7** (1995) 752–762.
19. Power, A.J.: Premonoidal categories as categories with algebraic structure. Theoretical Computer Science (to appear).
20. Plotkin, G.D., Power, A.J., Sannella, D.T.: A generalised notion of logical relations. (submitted).
21. Power, A.J., Robinson, E.P. Premonoidal categories and notions of computation. Math. Structures in Computer Science **7** (1997) 453–468.
22. Power, A.J., Thielecke, H.: Environments, Continuation Semantics and Indexed Categories. Proc TACS **97**. LNCS **1281** Abadi and Ito (eds) (1997) 391–414
23. Tennent, R.D.: Correctness of data representations in ALGOL-like languages. In: A Classical Mind, Essays in Honour of C.A.R. Hoare, A.W. Roscoe (ed.) Prentice-Hall (1994) 405–417.

Tutorial on Term Rewriting

Aart Middeldorp

Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba 305-8573, Japan
ami@is.tsukuba.ac.jp
http://www.score.is.tsukuba.ac.jp/~ami

Abstract. Term rewriting is an important computational model with applications in algebra, software engineering, declarative programming, and theorem proving. In term rewriting, computation is achieved by directed equations and pattern matching. In this tutorial we give an introduction to term rewriting.
The tutorial is organized as follows. After presenting several motivating examples, we explain the basic concepts and results in term rewriting: abstract rewriting, equational reasoning, termination techniques, confluence criteria, completion, strategies, and modularity. The tutorial concludes with a selection of more specialized topics as well as more recent developments in term rewriting: narrowing, advanced termination techniques (dependency pairs), conditional rewriting, rewriting modulo, tree automata techniques, and higher-order rewriting.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, p. 577, 1999.

Tutorial on Interactive Theorem Proving Using Type Theory

Douglas J. Howe

Bell Labs, Murray Hill, NJ 07974, USA
howe@research.bell-labs.com

Abstract. Interactive theorem provers are tools that assist humans in constructing formal proofs. They have been the subject of over two decades of research, and are now capable of tackling problems of real practical interest in software and hardware verification. Some of the most effective of these tools are based on expressive type theories. This tutorial is about interactive theorem proving based on type theory, with a slant toward type theories, such as Nuprl and PVS, where expressive power has been pushed at the expense of traditional properties such as decidability of typechecking. The tutorial will cover type theoretic foundations, practical issues in the design of type theories for verification, and techniques for automating reasoning in the context of interactive systems. We will also cover some of the recent work on cooperation between interactive provers and with automatic verification tools such as model checkers.

J. Flum and M. Rodríguez-Artalejo (Eds.): CSL'99, LNCS 1683, p. 578, 1999.

Author Index

T. Altenkirch 453
H.R. Andersen 111
C. Areces 307
T. Arts 96

V. Balat 250
J.L. Balcázar 2
P. Blackburn 307
L.D. Blekemishev 389
E. Bonelli 204
J.C. Bradfield 350

A. Compagnoni 420
R. Di Cosmo 250

G. Delzanno 50
V. Diekert 188

J. Esparza 50, 499

M. Fernández 220
M. Fränzle 126

P. Gastin 188, 515
H. Geuvers 439
J. Giesl 96
H. Goguen 420
E. Grädel 67
M. Grohe 14

S. Hagihara 277
J.E. Hannay 530
Y. Hirshfeld 172
F. Honsell 546
D.J. Howe 578
H. Hulgaard 111

S. Ishtiaq 235

A.D. Ker 405

Y. Kesten 141
Y. Kinoshita 562
S. Kreutzer 67
A. Kučera 499

C. Lautemann 322
D. Leivant 82
J. Lichtenberg 111

I. Mackie 220
J. Marcinkowski 338
M. Marx 307
A. Middeldorp 577
M. Mislove 515
J. Møller 111
P.D. Mosses 32

A. Neumann 484
H. Nickau 405

C.-H.L. Ong 405

A. Pnueli 141
A. Podelski 50
E. Poll 439
J. Power 562
D.J. Pym 235

A. Rabinovich 172
T.M. Rasmussen 157
B. Reus 453
L. Roversi 469

D. Sannella 546
H. Seidl 484
I.A. Stewart 374

S. Tobies 292
J. Torán 362

D. Vermeir 266
V. Vianu 1
M. De Vos 266

B. Weinzinger 322

N. Yonezaki 277

J. Zwanenburg 439