Yukiyoshi Kameyama
Peter J. Stuckey (Eds.)

LNCS 2998

Functional and Logic Programming

7th International Symposium, FLOPS 2004
Nara, Japan, April 2004
Proceedings

Springer

Lecture Notes in Computer Science 2998
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Yukiyoshi Kameyama   Peter J. Stuckey (Eds.)

Functional and Logic Programming

7th International Symposium, FLOPS 2004
Nara, Japan, April 7-9, 2004
Proceedings

Springer

eBook ISBN: 3-540-24754-8
Print ISBN: 3-540-21402-X

©2005 Springer Science + Business Media, Inc.

Print ©2004 Springer-Verlag
Berlin Heidelberg

All rights reserved

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic, mechanical, recording, or otherwise, without written consent from the Publisher

Created in the United States of America

Visit Springer's eBookstore at: http://ebooks.springerlink.com
and the Springer Global Website Online at: http://www.springeronline.com

Preface

This volume contains the proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS 2004), held in Nara, Japan, April 7–9, 2004 at the New Public Hall, Nara.

FLOPS is a forum for research on all issues concerning functional programming and logic programming. In particular it aims to stimulate the cross-fertilization as well as the integration of the two paradigms. The previous FLOPS meetings took place in Fuji-Susono (1995), Shonan (1996), Kyoto (1998), Tsukuba (1999), Tokyo (2001) and Aizu (2002). The proceedings of FLOPS 1999, FLOPS 2001 and FLOPS 2002 were published by Springer-Verlag in the Lecture Notes in Computer Science series, as volumes 1722, 2024 and 2441, respectively.

In response to the call for papers, 55 papers were submitted by authors from Australia (1), Austria (1), Canada (1), China (4), Denmark (2), Estonia ($\frac{1}{2}$), France ($3\frac{1}{2}$), Germany ($4\frac{1}{4}$), Italy (1), Japan (15), the Netherlands (1), Oman (1), Portugal ($\frac{1}{2}$), Singapore (2), Spain (8), UK (3), and USA ($6\frac{1}{4}$). Each paper was reviewed by at least three program committee members with the help of expert external reviewers. The program committee meeting was conducted electronically for a period of 2 weeks in December 2003. After careful and thorough discussion, the program committee selected 18 papers (33%) for presentation at the conference. In addition to the 18 contributed papers, the symposium included talks by three invited speakers: Masami Hagiya (University of Tokyo), Carsten Schürmann (Yale University), and Peter Selinger (University of Ottawa).

On behalf of the program committee, we would like to thank the invited speakers who agreed to give talks and contribute papers, and all those who submitted papers to FLOPS 2004. As program chairs, we would like to sincerely thank all the members of the FLOPS 2004 program committee for their excellent job, and all the external reviewers for their invaluable contribution. The support of our sponsors is gratefully acknowledged. We are indebted to the Kayamori Foundation of Informational Science Advancement, the Japan Society for Software Science and Technology (JSSST), the Association of Logic Programming (ALP), and the Asian Association for Foundation of Software (AAFS). Finally we would like to thank the members of the local arrangements committee, in particular the local arrangements chair Jacques Garrigue, for their invaluable support throughout the preparation and organization of the symposium.

February 2004

Yukiyoshi Kameyama
Peter J. Stuckey

This page intentionally left blank

Symposium Organization

Program Chairs

Yukiyoshi Kameyama	University of Tsukuba
Peter J. Stuckey	University of Melbourne

Program Committee

François Fages	INRIA Rocquencourt
Herman Geuvers	Nijmegen University
Michael Hanus	University of Kiel
Martin Hofmann	LMU München
Haruo Hosoya	University of Tokyo
Yukiyoshi Kameyama	University of Tsukuba
Naoki Kobayashi	Tokyo Institute of Technology
John Lloyd	Australian National University
Aart Middeldorp	University of Innsbruck
Atsushi Ohori	Japan Advanced Institute of Science and Technology
Germán Puebla	Technical University of Madrid
Morten Rhiger	IT-University of Copenhagen
Amr Sabry	Indiana University
Peter J. Stuckey	University of Melbourne
Martin Sulzmann	National University of Singapore
Kazunori Ueda	Waseda University
Philip Wadler	University of Edinburgh

Local Arrangements Chair

Jacques Garrigue	Kyoto University

Referees

Sergio Antoy
Krzysztof Apt
Richard Bird
Bernd Braßel
Francisco Bueno
Jesús Correas
Bruno Courcelle
Gregory J. Duck
Santiago Escobar
Kousha Etessami
Andrzej Filinski
Cormac Flanagan
Maarten Fokkinga
Jacques Garrigue
Robert Glück
Jose M. Gómez-Pérez
Rémy Haemmerlé
Masami Hagiya
Seif Haridi
Martin Henz
Hugo Herbelin
Ralf Hinze
Nao Hirokawa
Frank Huch
Atsushi Igarashi
Futoshi Iwama
Ben Kavanagh
Francisco López-Fraguas
Hans-Wolfgang Loidl
Kenny Lu Zhuo Ming
Wolfgang Lux
Michael Maher
Bart Massey
Yasuhiko Minamide
Shin-Cheng Mu
Masaki Murakami
Aleksandar Nanevski
Mizuhito Ogawa
Tomonobu Ozaki
Pawel Pietrzak
Rafael Ramirez
Mario Rodríguez-Artalejo
Chiaki Sakama
Taisuke Sato
Ken Satoh
Christian Schulte
Peter Selinger
Sylvain Soliman
Zoltan Somogyi
Harald Søndergaard
Bas Spitters
Koichi Takahashi
Andrew Tolmach
Yoshihito Toyama
Mirek Truszczynski
Christian Urban
Jaco van de Pol
Femke van Raamsdonk
René Vestergaard
Toshiko Wakaki
Freek Wiedijk
Toshiyuki Yamada
Akihiro Yamamoto
Tomokazu Yamamoto
Michal Young
Hans Zantema

Table of Contents

A Brief Survey of Quantum Programming Languages

Peter Selinger

Department of Mathematics, University of Ottawa
Ottawa, Ontario, Canada K1N 6N5
selinger@mathstat.uottawa.ca

Abstract. This article is a brief and subjective survey of quantum programming language research.

1 Quantum Computation

Quantum computing is a relatively young subject. It has its beginnings in 1982, when Paul Benioff and Richard Feynman independently pointed out that a quantum mechanical system can be used to perform computations [11, p.12]. Feynman's interest in quantum computation was motivated by the fact that it is computationally very expensive to simulate quantum physical systems on classical computers. This is due to the fact that such simulation involves the manipulation is extremely large matrices (whose dimension is exponential in the size of the quantum system being simulated). Feynman conceived of quantum computers as a means of simulating nature much more efficiently.

The evidence to this day is that quantum computers can indeed perform certain tasks more efficiently than classical computers. Perhaps the best-known example is Shor's factoring algorithm, by which a quantum computer can find the prime factors of any integer in probabilistic polynomial time [15]. There is no known classical probabilistic algorithm which can solve this problem in polynomial time. In the ten years since the publication of Shor's result, there has been an enormous surge of research in quantum algorithms and quantum complexity theory.

2 Quantum Programming Languages

Quantum physics involves phenomena, such as superposition and entanglement, whose properties are not always intuitive. These same phenomena give quantum computation its power, and are often at the heart of an interesting quantum algorithm. However, there does not yet seem to be a unifying set of principles by which quantum algorithms are developed; each new algorithm seems to rely on a unique set of "tricks" to achieve its particular goal.

One of the goals of programming language design is to identify and promote useful "high-level" concepts — abstractions or paradigms which allow humans

© Springer-Verlag Berlin Heidelberg 2004

to think about a problem in a conceptual way, rather than focusing on the details of its implementation. With respect to quantum programming, it is not yet clear what a useful set of abstractions would be. But the study of quantum programming languages provides a setting in which one can explore possible language features and test their usefulness and expressivity. Moreover, the definition of prototypical programming languages creates a unifying formal framework in which to view and analyze existing quantum algorithm.

2.1 Virtual Hardware Models

Advances in programming languages are often driven by advances in compiler design, and vice versa. In the case of quantum computation, the situation is complicated by the fact that no practical quantum hardware exists yet, and not much is known about the detailed architecture of any future quantum hardware.

To be able to speak of "implementations", it is therefore necessary to fix some particular, "virtual" hardware model to work with. Here, it is understood that future quantum hardware may differ considerably, but the differences should ideally be transparent to programmers and should be handled automatically by the compiler or operating system. There are several possible virtual hardware models to work with, but fortunately all of them are equivalent, at least in theory. Thus, one may pick the model which fits one's computational intuitions most closely.

Perhaps the most popular virtual hardware model, and one of the easiest to explain, is the *quantum circuit model.* Here, a quantum circuit is made up from quantum gates in much the same way as a classical logic circuit is made up from logic gates. The difference is that quantum gates are always reversible, and they correspond to unitary transformations over a complex vector space. See e.g. [3] for a succinct introduction to quantum circuits. Of the two basic quantum operations, unitary transformations and measurements, the quantum circuit model emphasizes the former, with measurements always carried out as the very last step in a computation.

Another virtual hardware model, and one which is perhaps even better suited for the interpretation of quantum programming languages, is the *QRAM model* of Knill [9]. Unlike the quantum circuit model, the QRAM models allows unitary transformations and measurements to be freely interleaved. In the QRAM model, a quantum device is controlled by a universal classical computer. The quantum device contains a large, but finite number of individually addressable quantum bits, much like a RAM memory chip contains a multitude of classical bits. The classical controller sends a sequence of instructions, which are either of the form "apply unitary transformation U to qubits i and j" or "measure qubit i". The quantum device carries out these instruction, and responds by making the results of the measurements available.

A third virtual hardware model, which is sometimes used in complexity theory, is the *quantum Turing machine.* Here, measurements are never performed, and the entire operation of the machine, which consists of a tape, head, and finite control, is assumed to be unitary. While this model is theoretically equivalent to the previous two models, it is not generally considered to be a very realistic approximation of which a future quantum computer might look like.

2.2 Imperative Quantum Programming Languages

The earliest proposed quantum programming languages followed the *imperative* programming paradigm. This line of languages was started by Knill [9], who gave a set of conventions for writing quantum algorithms in pseudo-code. While Knill's proposal was not very formal, it was very influential in the design of later imperative quantum programming languages. More complete imperative languages were defined by Ömer [10], Sanders and Zuliani [13], and Bettelli et al. [2].

A common feature of these imperative quantum programming languages is that a program is viewed as a sequence of operations which operate by updating some global state. These languages can be directly compiled onto (or interpreted in) the QRAM virtual hardware model. Quantum states in this paradigm are typically realized as *arrays* of qubits, and run-time checks are needed to detect certain error conditions. For instance, out-of-bounds checks are necessary for array accesses, and distinctness checks must be used to ensure $i \neq j$ when applying a binary quantum operation to two qubits i and j. As is typical for imperative programming languages, the type system of these languages is not rich enough to allow all such checks to be performed at compile-time. Also, typically these languages do not have a formal semantics, with the exception of Sanders and Zuliani's language, which possesses an operational semantics.

The various languages in this category each offer a set of advanced programming features. For instance, Ömer's language QCL contains such features as, automatic scratch space management, and a rich language for describing user-defined operators [10]. It also offers some higher-order operations such as computing the inverse of a user-defined operator.

The language of Bettelli et al. emphasizes practicality. It is conceived as an extension of C++, and it treats quantum operators as first-class objects which can be explicitly constructed and manipulated at run-time [2]. One of the most powerful features of this language is the on-the-fly optimization of quantum operators, which is performed at run-time.

Finally, Sanders and Zuliani's language qGCL is of a somewhat different flavor [13]. Based on Dijkstra's guarded command language, qGCL is as much a specification language as a programming language, and it supports a mechanism of stepwise refinement which can be used to systematically derive and verify programs.

2.3 Functional Quantum Programming Languages

In the functional programming style, programs do not operate by updating a global state, but by mapping specific inputs to outputs. The data types associated with purely functional languages (such as lists, recursive types) are more amenable to compile time analysis than their imperative counterparts (such as arrays). Consequently, even in very simple functional programming languages, many run-time checks can be avoided in such languages in favor of compile-time analysis.

The first proposal for a functional quantum programming language was made in [14]. In this paper, a language QFC is introduced, which represents programs via a functional version of flow charts. The language also has an alternative, text-based syntax. Both unitary operations and measurements are directly built into the language, and are handled in a type-safe way. Classical and quantum features are integrated within the same formalism. There are no run-time type checks or errors. The language can be compiled onto the QRAM model, and it also possesses a complete denotational semantics, which can be used to formally reason about programs. The denotational semantics uses complete partial orders of superoperators, and loops and recursion are interpreted as least fixpoints in the way which is familiar from domain-theoretic semantics.

The basic quantum flow chart language of [14] is functional, in the sense of being free of side-effects. However, functions are not themselves treated as data, and thus the language lacks the higher-order features typical of most functional programming languages such as ML or Haskell. It is however possible to extend the language with higher-order features. The main technical difficulty concerns the proper handling of linearity; here, one has to account for the fact that quantum information, unlike classical information, cannot be duplicated due to the so-called "no-cloning property". Van Tonder, in a pair of papers [16,17], has described a linear lambda calculus for quantum computation, with a type system based on Girard's linear logic [7].

Van Tonder's calculus is "purely" quantum, in the sense that it does not incorporate classical data types, nor a measurement operation. If one further extends the language with classical features and a measurement primitive, then a purely linear type system will no longer be sufficient; instead, one needs a system with linear and non-linear types. Such a language, with intuitionistic linear logic as its type system, will be presented in a forthcoming paper by Benoît Valiron.

We should also mention that there is some interesting work on using functional languages to *simulate* quantum computation. For instance, Sabry [12] shows how to model quantum computation in Haskell.

3 Semantics

The basic flow chart language of [14] has a satisfactory denotational semantics, but it lacks many language features that would be desirable, including higher-order features and side-effects. In trying to add new language features, one may either work syntactically or semantically. Semantic considerations, in particular, may sometimes suggest useful abstractions that are not necessarily apparent from a syntactic point of view. We briefly comment on some semantic projects.

Girard [8] recently defined a notion of quantum coherent spaces as a possible semantics for higher-order quantum computation. The class of quantum coherent spaces has good closure properties, for instance, it forms a *-autonomous category. However, the model is still incomplete, because the interpretation of quantum languages in this category is currently limited to the "perfect", or purely linear, fragment of linear logic. This means that classical data is subject to the same non-duplication restriction as quantum data in this model.

A different approach to a semantics for higher-order quantum computation is given by Abramsky and Coecke [1]. This work is more qualitative in nature, and relies on entanglement and quantum measurement to model higher-order functions and their applications, respectively.

Also on the semantic side, there have been a number of works on possible connections between quantum theory and domain theory. For instance, Edalat [5] gives a domain-theoretic interpretation of Gleason's theorem in the presence of partial information. Coecke and Martin [4] give a domain-theoretic treatment of the von Neumann entropy of a quantum state.

3.1 Topological Quantum Computation

An radically different direction in the semantics of quantum computation, and one which might lead to the discovery of new conceptual paradigms for quantum computation, is the work of Freedman, Kitaev, and Wang [6]. This line of work seeks to exploit connections between quantum computation and topological quantum field theories (TQFT's). In a nutshell, in topological quantum computation, a quantum state is represented by a physical system which is resistant to small perturbations. Thus, quantum operations are determined only by global topological properties, e.g., linking properties of the paths traversed by some particles. This leads to a potentially very robust model of quantum computation. It also suggests that there is a more discrete, combinatorial way of viewing quantum computation, which might in turns suggest new quantum algorithms. These topological approaches to quantum computation are currently limited to a description of unitary operators; measurements are not currently considered within this model.

4 Challenges

There are many remaining challenges in the design and analysis of quantum programming languages. One such challenge is to give a sound denotational semantics for a higher-order quantum programming language, including classical features and measurement. While there has been recent progress on this issue, both on the syntactic side and on the semantic side, the connection between syntax and semantics remains tenuous at this point, and typically covers only fragments of the language. A related question is how to model infinite data types, particularly types which include an infinite amount of "quantum" data.

Another challenge is to formulate a theory of "quantum concurrency". This is not far-fetched, as one can easily imagine networks of quantum processes which communicate by exchanging classical and quantum data. There is a considerable body of work in quantum information theory and quantum cryptography, which suggests some potential applications for quantum concurrent systems.

Another interesting research area is the implementation of quantum programming languages on imperfect hardware. Unlike the idealized "virtual machine" models of quantum computation, one may assume that real future implementations of quantum computation will be subject to the effects of random errors and decoherence. There are known error correction techniques for quantum information, but it is an interesting question to what extent such techniques can be automated, for instance, by integrating them in the compiler or operating system, or to what extent specific algorithms might require customized error correction techniques.

References

1. S. Abramsky and B. Coecke. Physical traces: Quantum vs. classical information processing. In R. Blute and P. Selinger, editors, *Proceedings of Category Theory and Computer Science, CTCS'02,* ENTCS 69. Elsevier, 2003.
2. S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming. arXiv:cs.PL/0103009 v2, Nov. 2001.
3. R. Cleve. An introduction to quantum complexity theory. In C. Macchiavello, G. Palma, and A. Zeilinger, editors, *Collected Papers on Quantum Computation and Quantum Information Theory,* pages 103–127. World Scientific, 2000.
4. B. Coecke and K. Martin. A partial order on classical and quantum states. Technical report, Oxford University Computing Laboratory, 2002. PRG-RR-02-07.
5. A. Edalat. An extension of Gleason's theorem for quantum computation. http://www.doc.ic.ac.uk/~ae/papers.html, 2003.
6. M. H. Freedman, A. Kitaev, and Z. Wong. Simulation of topological field theories by quantum computers. arXiv:quant-ph/0001071/ v3, Mar. 2000.
7. J.-Y. Girard. Linear logic. *Theoretical Comput. Sci.,* 50:1–102, 1987.
8. J.-Y. Girard. Between logic and quantic: a tract. Manuscript, Oct. 2003.
9. E. H. Knill. Conventions for quantum pseudocode. LANL report LAUR-96-2724, 1996.
10. B. Ömer. A procedural formalism for quantum computing. Master's thesis, Department of Theoretical Physics, Technical University of Vienna, July 1998. http://tph.tuwien.ac.at/~oemer/qcl.html.
11. J. Preskill. Quantum information and computation. Lecture Notes for Physics 229, California Institute of Technology, 1998.
12. A. Sabry. Modeling quantum computing in Haskell. In *ACM SIGPLAN Haskell Workshop,* 2003.
13. J. W. Sanders and P. Zuliani. Quantum programming. In *Mathematics of Program Construction,* Springer LNCS 1837, pages 80–99, 2000.
14. P. Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science.* To appear.
15. P. Shor. Algorithms for quantum computation: discrete log and factoring. In *Proceedings of the 35th IEEE FOCS,* pages 124–134, 1994.
16. A. van Tonder. A lambda calculus for quantum computation. arXiv:quant-ph/0307150/ v4, Dec. 2003.
17. A. van Tonder. Quantum computation, categorical semantics and linear logic. arXiv:quant-ph/0312174/ v1, Dec. 2003.

Analysis of Synchronous and Asynchronous Cellular Automata Using Abstraction by Temporal Logic

Masami Hagiya[1], Koichi Takahashi[2], Mitsuharu Yamamoto[3], and Takahiro Sato[1]

[1] Graduate School of Information Science and Technology, University of Tokyo
[2] National Institute of Advanced Industrial Science and Technology
[3] Faculty of Science, Chiba University

Abstract. We have been studying abstractions of linked structures, in which cells are connected by pointers, using temporal logic. This paper presents some our results for these abstractions. The system to be verified is a transition system on a graph. The shape of the graph does not change as a result of the transition, but the label assigned to each cell (node) changes according to rewrite rules. The labels of cells are changed synchronously or asynchronously. We abstract such systems using abstract cells and abstract graphs. Abstract cells are characterized by a set of temporal formulas, and different abstractions can be tried by changing the set of formulas. Some examples of analysis are also described.

1 Introduction

Previously, we introduced a method for defining abstractions of heap structures in which cells are connected by pointers, mainly for the purpose of verifying algorithms for concurrent garbage collection [1, 2]. The basic strategy of the method is to abstract each cell in a heap structure in terms of regular expressions taken from a fixed finite set. Each regular expression represents a property of a cell concerning its connectivity with other cells. For example, the regular expression w*g holds for a cell that can reach a gray cell via white cells, where w is the label of a white cell and g is that of a gray cell (Figure 1).

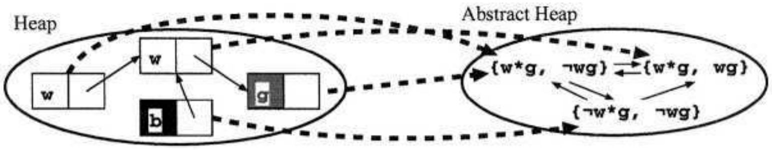


Fig. 1. Abstraction of a heap structure by regular expressions

Then, we introduced branching-time temporal logic for representing such properties [3]. For example, the regular expression w*g can be replaced with the

© Springer-Verlag Berlin Heidelberg 2004

temporal formula **E**(w **until** g) in CTL (computation tree logic) [4, 5]. Notice that the modality in temporal formulas corresponds to the connectivity in the heap structure. Using temporal logic, one can easily determine the relationship between the properties represented by formulas by deriving the implication or equivalence of the formulas.

In our method, we first fix a finite set of temporal formulas F. We abstract a cell in a heap according to whether each formula in F holds for the cell (Figure 2). Formally, let L denote a heap structure and s be a cell in the heap. We use $\alpha(L, s)$ to denote the set of formulas in F that holds at s, i.e., $\{\phi \in F \mid s \models_L \phi\}$. We call $\alpha(L, s)$ an abstract cell. In the simplest setting, an entire heap structure is abstracted by the set of all abstract cells corresponding to the cells in the heap. We call a set of abstract cells an abstract heap. Conversely, ordinary heaps are called concrete heaps.

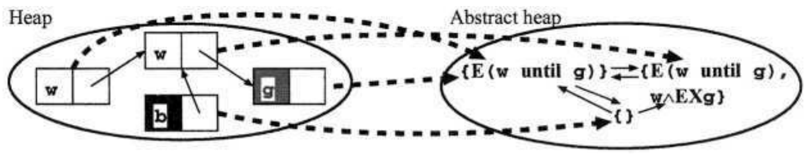


Fig. 2. Abstraction of a heap structure (F = {**E**(w **until** g),w ∧ **EX**g})

This abstraction method is similar to predicate abstraction, which has been well studied recently and is used for software model checking [6, 7]. In predicate abstraction, one introduces a set of predicates on states. Then, a state is abstracted using a set of predicates that hold at the state. Note that in ordinary predicate abstraction, the states change over time, while in our abstraction method, cells range over a heap structure and temporal formulas are used as predicates on cells.

If we want to verify an algorithm operating on a heap structure, we should formulate it as a state transition system whose state is a heap. One step of the state transition may change the label of a cell, change the connectivity of a cell, delete a cell, add a new cell, etc. A state transition on a heap structure induces a state transition on an abstract heap. Therefore, if one can effectively compute state transitions on abstract heaps, one can verify properties of the state transition system on concrete heaps. Previously, we verified the safety property of concurrent garbage collection.

The work reported here differs from our previous work in the following two points:

– In our previous work, we used ordinary CTL and CTL*, so forward and backward connectivities were treated separately. In this work, we introduce 2CTL (2-way computation tree logic), which allows us to treat forward and backward connectivities simultaneously [8].

– We analyze both synchronous and asynchronous transition systems in a uniform setting. Typical examples of synchronous transition systems composed of cells are cellular automata[9, 10]. By contrast, distributed systems are usually asynchronous.

This paper deals only with state transitions that change only the labels of cells. State transitions that change the connectivity of cells, or delete/add cells are left for future work.

The paper is organized as follows. Section 2 explains a concrete system that describes the target of verification. 2-Way CTL, the formal logic expressing the characteristics of a linked structure, is also explained in this section. Section 3 introduces an abstract system that uses temporal logic to abstract the states of cells linked by pointers. Some examples are shown in Sect. 4. We show how to check satisfiability, which plays a central role in abstraction, in Sect. 5. Section 6 contains our conclusions and remarks on future work.

2 Concrete System and 2-Way Computation Tree Logic

2.1 Concrete System

Let S be a set of cells, and $\mathcal{A}$ a non-empty finite set of labels for links. Then, we consider the graph $G = (S, \{R_a\})$ defined by S and a set $\{R_a \subseteq S \times S \mid a \in \mathcal{A}\}$ of labeled links between cells . We assume that for each label $a \in \mathcal{A}$, there exists $\overline{a} \in \mathcal{A}$ such that

$$R_{\overline{a}} = \{\langle s_1, s_2\rangle \mid \langle s_2, s_1\rangle \in R_a\},$$

and $\overline{\overline{a}} = a$.

Let P be a set of labels for cells. We call the labeling of cells with elements of P a *state of cells*. We consider a transition system on the graph G such that it changes the state of cells on the graph, but does not change the structure defined by G. The system is denoted by $\langle S, \{R_a\}, L_0, \Delta\rangle$, where $L_0 : S \to P$ is the initial state of the cells, and Δ is a finite set of rewrite rules. Each rewrite rule has the form $\phi \to p$, where ϕ is a formula in 2-way computation tree logic (2CTL), which will be defined later, and $p \in P$.

There are two types of transition: *synchronous* and *asynchronous*. We say that a state L of the cells can make a synchronous transition to L' if the following condition is satisfied:

$$\forall s \in S.\ \exists(\phi \to p) \in \Delta.\ (L'(s) = p) \wedge (s \models_L \phi)$$

where $s \models_L \phi$ denotes that ϕ holds at s in 2CTL, which is also explained in detail below. Similarly, a state L of cells can make an asynchronous transition to L' if the following condition is satisfied:

$$\exists s \in S.\ (\exists(\phi \to p) \in \Delta.\ (L'(s) = p) \wedge (s \models_L \phi)) \wedge$$
$$(\forall s' \in S.\ s' \neq s \Longrightarrow L'(s') = L(s'))$$

Namely, a synchronous transition represents the situation in which all the cells are rewritten simultaneously, and an asynchronous one represents the situation in which exactly one cell is rewritten.

Hereinafter, we call a cell/graph/transition/system defined as above a concrete cell/graph/transition/system, respectively, in contrast to their abstract counterparts, which are defined later.

2.2 2-Way Computation Tree Logic (2CTL)

In this section, we explain the 2-way computation tree logic (2CTL) that is used in the definition of transitions in concrete systems. 2CTL is an extension of the usual CTL with inverse modality. Several logics with inverse modality have been studied already. For example, Vardi gave a proof procedure based on automata theory for 2-way **μ-calculus** [8].

A (positive form) formula in 2CTL is defined as follows:

$$\phi ::= p \mid \neg p \mid M\phi \mid \phi \wedge \phi \mid \phi \vee \phi$$

where $p \in P$ is a label for a cell in a concrete system, and it represents an atomic proposition in 2CTL. M is one of the following modal operators:

$\mathsf{AX}_A \quad \mathsf{EX}_A \quad \mathsf{AG}_A \quad \mathsf{EG}_A \quad \mathsf{AF}_A \quad \mathsf{EF}_A$

where A is a set of modality labels that corresponds to a set $A \subseteq \mathcal{A}$ of labels for links in a concrete system. As we mentioned in Sect. 2.1, there exists an inverse modality label $\overline{a}$ for each modality label $a \in \mathcal{A}$, and $\overline{\overline{a}}$ equals a. We simply write AX_a for $\mathsf{AX}_{\{a\}}$. Although we can add an **until** operator, this is excluded from this paper for brevity.

The tuple $\langle S, \{R_a\}, L\rangle$ made of a concrete graph and a state of concrete cells can be regarded as a Kripke structure. In the rest of this subsection, we define the semantics of 2CTL positive form formulas with this structure.

We say that an infinite sequence $s_0, s_1, \cdots$ of cells is an A-path if for all i, there exists $a \in A$ such that $s_i R_a s_{i+1}$. We say that a finite sequence $s_0, s_1, \cdots, s_n$ is an A-path if for $i < n$, there exists $a \in A$ such that $s_i R_a s_{i+1}$, and there is no $a \in A$ and $t \in S$ satisfying $s_n R_a t$.

For cell s and formula ϕ, the relation $s \models_L \phi$ is defined as follows:

- $s \models_L p$ **iff** $p = L(s)$
- $s \models_L \mathsf{AX}_A\phi$ **iff** for any $a \in A$ and $t \in S$ satisfying sR_at, $t \models_L \phi$ holds.
- $s \models_L \mathsf{EX}_A\phi$ **iff** there exists $a \in A$ and $t \in S$ such that sR_at and $t \models_L \phi$.
- $s \models_L \mathsf{AG}_A\phi$ **iff** for any A-path $s = s_0, s_1, \cdots$ starting from s, $s_i \models_L \phi$ holds for all i.
- $s \models_L \mathsf{EG}_A\phi$ **iff** there exists an infinite A-path $s = s_0, s_1, \cdots$ starting from s such that $s_i \models_L \phi$ holds for all i.
- $s \models_L \mathsf{AF}_A\phi$ **iff** for any infinite A-path $s = s_0, s_1, \cdots$ starting from s, there exists i such that $s_i \models_L \phi$.
- $s \models_L \mathsf{EF}_A\phi$ **iff** there exists an A-path $s = s_0, s_1, \cdots$ starting from s and i such that $s_i \models_L \phi$.

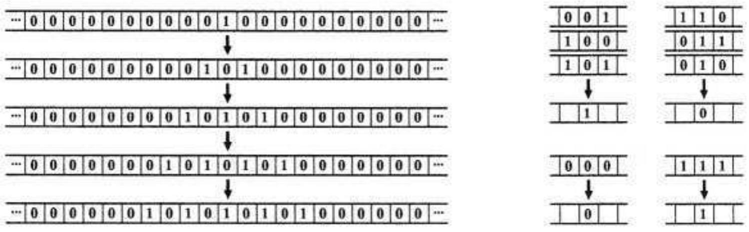
We sometimes omit L from $s \models_L \phi$ when it is obvious from the context.

If there exists a Kripke structure with state s satisfying $s \models \phi$, we say that ϕ is *satisfiable*. Satisfiability in 2CTL plays an important role in constructing the abstract graphs that are defined below.

2.3 Examples of Concrete Systems

In this section, we give some examples of concrete systems such that $\mathcal{A} = \{\rightarrow, \leftarrow\}$ (where $\overline{\rightarrow} = \leftarrow$). In this case, we consider $sR_{\rightarrow}s'$ to represent the situation in which s' is the *right* neighbor of s. We introduce some shorthand for the set of rewrite rules: $[q_1 \ldots q_n]p[r_1 \ldots r_m] \rightarrow p'$ represents $\{p \wedge \mathsf{EX}_{\leftarrow}q \wedge \mathsf{EX}_{\rightarrow}r \rightarrow p' \mid q \in \{q_1, \ldots, q_n\},\ r \in \{r_1, \ldots, r_m\}\}$. Moreover, $.p[r_1 \ldots r_m] \rightarrow p'$ is used as shorthand for $\{p \wedge \mathsf{EX}_{\leftarrow}q \wedge \mathsf{EX}_{\rightarrow}r \rightarrow p' \mid q \in P,\ r \in \{r_1, \ldots, r_m\}\}$. We define $[q_1 \ldots q_n]p.$ and $.p.$ similarly. When we use this shorthand, we may write $\Delta_1 \cup \Delta_2$ as $\Delta_1 \mid \Delta_2$ for sets Δ_1, Δ_2 of rewrite rules.

Synchronous Transition Let $S = \mathbb{Z}$, $R_{\rightarrow} = \{\langle i, i+1 \rangle \mid i \in \mathbb{Z}\}$, $P = \{0,1\}$, $L_0(0) = 1$, $L_0(s) = 0\ (s \neq 0)$, and the rewrite rules be $\Delta = [1]0. \rightarrow 1 \mid .0[1] \rightarrow 1 \mid [0]1. \rightarrow 0 \mid .1[0] \rightarrow 0 \mid [0]0[0] \rightarrow 0 \mid [1]1[1] \rightarrow 1$. This system can be illustrated as follows:



In this figure, the figure on the right represents the rewrite rules for the center cell. With these rules, a cell labeled 0 becomes 1 if at least one neighbor is 1, and a cell labeled 1 becomes 0 if at least one neighbor is 0. The uppermost infinite sequence of cells in the figure on the left represents the initial states of the cells, and they are changed simultaneously according to the rewrite rules. This is an example of a 1-dimensional cellular automaton.

Asynchronous Transition Let n be an integer greater than 0. Moreover, let $S = \{0, 1, \ldots, n\}$, $R_{\rightarrow} = \{\langle i, i+1 \rangle \mid i = 0, \ldots, n-1\} \cup \{\langle n, 0 \rangle\}$, $P = \{\mathbf{T}, \mathbf{U}, \mathbf{E}, \mathbf{t}, \mathbf{u}, \mathbf{e}\}$, and the rewrite rules be $\Delta = .\mathbf{T}[\mathbf{TUt}] \rightarrow \mathbf{U} \mid [\mathbf{Ttu}]\mathbf{U}. \rightarrow \mathbf{E} \mid .\mathbf{E}. \rightarrow \mathbf{T} \mid [\mathbf{Ttu}]\mathbf{t}. \rightarrow \mathbf{u} \mid .\mathbf{u}[\mathbf{TUt}] \rightarrow \mathbf{e} \mid .\mathbf{e}. \rightarrow \mathbf{t}$.

This is an example of the so-called dining philosophers. Cell 0 corresponds to the philosopher who takes the left fork first (left-handed), and the others correspond to those who take the right fork first (right-handed). States **T**, **U**, and **E** are for right-handed philosophers, and represent thinking, picking up the right fork, and eating, respectively. Similarly, states **t**, **u**, and **e** are for the left-handed philosopher, and represent thinking, picking up the left fork, and eating, respectively.

3 Abstract Systems

3.1 Overview of Analysis Using Abstract Systems

In this section, we consider an abstraction of the concrete systems given in Sect. 2.1 using abstract cells/graphs/transitions. The analysis involves by the following procedure:

1. Compute the initial abstract graph $\alpha(L_0)$ from the initial concrete state L_0 of the cells (Section 3.2).
2. Iteratively compute the abstract graphs that are reachable with respect to abstract transitions (Section 3.3).
3. With these operations, exhaustively collect the concrete graphs that are reachable in the concrete system using abstract graphs. Then, verify their properties on abstract graphs.

As explained below, an abstract graph is constructed from a finite set of abstract cells. Therefore, the number of abstract graphs is also finite.

We define the abstract system so that it has the soundness property with respect to the concrete system in the following sense:

– If L is the state of cells after n steps from the initial state, then for any concrete cell s, there exists the corresponding abstract cell $\alpha(L, s)$ in an abstract graph that is reachable by n steps from the initial abstract state.
– If L is the state of cells after n steps from the initial state, then for any concrete cells s and s' such that sR_as', there exists an abstract link labeled a from $\alpha(L, s)$ to $\alpha(L, s')$ in an abstract graph that is reachable by n steps from the initial abstract state.

Namely, we define an abstract system so that the abstract graph can conservatively simulate the neighborhood in the concrete system. With this soundness property, if we can show a safety modal formula, i.e., one that uses only AX_a and AG_a, then it is also satisfied in the concrete system.

3.2 Abstract Cells and Abstract Graphs

In the following, we make extensive use of the derivation in 2CTL when constructing an abstract system. When we derive formulas in 2CTL, we can use not only 2CTL theorems but also the properties of concrete cells as axioms. In the previous example of a cellular automaton, since the number of cells to the right and left of each cell is exactly one, $\mathsf{AX}_{\rightarrow}\phi$ and $\mathsf{EX}_{\rightarrow}\phi$ are equivalent. Likewise, $\mathsf{AX}_{\leftarrow}\phi$ and $\mathsf{EX}_{\leftarrow}\phi$ are equivalent. Hence, the negation of $\mathsf{AX}_{\rightarrow}\mathbf{0}$ is equivalent to $\mathsf{AX}_{\rightarrow}\mathbf{1}$ because the negation of $\mathbf{0}$ is $\mathbf{1}$. Likewise, the negation of $\mathsf{AX}_{\rightarrow}\mathsf{AX}_{\rightarrow}\mathsf{AG}_{\rightarrow}\mathbf{0}$ is equivalent to $\mathsf{AX}_{\rightarrow}\mathsf{AX}_{\rightarrow}\mathsf{EF}_{\rightarrow}\mathbf{1}$. We implicitly assume such axioms determined by the area of the problem in reasoning with 2CTL.

An abstract system is defined as a transition system on abstract graphs. In turn, an abstract graph is defined in terms of abstract cells. Therefore, we begin by explaining abstract cells.

To define abstract cells, we have to give a finite set $F \supseteq P$ of 2CTL formulas that includes all the elements in P. We can try different kinds of abstraction by changing the way the set F is selected.

For $C \subseteq F$, we write ϕ_C for

$$\left(\bigwedge_{\phi \in C} \phi\right) \wedge \left(\bigwedge_{\phi \in F-C} \neg\phi\right).$$

We consider subset C to represent a concrete cell at state L of cells such that $s \models_L \phi_C$. Therefore, we call subset C of F an *abstract cell* if ϕ_C is satisfiable and it contains exactly one element in P. Otherwise, there are no corresponding concrete cells so we do not regard it as an abstract cell.

We have to choose F so that the truth or falsehood of the rewriting conditions (i.e., ϕ in $(\phi \rightharpoonup p) \in \Delta$) in the concrete system can be determined from ϕ_C for each concrete cell C.

For abstract cells C, D, and a label $a \in \mathcal{A}$, we can place an *abstract link* labeled a from C to D if both $\phi_C \wedge \mathsf{EX}_a\phi_D$ and $\phi_D \wedge \mathsf{EX}_{\overline{a}}\phi_C$ are satisfiable.

Given a set $\mathcal{A}_R \subseteq 2^{\mathcal{A}}$ of subsets of modality labels $\mathcal{A}$, we consider the *reachability formulas* for a set of abstract cells. We introduce an atomic proposition p_C for each concrete cell C. Then, the reachability from an abstract cell C to either of the abstract cells $D_1, \ldots, D_n$ using the modality labels in $A \in \mathcal{A}_R$ is expressed using the following formula:

$$p_C \supset \mathsf{EF}_A(p_{D_1} \vee \cdots \vee p_{D_n}).$$

This formula represents the situation in which for any concrete cell s that is represented by the abstract cell C, there exists a finite A-path from s to t such that t is represented by either of the abstract cells $D_1, \ldots, D_n$.

An abstract graph $(\mathcal{C}, \{\mathcal{L}_a\}, \mathcal{R})$ is defined using the components mentioned above:

– A set $\mathcal{C}$ of abstract cells.
– A set $\{\mathcal{L}_a\}$ of labeled abstract links that satisfy the above condition.
– A set $\mathcal{R}$ of reachability formulas.

Given a concrete graph $G = (S, \{R_a\})$, a state L of concrete cells, a set F of 2CTL formulas, and a set $\mathcal{A}_R$ of subsets of modality labels, we define an abstract graph $\alpha(L) = (\mathcal{C}, \{\mathcal{L}_a\}, \mathcal{R})$ as follows:

– Define $\mathcal{C} = \{\alpha(L, s) \mid s \in S\}$ where $\alpha(L, s)$ is an abstract cell corresponding to a concrete cell s defined as follows:

$$\alpha(L, s) = \{\phi \in F \mid s \models_L \phi\}.$$

– $\{\mathcal{L}_a\}$ is defined as $\forall C_1 C_2 \in \mathcal{C}.\ C_1 \mathcal{L}_a C_2 \iff \exists s_1 s_2.\ s_1 R_a s_2$ and $C_1 = \alpha(L, s_1)$ and $C_2 = \alpha(L, s_2)$. Note that this $\{\mathcal{L}_a\}$ satisfies the condition for abstract links.

– $\mathcal{R}$ is a set of reachability formulas derived from the reachability between corresponding concrete cells, i.e.,

$$\begin{aligned}\mathcal{R} = \{&p_C \supset \mathsf{EF}_A p_D \mid A \in \mathcal{A}_{\mathrm{R}}, \forall s.\, C = \alpha(L,s) \Longrightarrow \\ &\exists t.\, D = \alpha(L,t) \text{ and there exists a finite } A\text{-path from } s \text{ to } t \text{ in } G\}\end{aligned}$$

Then, the following properties hold for arbitrary L:

– For a concrete cell s, there exists an abstract cell $\alpha(L,s)$ in $\alpha(L)$.
– If sR_as' holds in a concrete graph, then there is an abstract link labeled a from $\alpha(L,s)$ to $\alpha(L,s')$ in $\alpha(L)$.

3.3 Abstract Transitions

An abstract transition involves the following three steps:

1. Rewrite the abstract cells according to their formulas and the rewrite rules in the concrete system.
2. Reevaluate the formulas in the abstract cells.
3. Merge the abstract cells having the same set of formulas.

In the first step, a new state $q \in P$ is computed for each abstract cell C using the rewrite rules. For an abstract graph $(\mathcal{C}, \{\mathcal{L}_a\}, \mathcal{R})$, a tuple $(\mathcal{C}^1, \{\mathcal{L}_a^1\}, \mathcal{R}^1)$ of a graph and a reachability formulas is called a result of a rewrite if the following condition is satisfied:

– For a synchronous transition:

$$\begin{aligned}&(\mathcal{C}^1 = \{(C,q) \mid C \in \mathcal{C},\ \exists(\phi \to q) \in \Delta.\ \phi_C \supset \phi \text{ is derivable}\} \land \\ &(\forall (C,q),(C',q') \in S'.\ (C,q)\mathcal{L}_a^1(C',q') \Longleftrightarrow C\mathcal{L}_aC') \land \\ &(\mathcal{R}^1 = \{p_{(C,q)} \supset \mathsf{EF}_A(\textstyle\bigvee_i \bigvee_j p_{(D_i,q_{ij})}) \mid p_C \supset \mathsf{EF}_A(\bigvee_i p_{D_i}) \in \mathcal{R}, \\ &\qquad\qquad (C,q) \in \mathcal{C}^1,\ (D_i,q_{ij}) \in \mathcal{C}^1\})\end{aligned}$$

– For an asynchronous transition:

$$\begin{aligned}&(\exists C \in \mathcal{C}.\ \exists(\phi \to q) \in \Delta.\ \phi_C \supset \phi \text{ is derivable} \land \\ &\qquad \mathcal{C}^1 = \{(C,q)\} \cup \{(C',q') \mid C' \in \mathcal{C}, \{q'\} = C \cap P\}) \land \\ &(\forall (C,q),(C',q') \in \mathcal{C}^1.\ (C,q)\mathcal{L}_a^1(C',q') \Longleftrightarrow C\mathcal{L}_aC') \land \\ &(\mathcal{R}^1 = \{p_{(C,q)} \supset \mathsf{EF}_A(\textstyle\bigvee_i \bigvee_j p_{(D_i,q_{ij})}) \mid p_C \supset \mathsf{EF}_A(\bigvee_i p_{D_i}) \in \mathcal{R}, \\ &\qquad\qquad (C,q) \in \mathcal{C}^1,\ (D_i,q_{ij}) \in \mathcal{C}^1\})\end{aligned}$$

A pair $(C,q) \in \mathcal{C}^1$ represents the situation in which the state of a concrete cell that is represented by C is rewritten to $q \in P$. For a synchronous transition, an abstract cell C can represents multiple concrete cells that can be rewritten to different results if the rewrite rules are nondeterministic. Thus we consider all the possible results of the rewrite at once.

For a synchronous transition, the result of rewrite $(\mathcal{C}^1, \{\mathcal{L}_a^1\}, \mathcal{R}^1)$ is determined uniquely if it exists. By contrast, there may be multiple results of a rewrite for an asynchronous transition. In such a case, the reevaluation step (the second step) is performed for each result of the rewrite.

Next, we reevaluate a set of formulas in F for each abstract cell with a new state $(C, q) \in \mathcal{C}^1$. Namely, for each $\phi \in F$, we examine whether ϕ or $\neg\phi$ holds (or we cannot determine which of them holds) at the concrete cells that (C, q) represents.

This determination is done by checking the satisfiability of $p_{(C,q)} \wedge \phi$ and that of $p_{(C,q)} \wedge \neg\phi$ under the following assumptions:

- Axioms determined by the area of the problem.
- $p_{(C,q)} \supset q$ for each $(C, q) \in \mathcal{C}^1$.
- $p_{(C,q)} \supset \neg q'$ if $q \neq q'$ for each $(C, q) \in \mathcal{C}^1$.
- $p_{(C,q)} \supset \mathsf{AX}_a(\bigvee\{p_{(C',q')} \mid (C,q)\mathcal{L}_a^1(C',q')\})$ for each $(C, q) \in \mathcal{C}^1$.
- Reachability formulas $\mathcal{R}^1$.

We simply say that $p_{(C,q)} \wedge \phi$ (or $p_{(C,q)} \wedge \neg\phi$) is satisfiable in $(\mathcal{C}^1, \{\mathcal{L}_a^1\}, \mathcal{R}^1)$ when it is satisfiable under the above assumptions.

In general, we cannot determine the truth/falsehood of non-universal formulas such as the one containing EF_a using only the first four assumptions above, because the existence of abstract links does not assure the existence of the corresponding concrete links. Therefore, we include the reachability formulas in an abstract system so that we can use more information to determine the truth/falsehood of formulas in F. In general, we can include any kinds of formulas among abstract cells other than reachability, provided the formulas can be inherited conservatively when the operations, such as rewriting, splitting, and merging, in abstract transitions are performed.

Even if we make use of reachability formulas, perhaps both $p_{(C,q)} \wedge \phi$ and $p_{(C,q)} \wedge \neg\phi$ are satisfiable in $(\mathcal{C}^1, \{\mathcal{L}_a^1\}, \mathcal{R}^1)$. Then, we have to split the case into one in which formula ϕ is true and one in which it is false. This corresponds to splitting an abstract cell, and it is expressed as the following function γ: for $(C, q) \in \mathcal{C}^1$, we define

$$\begin{aligned}\gamma(C,q) = \{(C,C') \mid{} & C' \subseteq F,\ C' \cap P = \{q\} \text{ and} \\ & \phi \in C' \cap F \Longrightarrow p_{(C,q)} \wedge \phi \text{ is satisfiable in } (\mathcal{C}^1, \{\mathcal{L}_a^1\}, \mathcal{R}^1) \text{ and} \\ & \phi \in F \setminus C' \Longrightarrow p_{(C,q)} \wedge \neg\phi \text{ is satisfiable in } (\mathcal{C}^1, \{\mathcal{L}_a^1\}, \mathcal{R}^1)\}\end{aligned}$$

The result of the reevaluation from $(\mathcal{C}^1, \{\mathcal{L}_a^1\}, \mathcal{R}^1)$ is defined as $(\mathcal{C}^2, \{\mathcal{L}_a^2\}, \mathcal{R}^2)$ where

$$\begin{aligned}& \mathcal{C}^2 = \bigcup\{\gamma(C,q) \mid (C,q) \in \mathcal{C}^1\} \\ & \forall (C_1, C_1'), (C_2, C_2') \in \mathcal{C}^2.\ (C_1, C_1')\mathcal{L}_a^2(C_2, C_2') \Longleftrightarrow \\ & \qquad (\exists q_1\, q_2.\ (C_1, q_1)\mathcal{L}_a^1(C_2, q_2),\ \{q_1\} = C_1' \cap P,\ \{q_2\} = C_2' \cap P) \text{ and} \\ & \qquad \text{both } \phi_{C_1'} \wedge \mathsf{EX}_a \phi_{C_2'} \text{ and } \phi_{C_2'} \wedge \mathsf{EX}_{\overline{a}} \phi_{C_1'} \text{ are satisfiable.} \\ & \mathcal{R}^2 = \{p_{(C,C')} \supset \mathsf{EF}_A(\textstyle\bigvee_i \bigvee\{p_{(D,D')} \mid (D,D') \in \gamma(D_i, q_i)\}) \mid \\ & \qquad \exists q.\ (C,C') \in \gamma(C,q) \text{ and } (p_{(C,q)} \supset \mathsf{EF}_A(\textstyle\bigvee_i p_{(D_i,q_i)})) \in \mathcal{R}^1\}\end{aligned}$$

Namely, the set $\mathcal{L}_a^2$ of links is constructed by inheriting links in $\mathcal{L}_a^1$ but excluding the ones that do not satisfy the condition for abstract links.

After the reevaluation of all the results of rewrites is finished, multiple abstract cells that have the same set of formulas are merged as the third step of an abstract transition. This merge operation enables us to bound the size of the abstract graph using the number of different abstract cells.

The result $(\mathcal{C}', \{\mathcal{L}'_a\}, \mathcal{R}')$ of a merge is defined as follows:

$$
\begin{aligned}
&\mathcal{C}' = \{C' \mid \exists C.\ (C, C') \in \mathcal{C}^2\}\\
&\forall C'_1, C'_2 \in \mathcal{C}'.\ C'_1 \mathcal{L}'_a C'_2 \iff \exists C_1\, C_2\, .(C_1, C'_1)\mathcal{L}^2_a(C_2, C'_2)\\
&\mathcal{R}' = \{p_{C'} \supset \mathsf{EF}_A(p_{D'_1} \vee \cdots \vee p_{D'_n}) \mid\\
&\quad \forall C.\ (C, C') \in \mathcal{C}^2 \Longrightarrow \exists D_1 \ldots D_m\, D''_1 \ldots D''_m.\\
&\qquad\qquad\qquad\qquad (p_{(C,C')} \supset \mathsf{EF}_A(p_{(D_1,D''_1)} \vee \cdots \vee p_{(D_m,D''_m)})) \in \mathcal{R}^2 \wedge\\
&\qquad\qquad\qquad\qquad \{D''_1, \ldots, D''_m\} \subseteq \{D'_1, \ldots, D'_n\}\}
\end{aligned}
$$

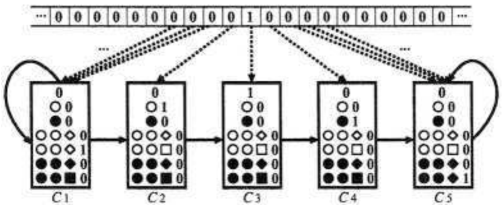
As for an asynchronous transition, the repeated computation of abstract transitions constructs a tree of transitions in general, because there are multiple results $(\mathcal{C}', \{\mathcal{L}'_a\}, \mathcal{R}')$ of the abstract transition for a single $(\mathcal{C}, \{\mathcal{L}_a\}, \mathcal{R})$. We may conservatively merge these results of the abstract transition using the merge operation similar to the above one in order to avoid the number of the set of abstract graphs becomes huge (although it is bounded).

4 Examples

4.1 Synchronous Transition: 1-Dimensional Cellular Automaton

Abstract Graph In this section, we give an abstraction of 1-dimensional cellular automaton as an example of a synchronous system. The concrete system was given in Sect. 2.3.

The following figure illustrates the abstract graph of the initial concrete graph. The dotted lines represent the abstraction function α.



In the above figure, ○, ●, □, ■, ◇, and ♦ are shorthands of $\mathsf{AX}_{\rightarrow}$, $\mathsf{AX}_{\leftarrow}$, $\mathsf{AG}_{\rightarrow}$, $\mathsf{AG}_{\leftarrow}$, $\mathsf{EF}_{\rightarrow}$, and $\mathsf{EF}_{\leftarrow}$, respectively.

In this example, we take F as the collection of the following formulas.

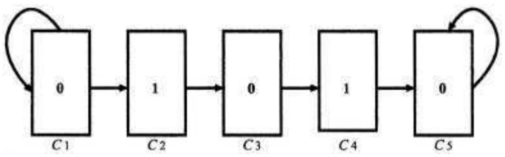
$$
\begin{array}{llll}
\mathbf{0} & \mathsf{AX}_{\rightarrow}\mathbf{0} & \mathsf{AX}_{\leftarrow}\mathbf{0} & \\
\mathsf{AX}_{\rightarrow}\mathsf{AX}_{\rightarrow}\mathsf{AG}_{\rightarrow}\mathbf{0} & \mathsf{AX}_{\rightarrow}\mathsf{AX}_{\rightarrow}\mathsf{AG}_{\rightarrow}\mathbf{1} & \mathsf{AX}_{\leftarrow}\mathsf{AX}_{\leftarrow}\mathsf{AG}_{\leftarrow}\mathbf{0} & \mathsf{AX}_{\leftarrow}\mathsf{AX}_{\leftarrow}\mathsf{AG}_{\leftarrow}\mathbf{1}
\end{array}
$$

As we noted earlier, there are axioms that can be derived from the area of the problem. For example, the negation of $\mathsf{AX}_{\rightarrow}\mathsf{AX}_{\rightarrow}\mathsf{AG}_{\rightarrow}\mathbf{0}$ is equivalent to

$\mathsf{AX}_{\rightarrow}\mathsf{AX}_{\rightarrow}\mathsf{EF}_{\rightarrow}\mathbf{1}$. So in the above figure, the formula corresponding to $\phi \in F - C$ is shown as the formula that is equivalent to the negation $\neg\phi$.

Abstract links labeled "$\rightarrow$" are derived from the concrete initial state are drawn in solid lines. Reachability formulas are $\{p_{C_i} \supset \mathsf{EF}_{\rightarrow}p_{C_j} \mid i \leq j\}$ if we name abstract cells $C_1, \ldots, C_5$ from left to right.

Abstract Transition Next, we compute the result of rewrite as the first step of the abstract transition. In this example, we have either $\mathbf{0}$ or $\mathbf{1}$, either $\mathsf{AX}_{\rightarrow}\mathbf{0}$ or $\mathsf{AX}_{\rightarrow}\mathbf{1}$, and either $\mathsf{AX}_{\leftarrow}\mathbf{0}$ or $\mathsf{AX}_{\leftarrow}\mathbf{1}$ in each abstract cell, so the result of rewrite is determined as follows:

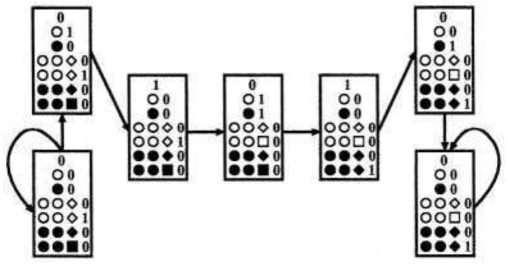


Then, we reevaluate each formula in F. Formulas containing only $\mathsf{AX}_{\rightarrow}$, $\mathsf{AG}_{\rightarrow}$, $\mathsf{AX}_{\leftarrow}$, $\mathsf{AG}_{\leftarrow}$ are true if it is true when we regard the above figure as a Kripke structure. For example, $\mathsf{AX}_{\rightarrow}\mathsf{AX}_{\rightarrow}\mathsf{AG}_{\rightarrow}\mathbf{0}$ holds at C_3.

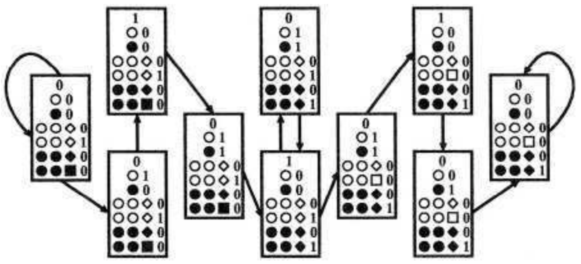
The truth/falsehood of a formula containing $\mathsf{EF}_{\rightarrow}$ is sometimes determined from reachability. Here we consider the formula $\mathsf{AX}_{\rightarrow}\mathsf{AX}_{\rightarrow}\mathsf{EF}_{\rightarrow}\mathbf{1}$ as an example. Note that its negation is equivalent to $\psi = \mathsf{AX}_{\rightarrow}\mathsf{AX}_{\rightarrow}\mathsf{AG}_{\rightarrow}\mathbf{0}$ as we mentioned before. Then, $p_{C_1} \wedge \psi$ is unsatisfiable because $p_{C_1} \supset \mathsf{AX}_{\rightarrow}\mathsf{AX}_{\rightarrow}(p_{C_1} \vee p_{C_2} \vee p_{C_3})$ holds, and $(p_{C_1} \vee p_{C_2} \vee p_{C_3}) \supset \mathsf{EF}_{\rightarrow}p_{C_4}$ is derivable using reachability formulas. That corresponds to the fact that it is not the case that the right neighbor of the concrete cell corresponding to C_1 is always abstracted to C_1.

As we mentioned, truth or falsehood of some properties in F may not be determined in reevaluation in some cases. For example, we can say neither $\mathsf{AX}_{\rightarrow}\mathbf{0}$ nor its negation $\mathsf{AX}_{\rightarrow}\mathbf{1}$ for C_1. So this cell is split into two cells, one having $\mathsf{AX}_{\rightarrow}\mathbf{0}$ and the other having $\mathsf{AX}_{\rightarrow}\mathbf{1}$.

As a result of reevaluation and reconstruction of abstract links, we obtain the following abstract graph:



If the result of reevaluation produces the multiple abstract cells having same set of formulas in F, they are merged. Repeatedly applying abstract transitions, we reaches to the following fixed point of the abstract graph:

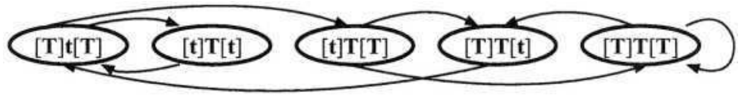


4.2 Asynchronous Transition: Dining Philosophers

For the example of dining philosophers that we mentioned in Sect. 2.3, we can show that no deadlock occur if there are $n(\geq 1)$ right-handed philosophers and exactly one left-handed philosopher independent of n.

We take a set F that characterize abstract cells as $S \cup \{\mathsf{EX}_{\rightarrow} p \mid p \in S\} \cup \{\mathsf{EX}_{\leftarrow} p \mid p \in S\}$. To avoid complicated notation, we use shorthand $[\alpha]p[\beta]$ to represent a set of abstract cells, where p is a state of the cell itself, α and β are a sequence that represents a set of states of left and right neighbor cell can take, respectively. For example, **[Tt]T[Tt]** represents four abstract cells. Reachability formulas are not used in this example.

With the above notation, the initial abstract cells where all philosophers are thinking are represented as **[Tt]T[Tt]**, **[T]t[T]**.



Starting with this initial abstract cells, the reachable ones are computed as follows:

$$[\mathbf{TUEtue}]\mathbf{T}[\mathbf{TUEtue}],\ [\mathbf{TUEtue}]\mathbf{U}[\mathbf{TUt}],\ [\mathbf{Ttu}]\mathbf{E}[\mathbf{TUt}],$$
$$[\mathbf{TUE}]\mathbf{t}[\mathbf{TUE}],\ [\mathbf{T}]\mathbf{u}[\mathbf{TUE}],\ [\mathbf{T}]\mathbf{e}[\mathbf{TU}]$$

Because there is exactly one left-handed philosopher, left and right neighbors of the left-handed philosopher in the concrete system must be represented by either of **[TUE]t[TUE],** **[T]u[TUE],** or **[T]e[TU].** For each case, we show that there exists at least one concrete cell that can be changed its state by the transition. For those including **e** or **E**, it can be changed to **t** or **T** by the transition, respectively. For **[T]u[TU],** there is a transition to **[T]e[TU].**

The remaining case is **[TU]t[TU].** This can be divided into **[T]t[TU],** **[U]t[T],** and **[U]t[U].** For the first one, there is a transition **[T]u[TU].** The last one can be changed to either **[U]t[E]** or **[E]t[E]** (the latter case occurs when there is only one right-handed philosopher).

Then the remaining case is now **[U]t[T].** Here, the state of the second right neighbor of the left-handed philosopher must not be **t,** **u,** or **e.** The latter two cases contradict with the fact that there is only one left-handed philosopher, and the first one also leads to contradiction because the left neighbor of the left-handed philosopher is now picking up the right fork. Therefore, the possible state of the second right neighbor is either **T,** **U,** or **E**. For **T** and **U,** there is a transition from **[U]t[T][TU]** to **[U]t[U][TU].** Finally, for **E,** there is a transition from **[U]t[T][E]** to **[U]t[T][T].**

5 Checking Satisfiability

For the completeness of the paper, we briefly describe an algorithm for checking satisfiability of 2CTL in this section. This is a straightforward extension of the standard tableau method for CTL [11].

For a formula ϕ_0, we define a set $cl(\phi_0)$ of formulas as the smallest set satisfying the following conditions:

- $\phi_0 \in cl(\phi_0)$.
- If $\phi_1 \wedge \phi_2 \in cl(\phi_0)$, then $\phi_1 \in cl(\phi_0)$ and $\phi_2 \in cl(\phi_0)$.
- If $\phi_1 \vee \phi_2 \in cl(\phi_0)$, then $\phi_1 \in cl(\phi_0)$ and $\phi_2 \in cl(\phi_0)$.
- If $M\phi \in cl(\phi_0)$, then $\phi \in cl(\phi_0)$.
- If $\phi \in cl(\phi_0)$ and $expand(\phi)$ is defined, then $expand(\phi) \in cl(\phi_0)$, where $expand(\phi)$ is defined as follows:
  $expand(\mathsf{AG}_A\phi) = \phi \wedge \mathsf{AX}_A\mathsf{AG}_A\phi$, $expand(\mathsf{EG}_A\phi) = \phi \wedge \mathsf{EX}_A\mathsf{EG}_A\phi$,
  $expand(\mathsf{AF}_A\phi) = \phi \vee \mathsf{AX}_A\mathsf{AF}_A\phi$, $expand(\mathsf{EF}_A\phi) = \phi \vee \mathsf{EX}_A\mathsf{EF}_A\phi$.

Suppose $\Gamma \subseteq cl(\phi_0)$. If $\phi_1 \vee \phi_2 \in \Gamma$ implies $f(\phi_1 \vee \phi_2) \in \{1, 2\}$, and $\mathsf{EX}_A\phi \in \Gamma$ implies $f(\mathsf{EX}_A\phi) \in A$, then we call f a *choice function* on Γ. When the following conditions are satisfied, a pair $\langle \Gamma, f \rangle$ of Γ and a choice function f on Γ is called a ϕ_0-type.

- $p \in \Gamma$ and $\neg p \in \Gamma$ do not hold simultaneously.
- If $\phi_1 \wedge \phi_2 \in \Gamma$, then $\phi_1 \in \Gamma$ and $\phi_2 \in \Gamma$.
- If $\phi_1 \vee \phi_2 \in \Gamma$, then $\phi_{f(\phi_1 \vee \phi_2)} \in \Gamma$.
- If $\phi \in \Gamma$ and $expand(\phi)$ is defined, then $expand(\phi) \in \Gamma$.

For each modality label a, a transition relation $\langle \Gamma, f \rangle \xrightarrow{a} \langle \Gamma', f' \rangle$ between ϕ_0-types is defined as follows:

- There exists $\mathsf{EX}_A\phi \in \Gamma$ such that $f(\mathsf{EX}_A\phi) = a$ and $\phi \in \Gamma'$.
- For all $\mathsf{AX}_A\psi \in \Gamma$, $\psi \in \Gamma'$ if $a \in A$.
- For all $\mathsf{AX}_A\psi \in \Gamma'$, $\psi \in \Gamma$ if $\overline{a} \in A$.
- There is no $\mathsf{AF}_A\psi$ satisfying the following conditions:
  $a \in A$ and $\overline{a} \in A$, $\mathsf{AF}_A\psi \in \Gamma$ and $\mathsf{AF}_A\psi \in \Gamma'$,
  $f(\psi \vee \mathsf{AX}_A\mathsf{AF}_A\psi) = 2$, $f'(\psi \vee \mathsf{AX}_A\mathsf{AF}_A\psi) = 2$.

The last condition above characterizes the fact that 2-way CTL has the inverse modality.

The procedure below is to determine satisfiability of the given formula ϕ_0 by repeatedly removing inconsistent ϕ_0-types.

When $\mathsf{EX}_A\phi \in \Gamma$ holds, if all ϕ_0-type $\langle\Gamma', f'\rangle$ satisfying $\langle\Gamma, f\rangle \xrightarrow{f(\mathsf{EX}_A\phi)} \langle\Gamma', f'\rangle$ have been removed, then remove $\langle\Gamma, f\rangle$.

Let $\psi_0 = \mathsf{AF}_A\psi$ or $\psi_0 = \mathsf{EF}_A\psi$. Suppose $\langle\Gamma_0, f_0\rangle$ is a ϕ_0-type, and $\psi_0 \in \Gamma_0$. We consider a labeled finite tree T_0 where the label of each node is a ϕ_0-type.

- The root node has $\langle\Gamma_0, f_0\rangle$ as its label.
- For any node and its child, there holds a transition relation $\xrightarrow{a}$ between labels of two nodes for some $a \in A$.
- For any node in T_0, its label $\langle\Gamma, f\rangle$ satisfies $\psi_0 \in \Gamma$.

In the case that $\psi_0 = \mathsf{AF}_A\psi$, the following conditions are added:

- For any internal node in T_0 and its label $\langle\Gamma, f\rangle$, for any $\mathsf{EX}_B\phi \in \Gamma$ satisfying $f(\mathsf{EX}_B\phi) \in A$, there exists its child whose label is $\langle\Gamma', f'\rangle$ such that $\langle\Gamma, f\rangle \xrightarrow{f(\mathsf{EX}_B\phi)} \langle\Gamma', f'\rangle$ and $\phi \in \Gamma'$ hold.
- For any internal node in T_0 and its label $\langle\Gamma, f\rangle$, $f(\psi \vee \mathsf{AX}_A\psi_0) = 2$ holds.
- For any external node in T_0 and its label $\langle\Gamma, f\rangle$, either $f(\psi \vee \mathsf{AX}_A\psi_0) = 1$ holds or there exists no $\mathsf{EX}_B\phi \in \Gamma$ such that $f(\mathsf{EX}_B\phi) \in A$.

In the above conditions, an internal node in T_0 is a node that has at least one child, and an external node is a node that is not an internal one.

In the case that $\psi_0 = \mathsf{EF}_A\psi$, the following conditions are added:

- T_0 is a path from its root to the unique external node and satisfies the following conditions.
- For any internal node on the path and its label $\langle\Gamma, f\rangle$, $f(\psi \vee \mathsf{EX}_A\psi_0) = 2$ holds, and for the next node on the path and its label $\langle\Gamma', f'\rangle$, $\langle\Gamma, f\rangle \xrightarrow{f(\mathsf{EX}_A\psi_0)} \langle\Gamma', f'\rangle$ and $\psi_0 \in \Gamma'$ hold.
- For any external node on the path and its label $\langle\Gamma, f\rangle$, $f(\psi \vee \mathsf{EX}_A\psi_0) = 1$ holds.

If there exists no tree T_0 satisfying the above conditions, we remove $\langle\Gamma_0, f_0\rangle$.

We repeatedly remove ϕ_0-types as much as possible. If ϕ_0-type $\langle\Gamma, f\rangle$ satisfying $\phi_0 \in \Gamma$ remains at the stage where we cannot remove anymore, then ϕ_0 is satisfiable.

6 Conclusion and Future Work

We described some results of our analysis method using abstraction by 2-way CTL for a linked structure in which labels of cells are changed by synchronous or asynchronous transitions. Many directions can be considered for future work:

- Implementation of the satisfiability checking algorithm.
- Extension to more expressive logics, such as CTL*, μ-calculus, and guarded first-order logic [12].
- Extension to the logic that reflects the properties on concrete cells.

– Abstraction of “hybrid cellular automata”, i.e., systems having not only discrete labels but also continuous (e.g., real-time) parameters in a state of a cell. For this purpose, we need extension to logics with time such as TCTL [13].

Acknowledgement. This work was partially supported by the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Exploratory Research, 14658088, 2002, and Grant-in-Aid for Scientific Research on Priority Areas, 15017212, 2003.

References

[1] Takahashi, K., Hagiya, M.: Abstraction of link structures by regular expressions and abstract model checking of concurrent garbage collection,. In: First Asian Workshop on Programming Languages and Systems, National University of Singapore (2000) 1–8 `http://nicosia.is.s.u-tokyo.ac.jp/members/hagiya.html`.
[2] Takahashi, K., Hagiya, M.: Formal proof of abstract model checking of concurrent garbage collection. In Kamareddine, F., ed.: Workshop on Thirty Five years of Automath, Informal Proceedings, Heriot-Watt University (2002) 115–126 `http://nicosia.is.s.u-tokyo.ac.jp/members/hagiya.html`.
[3] Takahashi, K., Hagiya, M.: Abstraction of graph transformation using temporal formulas. In: Workshop on Model-Checking for Dependable Software-Intensive Systems, International Conference on Dependable Systems and Networks, DSN2003. (2003) `http://nicosia.is.s.u-tokyo.ac.jp/members/hagiya.html`.
[4] Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Logic of Programs: Workshop, Yorktown Heights. Volume 131 of Lecture Notes in Computer Science., Springer Verlag (1981) 52–71
[5] Emerson, E.A., Clarke, E.M.: Characterizing correctness properties of parallel programs using fixpoints. In: Automata, Languages and Programming. Volume 85 of Lecture Notes in Computer Science., Springer Verlag (1980) 169–180
[6] S. Graf, H. Saidi: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: Proc. 9th INternational Conference on Computer Aided Verification (CAV’97). Volume 1254., Springer Verlag (1997) 72–83
[7] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Symposium on Principles of Programming Languages. (2002) 58–70
[8] Vardi, M.Y.: Reasoning about the past with two-way automata. In: Automata, Languages and Programming ICALP 98. Volume 1443 of Lecture Notes in Computer Science., Springer-Verlag (1998) 628–641
[9] von Neumann, J.: Theory of Self-Reproducing Automata. Urbana: University of Illinois Press (1966)
[10] Wolfram, S.: Cellular Automata and Complexity. Addison-Wesley (1994)
[11] Kozen, D., Tiuryn, J.: Logic of programs. In: Formal Models and Sematics. Volume B of Handbook of Theoretical Computer Science. Elsevier Science Publishers (1990) 789–840
[12] Andréka, H., Németi, I., van Benthem, J.: Modal languages and bounded fragments of predicate logic. Journal of Philosophical Logic **27** (1998) 217–274
[13] Alur, R., Courcoubetis, C., Dill, D.: Model checking for real-time systems. In: Proceedings of the Fifth Annual Symposium on Logic in Computer Science, IEEE Computer Society Press (1990) 414–425

Twelf and Delphin: Logic and Functional Programming in a Meta-logical Framework

Carsten Schürmann

Yale University, New Haven, USA

Logical framework research studies the design of meta-languages that support efficient encodings of object-languages and deductions prevalent in the areas of programming languages, mobile code and logics design. A meta-logical framework is a logical framework that, in addition, supports reasoning about these encodings. Over the previous decades various meta-logical frameworks have been developed and employed among them Nuprl, Isabelle, Coq, Lego, Maude, Elan and, of course, the meta-logical framework Twelf [PS99] that is being discussed here. [Pfe99] provides an excellent overview and a historical perspective of the field of logical frameworks. The fields of programming language design, safe and secure mobile code, verification of authorization and other security protocols and lately even experiment design in biology drive the development of the aforementioned frameworks and greatly benefit from their progress.

Twelf's design is based on the dependently typed Edinburgh logical framework LF [HHP93] and directly supports the judgment-as-types paradigm for encoding object languages and their semantics. Encodings may be higher-order and can therefore be used to capture binding constructs of languages as well as the meaning of hypothetical judgments without the standard restrictions usually associated with inductive datatypes (i.e. the positivity condition [PM93]). Reasoning about LF encodings takes place outside the logical framework, i.e. in a special purpose meta-logic designed for LF. In this respect, the design philosophy underlying Twelf is complementary to that of $\mathbf{FOL}^{\Delta\mathbb{N}}$ [MM97], a fixed meta-logic that can be parameterized by different logical frameworks. Besides the meta-theorem prover that searches for proofs within that meta-logic [Sch00], Twelf is equipped with a logic programming language called Elf that works directly with higher-order encodings.

Recently, my students and I have developed a functional programming language called Delphin [SP03b], that admits higher-order LF objects as first-class citizens and thus provides an alternative to the usual interpretation of datatypes. Delphin's most prominent features include function definition by cases; pattern matching against arbitrary LF objects (including those of functional type); general recursion; runtime extensible datatypes; a dependent type system and a novel world system that guarantees sound usage of dynamic constructors, termination and a coverage checker [SP03a] that checks for non-exhaustive sets of patterns. The projected goals of the Delphin project include access to logical framework technology for functional programmers, richer type systems that capture more refined invariants and support for shorter and more concise code.

© Springer-Verlag Berlin Heidelberg 2004

Delphin can be used to write type preserving compilers, proof-carrying architectures, proof emitting theorem provers and even proof transformations in between different logics.

Twelf and Delphin are closely related because they share the logical framework LF as a common data structure and they provide two different yet closely related operational semantics, one for logic and the other for functional programming. Driven by practical considerations, it has become important to be able to translate logic programs into functional programs in order to study their meta-theoretic properties. For example, in Twelf, logic programs play the role of proofs if and only if it can be shown that they are total, i.e. terminating and cover all cases. Coverage, however, is notoriously difficult to define and decide for Elf logic programs, but relatively straightforward for Delphin programs. Challenges that are addressed by this translation include a characterization of the interestingly large subset of logic programs as domain, the treatment of existential variables and the removal of deep backtracking.

In my talk, I will focus on Twelf and Delphin and the relation between the two.

References

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery,* 40(1):143–184, January 1993.

[MM97] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax: An extended abstract. In Glynn Winskel, editor, *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science,* pages 434–445, Warsaw, Poland, June 1997.

[Pfe99] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning.* Elsevier Science Publishers, 1999. In preparation.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications,* pages 328–345, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16),* pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[Sch00] Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems.* PhD thesis, Carnegie Mellon University, 2000. CMU-CS-00-146.

[SP03a] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In David Basin and Burkhart Wolff, editors, *Proceedings of Theorem Proving in Higher Order Logics (TPHOLs'03),* Rome, Italy, 2003. Springer Verlag LNCS 2758.

[SP03b] Carsten Schürmann and Adam Poswolsky. Delphin — towards functional programming with logical frameworks. Draft. Available from `http://www.cs.yale.edu/~carsten`, 2003.

Online Justification for Tabled Logic Programs*

Giridhar Pemmasani[1], Hai-Feng Guo[2], Yifei Dong[3], C.R. Ramakrishnan[1], and I.V. Ramakrishnan[1]

[1] Department of Computer Science, SUNY, Stony Brook, NY, 11794, USA,
`{giri,cram,ram}@cs.sunysb.edu`
[2] Department of Computer Science, University of Nebraska at Omaha
Omaha, NE, 68182, USA,
`haifengguo@mail.unomaha.edu`
[3] School of Computer Science, University of Oklahoma
Norman, OK 73019, USA,
`dong@ou.edu`

Abstract. Justification is the process of constructing evidence, in terms of proof, for the truth or falsity of an answer derived by tabled evaluation. The evidence is most easily constructed by post-processing the memo tables created during query evaluation. In this paper we introduce *online justification,* based on program transformation, to efficiently construct the evidence *during* query evaluation, while adding little overhead to the evaluation itself. Apart from its efficiency, online justification separates evidence generation from exploration thereby providing flexibility in exploring the evidence either declaratively or procedurally. We present experimental results obtained on examples that construct large evidences which demonstrate the scalability of online justification.

1 Introduction

Explaining or understanding the results of logic program evaluation, to aid in debugging or analyzing the program, has been a very active field of research. The complex machinery used to evaluate tabled logic programs makes the generation of explanations for query evaluation considerably difficult. Justification [17, 9] is the process of generating evidence, in terms of a high level proof, based on the answer tables created during query evaluation. Justification has two important advantages over techniques which generate evidence based on execution traces, viz. (i) it is independent of the evaluation machinery (e.g. the techniques scheduling goals in a tabled evaluation), and (ii) it enables direct construction of "meta-evidences": evidences for proof systems implemented as logic programs. Justification has played a fundamental role in generating proofs or counter examples for several problems in automatic verification (e.g., see [15, 16, 1]).

In earlier works [17, 9], we presented justification algorithms for logic programs by post-processing the memo tables created during query evaluation. Justification in this post-processing fashion is "non-intrusive" in the sense that it

* This research was supported in part by NSF grants CCR-9876242, IIS-0072927, CCR-0205376, CCR-0311512, and ONR grant N000140110967.

© Springer-Verlag Berlin Heidelberg 2004

is completely decoupled from query evaluation process and is done only after the evaluation is completed. However, post-processing introduces performance overheads which affect the scalability of the technique. In this paper we present an *online* technique for justification, which generates evidence, whenever possible, *during* query evaluation itself. Online justification is presented as a program transformation, and hence is still independent of the query evaluation machinery.

At a high level, our online justification technique transforms the given program in such a way that the query evaluation in the transformed program automatically constructs the evidence. For each literal in the program, we create two transformed literals: one to generate the evidence for its truth, and the other for its falsity. Evidence for derivability (truth) of an answer can be easily constructed by adding an extra argument in the predicate definitions that captures the evidence. However, in the presence of tabled evaluation, the extra evidence argument causes serious performance problems since the number of proofs for a goal may far exceed the number of steps it takes to compute the goal. For instance, consider the problem of a parser for an ambiguous grammar[21]: determining whether there is a parse tree can be done in time cubic on the length of the string whereas there may be exponentially many parse trees. We use a different transformation scheme for tabled predicates, storing the *first* evidence for an answer in a database, and thereby avoid this blow-up (see Section 3).

Generating evidences for false literals is more difficult, since evaluation of false literal simply fails without providing any information regarding the failure. We generate evidences for false literals by first constructing a *dual* program (based the notion of completed definition [11]) such that of a literal in the dual program is true if and only if its corresponding literal in the original program is false. Thus, evidences for false literals are constructed using the evidences for the corresponding (true) literals in the dual program (see Section 4).

Related Work: Extensive research has been done to help debug, analyze, explain or understand logic programs. The techniques that have been developed can be partitioned into three (sometimes overlapping) stages: instrumenting and executing the program, collecting data, and analyzing the collected data. Most of the techniques focus primarily on one of these three stages. For instance, the works on algorithmic debugging [18], declarative debugging [10, 13], and assertion based debugging [14] can be seen as primarily focussing on instrumenting the program; works on explanation techniques (e.g.,[12, 4, 19]) focus primarily on data collection; and works on visualization (e.g. [4, 2]) focus on the data analysis stage. Justification focusses on the data collection stage.

Unlike algorithmic debugging, justification only shows those parts of the computation which led to the success/failure of the query, and unlike declarative debugging, justification does not demand any creative input from the user regarding the intended model of the program, which can be very hard or even impossible to do as will be the case in model checking [3]. Among explanation techniques, [19] proposes a source-to-source transformation technique, which is very similar to our technique, to transform logic programs in the context of deductive databases. This technique generates evidence in bottom-up evaluation and is limited non-tabled programs, making it expensive especially in the presence of redundant computations. A later work [12] generates explanations by meta-interpreting intermediate information (traces) computed by the database engine, and overcomes the problems due to redundancy by caching the results. However, the explicit cycle checking done when generating explanations imposes a quadratic time overhead for evidence generation.

Trace-based debuggers (of which Prolog's 4-port debugger is a primitive example) provide only a procedural view of query evaluation. Procedural view provides information about the proof search, rather than the proof itself. The complex evaluation techniques used in tabling make this procedural view virtually unusable. Moreover, in the presence of multiple evaluation strategies and engines (SLG-WAM [20], Local vs. Batched scheduling [7], DRA [8] etc.), there is no uniform method to convert a trace of events during a proof search into a view of the evidence (proof, or its lack thereof) itself. Finally, by delinking evidence generation from evidence navigation, justification enables a user to selectively explore parts of the evidence and even re-inspect an explored part without restarting the debugging process.

Online Justification vs. Post-Processing: We originally presented a technique for justification of tabled logic programs [17] and later refined the technique to efficiently handle programs that mixed the evaluation of tabled with nontabled goals [9]. However, both the techniques post-processed the memo tables to build evidences. To understand the two main drawbacks of these techniques, consider the evaluation of query p over the tabled logic program given below:

```
:- table p/0.
p :- p.        p :- q.        q.
```

Post-processing-based justification is done by meta-interpreting the clauses of the program and the memo tables built during evaluation. For instance, the evidence for the truth of p is constructed by selecting a clause defining p such that the definition is true. Note that, in this case, the right hand sides of both clauses of p are true. For p to be true in the least model, its truth cannot be derived from itself. Hence the first clause is not an explanation for the truth of p. The meta-interpreter keeps a history of goals visited as it searches for an explanation and rejects any goal that will lead to a cyclic explanation. It will hence reject the first clause. Further, the justifier will produce the explanation that p is true due to q, which in turn is true since it is a fact.

First of all, note that justification appears to perform the same kind of search that the initial query evaluation did in the first place to determine the answers. Worse still, meta-interpretation is considerably slower than the original query evaluation. Secondly, determining whether a goal has been seen before is exactly what a tabling engine does well, and one which is tricky to do efficiently in a meta-interpreter. The justifiers we had built earlier keep the history of goals as a list (to make backtracking over the history inexpensive), and hence cycle-checking makes the justifier take time quadratic in the size of the evidence.

In contrast, online justification generates evidence by evaluating a transformed program directly, exploiting the tabling engine's (optimized) techniques for cycle detection, and eliminating the meta-interpretation overheads.

Justification in Practice: We present the justification technique initially for pure logic programs with a mixture of tabled and nontabled predicates and stratified negation. The technique can be readily extended to handle programs with builtin or nonlogical predicates (e.g. `var`) and aggregation (e.g. `findall`). Programs involving database updates (e.g. `assert`) and tabling are very uncommon; nevertheless our technique can be extended to such programs as well (see Section 5).

We have implemented an online justifier in the XSB logic programming system [22] and used it to build evidences in a complex application: the XMC model checking environment [15]. We use the XMC system as the primary example, since (i) evidences generated for XMC are large and have different structures based on the system and property being verified, thereby forming a platform to easily validate the scalability and understand the characteristics of the evidence generation technique; and (ii) counter-example generation was added to XMC using justification without modifying XMC itself, thereby demonstrating the flexibility offered by justification. Preliminary performance evaluation indicates that the online justifier for the XMC system adds very little overhead to the XMC model checker (see Section 6). When the model checker deems that a property is true, the overhead for justification to collect that proof is less than 8%. When a property is false, generating the evidence for the absence of a proof the overhead is at most 50%. In contrast, the post-processing based justifier originally implemented in the XMC system had an overhead of 4 to 10 *times* the original evaluation time [9], regardless of the result of the model checking run.

2 Evidence for Tabled Logic Programs

In this section, we give the intuition and formal definition of evidence in the context of tabled logic programs with left-to-right selection rule. By evidence, we mean the data in a proof, i.e., the subgoals and derivation relation between the subgoals. Proofs for (non-tabled) logic programs are traditionally represented by trees, or so-called "proof trees", where a goal is recursively decomposed to a set of subgoals until the subgoal indicates the presence or absence of a fact. In the case of tabled logic programs, however, a proof is not necessarily a tree because of the fixed-point semantics of tabling, i.e., a tabled predicate may fail due to circular reasoning. In [17, 9], proof trees are augmented by "ancestor" nodes or edges to form justifications for tabled programs, essentially indicating that the derivation relation of the subgoals is potentially a graph with loops.

Formally, we define an evidence (for a tabled logic program) as a graph whose vertices are literals and their truth values, and edges reflect the derivation relation between the subgoals represented by the literals. We use $succ(v)$ to denote the set of successors of a vertex v in a graph.

Definition 1 (Evidence). *An evidence for a literal L being true (false) with respect to a program $\mathcal{P}$, denoted by $\mathcal{E}_{\mathcal{P}}(L)$, is a directed graph $\langle V, E\rangle$ such that:*

1. *Each vertex $v \in V$ is uniquely labeled by a literal of $\mathcal{P}$ and a truth value, denoted by $(l(v), \tau(v))$;*
2. *There exists a vertex $v_0 \in V$ such that $l(v_0) = L$, $\tau(v_0) = true(false)$, and all other vertices in V are reachable from v_0;*
3. *For each vertex $v \in V$*
   (a) *if $\tau(v) = true$, $succ(v) = \{v'_1, \ldots, v'_n\}$ if and only if*
      i. $\exists\, C \equiv (\alpha :- \beta_1, \ldots, \beta_n) \in P$ *and a substitution* θ :
         $\alpha\theta = l(v) \wedge (\beta_1, \ldots, \beta_n)\theta = (l(v'_1), \ldots, l(v'_n))$.
      ii. $\forall 1 \leq i < n : \tau(v'_i) = true$
      iii. $(v, v) \notin E^+$
   (b) *if $\tau(v) = false$, $succ(v)$ is the smallest set such that*
      $\forall\, C \equiv (\alpha :- \beta_1, \ldots, \beta_n) \in P$:
      $\forall$ *substitution* $\theta : \alpha\theta = l(v)\theta \implies$
         $\exists\, 1 \leq k \leq n$ *and* $\{v'_1, \ldots, v'_k\} \subseteq succ(v)$:
            $(\beta_1, \ldots, \beta_k)\theta = (l(v'_1), \ldots, l(v'_k))\theta$
            $\wedge (\forall\, 1 \leq i < k : \tau(v'_i) = true)$
            $\wedge\, \tau(v'_k) = false$

Intuitively an evidence carries only the relevant information to establish a literal's truth or falsity. It is an AND-graph where each node is supported by all its successors. For a true literal, only one explanation matching a predicate definition is needed (Item 3.a.i); for a false literal, it must be shown that every possible combination of its predicate definition ultimately fails (Item 3.a.i). Furthermore only false literals can be in a loop, due to the least fixed-point semantics obeyed by tabled logic programs (Item 3.a.ii).

Definition 1 is logically equivalent to the definitions of justification in [17, 9] which define a spanning tree of evidence, where the backward edges are labeled by "ancestor" and leaves with truth value *true* and *false* are labeled by an edge to node "fact" and "fail" respectively. The benefit of the new definition is that same result will be generated from different traversal orders. Applying the result of [17, 9], we establish the usefulness of evidence by the following theorem.

Theorem 1 (Soundness and Completeness). *The query of a literal l succeeds (fails) if and only if there is an evidence for l being true (false).*

Hereafter when P is obvious from the context, we abbreviate $\mathcal{E}_P(L)$ to $\mathcal{E}(L)$. Sometimes we also abbreviate v's label $(l(v), \tau(v))$ to $l(v)$ or $\neg l(v)$ depending on whether $\tau(v)$ is *true* or *false*.

Example 1. Consider the following three programs:

```
P1: p.            P2: p :- q.       P3: :- table p/0.
                      q.                p :- q.
                                        q :- p.
```

The evidence for p being *true* in P_1 is just a single node labeled by p. The node has no successor because it is fact hence does not need further explanation. The evidence for p being *true* in P_2 contains two nodes labeled by p and q

respectively and an edge from p to q, meaning that p is *true* because q is *true*. Program P_3 encodes a circular reasoning, therefore p is *false*. The evidence of p being *false* is a loop from p to q then back to p. Formally,

$$\begin{aligned}\mathcal{E}_{P_1}(p) &= \langle\{(p, \mathit{true})\}, \emptyset\rangle \\ \mathcal{E}_{P_2}(p) &= \langle\{(p, \mathit{true}), (q, \mathit{true})\}, \{(p \rightarrow q)\}\rangle \\ \mathcal{E}_{P_3}(\neg p) &= \langle\{(p, \mathit{false}), (q, \mathit{false})\}, \{(\neg p \rightarrow \neg q), (\neg q \rightarrow \neg p)\}\rangle\end{aligned}$$

Example 2. In the following logic program, the predicate *reach*(*A*,*B*) defines the reachability from node *A* to node *B*, and *arc*(*From*, *To*) encodes the edge relation of a graph.

```
:- table reach/2.                               arc(a,b).
reach(A,B) :- arc(A,B).                         arc(b,a).
reach(A,B) :- arc(A,C), reach(C,B).             arc(c,a).
```

The evidences for $reach(a, a)$ being *true* and $reach(a, c)$ being *false* are depicted in Figure 1.

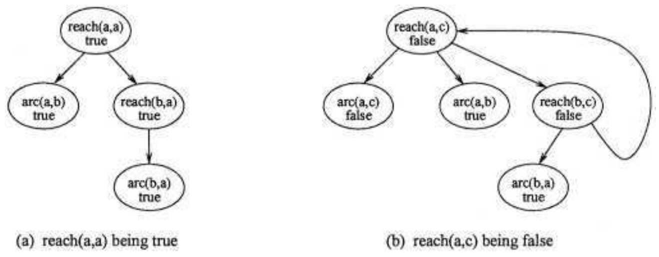


Fig.1. Two evidence examples in reachability program

3 Evidence Generation for True Literals

The key idea of online evidence generation is to generate the evidence for a literal while the literal is being evaluated. A simple way to implement this idea is to extend each clause

$$\alpha \text{ :- } \beta_1, \ldots, \beta_m.$$

to

$$\alpha' \text{ :- } \beta_1', \ldots, \beta_m', \texttt{store\_evid}(\alpha, [\beta_1, \ldots, \beta_m]). \tag{1}$$

where α' and β_i' are same as α and β_i, respectively, but indicate transformed predicates. When the query to a literal $L = \alpha\theta$ succeeds, the successors of L in the evidence, $[\beta_1\theta, \ldots, \beta_m\theta]$, are recorded by `store_evid`. Note that `store_evid` simply records a successful application of a clause and hence does not change the meaning of the program.

Unfortunately, this simple technique may generate more information than necessary for the purpose of evidence. Recall that an evidence carries only the relevant information to establish a literal's truth or falsity, therefore a backtracked call should not be part of the explanation for the final true answer. But the above transformation stores evidence for calls to α that are backtracked in a higher-level call.

Evidence as Explicit Arguments. We solve the above problem by passing the evidence as an argument of the predicates. We add an argument E to each atom α to form a new atom $\alpha'_{\mathtt{E}}$ which returns the evidence for α in E. Suppose $\alpha = p(t_1, \ldots, t_n)$, then $\alpha'_{\mathtt{E}} = p'(t_1, \ldots, t_n, \mathtt{E})$, where p' is a new name uniquely selected for p . The clause

$$p(t_1, \ldots, t_n) \text{ :- } \beta_1, \ldots, \beta_m.$$

is then transformed to

$$p'(t_1, \ldots, t_n, \mathtt{E}) \text{ :- } \beta'_{1\mathtt{E}_1}, \ldots, \beta'_{m\mathtt{E}_m}, \mathtt{E} = [(\beta_1, \mathtt{E}_1), \ldots, (\beta_m, \mathtt{E}_m)]. \qquad (2)$$

where $\mathtt{E}, \mathtt{E}_1, \ldots, \mathtt{E}_m$ are distinct new variables. The last statement in the clause combines the evidence for the subgoals to form the evidence for the top-level query, thus effectively building a tree. Similar to the first transformation, the new predicate $p'/(n+1)$ executes the same trace as the original predicate p/n. Since the evidence is now returned as an argument, backtracked predicates no longer generate unnecessary information.

Evidence for Tabled Predicates as Implicit Arguments. The situation becomes complicated when tabled predicates are present. Because all successful calls to tabled predicates are stored in tables, the space is not reclaimed when the predicates are backtracked. Furthermore, additional arguments for tabled predicates can increase the table space explosively [21]. Therefore there is no saving in passing evidence as an argument in tabled predicates.

To avoid the overhead, we do not pass evidence as arguments in tabled predicates. Instead, we use the store_evid method in Clause (1) to store a *segment* of evidence in database, where the evidence for a tabled subgoal β_i is recorded as $ref(\beta_i)$ pointing to β_i's record in the database, and the evidence for a tabled subgoal is the same as in Clause (2).

The complete transformation rule for true literals is shown in Figure 2.

Note that store_evid always succeeds, therefore adding it to the original predicate does not change the program's semantics. And by induction on the size of evidence, we have:

Proposition 1. *Let* E *be a fresh variable. For any literal $L \in P$*

- *the query $L'_{\mathtt{E}}$ succeeds if and only if L succeeds*
- *if the query $L'_{\mathtt{E}}$ succeeds, then* E *returns an evidence for L being true.*

For each clause $p(t_1, \ldots, t_n)$:- $\beta_1, \ldots, \beta_m$.

– If p/n is non-tabled, transform the clause to

$$\begin{aligned} p'(t_1, \ldots, t_n, \mathtt{E}) \ \text{:-}\ & \beta'_{1\mathtt{E}_1}, \ldots, \beta'_{m\mathtt{E}_m}, \\ & \mathtt{E} = [(\beta_1, \mathtt{E}_1), \ldots, (\beta_m, \mathtt{E}_m)]. \end{aligned}$$

– If p/n is tabled, transform the clause to

$$\begin{aligned} p'(t_1, \ldots, t_n) \ \text{:-}\ & \beta'_{1\mathtt{E}_1}, \ldots, \beta'_{m\mathtt{E}_m}, \\ & \mathtt{store\_evid}(p(t_1, \ldots, t_n), [(\beta_1, \mathtt{E}_1), \ldots, (\beta_m, \mathtt{E}_m)]). \end{aligned}$$

where $\mathtt{E}, \mathtt{E}_1, \ldots, \mathtt{E}_m$ are distinct new variables, and
for each $\beta = q(u_1, \ldots, u_l)$,

$$\beta'_{\mathtt{E}} = \begin{cases} q'(u_1, \ldots, u_l, \mathtt{E}) & \text{if } \beta \text{ is a non-tabled predicate} \\ (q'(u_1, \ldots, u_l), \mathtt{E} = \mathit{ref}(\beta)) & \text{if } \beta \text{ is a tabled predicate} \end{cases}$$

Fig. 2. Transformation Rules for True Literals

Example 3. The program in Example 2 is transformed as follows.

```
:- table reach_t/2.
reach_t(A,B) :- arc_t(A,B,E),
                store_evid(reach(A,B), [((arc(A,B),true),E)]).
reach_t(A,B) :- arc_t(A,C,E1), reach_t(C,B),
                store_evid(reach(A,B),
                           [((arc(A,C),true),E1),
                            ((reach(C,B),true),ref(reach(C,B)))]).
arc_t(a,b,[]).          arc_t(b,a,[]).          arc_t(c,a,[]).
```

The query `reach_t(a,a)` will succeed with the evidence stored in two records:

$$\mathit{reach}(a,a) \rightarrow [((\mathit{arc}(a,b), \mathit{true}), []), ((\mathit{reach}(b,a), \mathit{true}), \mathit{ref}(\mathit{reach}(b,a)))]$$
$$\mathit{reach}(b,a) \rightarrow [((\mathit{arc}(b,a), \mathit{true}), [])]$$

4 Evidence Generation for False Literals

Evidence generation for false literals is more difficult than that of true literals, because when the extended predicates fail, they do not return any tracing information. We solve this problem by justifying the *negation* of false literals. We present the solution in two steps. In the first step, for each literal L, we compute a dual literal $\overline{L}$ which is equivalent to $\neg L$. In the second step, we apply the transformation rule for the true literals to $\overline{L}$.

Dual Predicates. Let p/n be a predicate, where p is the predicate name and n is the arity of p. We say that the predicate $\overline{p}/n$ is the *dual predicate* of p if $\overline{p}$ and

p return complementary answers for the same input, *i.e.* for any literal instance $p(t_1,\ldots,t_n)$, where $t_1,\ldots,t_n$ are terms, $\overline{p}(t_1,\ldots,t_n) = \neg p(t_1,\ldots,t_n)$.

Recall from the definition of evidence (Definition 1) that a literal L is false if for every clause $\alpha : -\beta_1,\ldots,\beta_n$ such that L unifies with α, under any substitution θ, there is some $j \leq n$ such that $\beta_l\theta$ is false and for all $1 \leq i < j$, $\beta_i\theta$ is true. This directly leads to the following definition of dual predicates. For the sake of simplicity, let p be defined using k clauses in the form of $p(t_i)$:- $\beta_{i,1}, \beta_{i,2}$. Then,

$$\overline{p}(x) \text{ :- } \overline{p_1}(x), ..., \overline{p_k}(x)$$

where $\overline{p_i}$ captures the failure of the i-th clause of p. Now each i-th clause fails if either the arguments of p do not match with the arguments of p_i, or $\beta_{i,1}$ fails, or for every answer of $\beta_{i,1}$, $\beta_{i,2}$ fails. This is captured by the following rule:

$$\begin{aligned}\overline{p_i}(x) &\text{ :- } x \neq t_i \\ \overline{p_i}(t_i) &\text{ :- } \overline{\beta_{i,1}} \text{ ; } \mathtt{forall}(\beta_{i,1}, \overline{\beta_{i,2}})\end{aligned}$$

where the predicate $\mathtt{forall}(\beta_1, \beta_2)$ encodes $\forall\theta : \beta_1\theta \implies \beta_2\theta$.

Dual Definitions for Tabled Predicates. For a tabled predicate involving recursive definitions, however, the dual predicates defined in the above scheme is incorrect in general. We can view the above definition of the dual as being obtained from the completion of a logic program [11]. It is well known that completion has unsatisfactory semantics for negation. This impacts our justification algorithm. Consider the simple prepositional program `p :- p`, where $p/0$ is tabled. The dual predicate produced by the above transformation rule is `p_f :- p_f`. Because there is a loop in the definition of `p_f`, if `p_f` is not tabled, the query does not terminate; if it is tabled, it will give the answer *false*. However, since `p` fails, `p_f` should succeed.

To break the loops in dual predicates, we use the explicit negation of the tabled predicates instead of their duals in the definitions. In XSB, that a tabled predicate α has no answer, *i.e.* $\neg\exists\theta : \alpha\theta$, is encoded by $\mathtt{sk\_not}(\alpha)$. In other tabled LP systems such as TALS and B-Prolog, the operator for tabled negation is the same as for non-tabled cases. Here we use `table_not` to represent this negation operator for all tabled systems. For the above example, we replace `p_f` in the body of of dual for `p` by `table_not(p)`. Now the dual predicate becomes `p_f :- table_not(p)`, so the query `p_f` correctly succeeds.

The benefit of using `table_not` for tabled predicate in the dual definitions is that there are no recursive calls to the duals of tabled predicates, hence the duals need not be tabled. This not only avoids cycle checking but also enables us to implement the justifier as a zero-storage producer, as the evidence does not consume permanent space when being passed as an argument.

Generating Evidence for the Dual Predicates. We apply the transformation rule for true literals presented in Section 3 to the dual predicates, so that for each predicate α, we generate an extended dual predicate $\overline{\alpha}'_{\mathtt{E}}$ that returns the evidence for $\neg\alpha$ in the variable `E`.

The two predicates introduced during dualization, `forall` and `table_not`, need special treatment in the transformation.

- We implement a predicate $\texttt{all\_evid}((\beta'_{1\texttt{E}_1}, \texttt{E}_1), (\beta'_{2\texttt{E}_2}, \texttt{E}_2), \texttt{E})$ to computes the evidence of $\texttt{forall}(\beta_1, \beta_2)$:

$$\mathcal{E}(\texttt{forall}(\beta_1, \beta_2)) = \bigcup_{\forall\theta:\beta_1\theta} \{(\beta_1\theta, \mathcal{E}(\beta_1\theta)), (\beta_2\theta, \mathcal{E}(\beta_2\theta))\}$$

  where $\texttt{E}_1$, $\texttt{E}_2$, and $\texttt{E}$ hold the evidences of $\beta_1\theta$, $\beta_2\theta$, and $\texttt{forall}(\beta_1, \beta_2)$ respectively.
- To extend the predicate $\texttt{table\_not}(\beta)$, we apply the same technique used for true tabled predicates, *i.e.* to store only a pointer to the evidence for $\neg\beta$, denoted by $\mathit{ref}(\beta)$. Similar to the evidences for true literals, the evidences for false tabled literals are stored in segments. The evidence segment for $\neg\beta$ can be generated by calling $\overline{\beta}'_{\texttt{E}}$, which can be done at any time, giving us an algorithm of generating partial evidence *on demand.* The evidence is fully generated when the evidence segments to all referred literals are computed.

The complete transformation rule for false literals is in Figure 3. A special case of this transformation is that when a predicate p/n has no definition (thus p trivially fails), a fact $\overline{p}'(\texttt{X}_1, \ldots, \texttt{X}_n, [])$. is generated according to Step 1, meaning $\overline{p}'$ trivially succeeds.

Based on the definition of dual predicates and Proposition 1, we can establish the consistency of the transformed predicate. Also by induction on the size of evidence, we have:

Proposition 2. *Let* E *be a fresh variable. For any literal* $L \in P$,

- *the query* $\overline{L}'_{\texttt{E}}$ *succeeds if and only if* L *fails*
- *if the query* $\overline{L}'_{\texttt{E}}$ *succeeds then* E *returns an evidence for* L *being false.*

Example 4. The program in Example 2 is transformed as follows.

```
reach_f(A,B,E)  :- reach_f1(A,B,E1),reach_f2(A,B,E2),concat([E1,E2],E).
reach_f1(A,B,E)  :- arc_f(A,B,E).
reach_f2(A,B,E)  :- arc_f(A,C,E).
reach_f2(A,B,E)  :- all_evid((arc_t(A,C,E1),E1),
                          (reach_f(C,B),E2=ref(reach(C,B))), E).
arc_f(A,B,E)    :- arc_f1(A,B,E1), arc_f2(A,B,E2), arc_f3(A,B,E3),
                   concat([E1,E2,E3],E).
arc_f1(A,B,[])  :- (A,B) \= (a,b).
arc_f2(A,B,[])  :- (A,B) \= (b,a).
arc_f3(A,B,[])  :- (A,B) \= (c,a).
```

The query `reach_f(a,c,E)` will succeed, returning one evidence segment:

$$\begin{aligned} \mathit{reach}(a,c) \rightarrow\ & [((\mathit{arc}(a,c), \mathit{false}), []), ((\mathit{arc}(a,b), \mathit{true}), []), \\ & ((\mathit{reach}(b,c), \mathit{false}), \mathit{ref}(\mathit{reach}(b,c)))] \end{aligned}$$

For each predicate p/n whose definition is composed of k clauses in the form of

$$p(t_{i,1},\ldots,t_{i,n}) \texttt{ :- } \beta_{i,1},\ldots,\beta_{i,m_i}.$$

where $1 \leq i \leq k$, the extended dual predicate $\overline{p}'$ is defined in two parts:

1. The top-level predicate is

$$\begin{aligned}\overline{p}'(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{E}) \texttt{ :- } & \overline{p}'_1(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{E}_1),\ldots,\overline{p}'_k(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{E}_k),\\ & \texttt{concat}([\mathtt{E}_1,\ldots,\mathtt{E}_k],\mathtt{E}).\end{aligned}$$

   where $\texttt{concat}([\mathtt{E}_1,\ldots,\mathtt{E}_k],\mathtt{E})$ is a predicate that concatenates $\mathtt{E}_1,\ldots,\mathtt{E}_k$ together to a single list $\mathtt{E}$.
2. For each $1 \leq i \leq k$, the predicate $\overline{p}'_i/(n+1)$ is defined by two clauses:

$$\begin{aligned}\overline{p}'_i(\mathtt{X}_1,\ldots,\mathtt{X}_n,[\,]) &\texttt{ :- } \texttt{not}((\mathtt{X}_1,\ldots,\mathtt{X}_n) = (t_{i1},\ldots,t_{in})).\\ \overline{p}'_i(t_{i,1},\ldots,t_{i,n},\mathtt{E}) &\texttt{ :- } \mathit{fevid}([\beta_{i,1},\ldots,\beta_{i,m_i}],\mathtt{E}).\end{aligned}$$

   where *fevid* is a macro recursively defined as

$$\begin{aligned}\mathit{fevid}([\,],\mathtt{E}) &\overset{def}{=} \texttt{fail}\\ \mathit{fevid}([\beta_1|B],\mathtt{E}) &\overset{def}{=} (\overline{\beta_1}'_{\mathtt{E}_1^f} \texttt{ -> } \mathtt{E} = [((\beta_1, false),\mathtt{E}_1^f)]\\ &\qquad\qquad ;\ \texttt{all\_evid}((\beta'_{1\mathtt{E}_1^t},\mathtt{E}_1^t),(\mathit{fevid}(B,\mathtt{E}_2^f),\mathtt{E}_2^f),\mathtt{E}).\end{aligned}$$

   and $\mathtt{E}$,$\mathtt{E}_1^f$,$\mathtt{E}_1^t$, and $\mathtt{E}_2^f$ are distinct new variables. For each atom $\beta = q(u_1,\ldots,u_l)$ appearing in the body of a clause, its extended dual expression is defined as

$$\overline{\beta}'_{\mathtt{E}} = \begin{cases} (\texttt{table\_not}(q'(u_1,\ldots,u_l)),\mathtt{E} = \mathit{ref}(\beta)) & \text{(if } \beta \text{ is a tabled predicate)}\\ \overline{q}'(u_1,\ldots,u_l,\mathtt{E}) & \text{(otherwise)}\end{cases}$$

Fig. 3. Transformation Rules for False Literals

To produce the evidence segment for $reach(b,c)$, we call `reach_f(b,c,E)`, which returns

$$reach(b,c) \rightarrow [((arc(b,a),true),[\,]),((reach(a,c),false),ref(reach(a,c)))]$$

Now since all referred literals have been visited, the evidence is fully generated.

5 Practical Aspects

In Sections 3 and 4, we described general transformation rules for pure logic programs. In practice, however, most programs have non-logical constructs (such as `assert`, `retract`) and meta-operators (such as `var`, `nonvar`). To make online justification practical, below we describe transformation techniques necessary to handle such programs. In addition, we show how online justification can be used as a flexible evidence explorer.

Meta-operators. Since we transform predicate names into different names using p' and $\overline{p}'$, literal call(P) in the original program has to be transformed so that it calls the appropriate transformed literal during execution. If P is a non-variable, then it is transformed using the general transformation rules; otherwise, it is transformed into $call'(P)$ or $\overline{call}'(P)$, depending on whether call is being transformed as true literal or false literal, respectively. $call'(P)$ and $\overline{call}'(P)$ are part of run-time support for the justifier, which transforms P during execution and call the resulting predicate.

XMC model checker uses predicate forall(Vars, Antecedent, Consequent) which is defined as

```
forall(_Variables, Antecedent, Consequent) :-
   findall(Consequent, Antecedent, AllConsequents),
   all_true(AllConsequents).
all_true([]).
all_true([C|Cs]) :- call(C), all_true(Cs).
```

If forall is transformed using the general transformation rules, then all_true generates quadratic amount of evidence. The transformed predicates for forall should first collect evidence for antecedent from findall, evidence for consequents from all_true and then assemble both to generate the evidence for forall. There is no general technique to handle this behavior; hence we implemented these transformed predicates by hand.

Controlling Evidence. As can be seen in Section 6, the transformed program has very little time overhead, but in some cases, the space overheads may be high; e.g., justification of non-tabled predicates such as append take exponential amount of space to store the evidence due to recursive definitions. To avoid generating evidence for such predicates, we provide a mechanism to specify the predicates the user intends to justify and transform only those predicate definitions. Limiting the amount of evidence not only results in reducing the overheads, but also helps the user explore the evidence easily.

Constructs with Side-effects. Pure logic predicates have no side-effects, hence can be executed many times, without changing the semantics of the program. However, assert and retract cannot be executed more than once, as they change the semantics of the program. For true literals, the evidence is generated during the evaluation of the program, which can be retrieved without executing the literal any more. The dual definitions for the literals, on the other hand, should not call assert and retract, as the original program wouldn't have executed them. To avoid changing the database during execution of false literals, the predicates with assert/retract can be transformed in such a way that they first make a copy of the current database into another database, during the execution change only the copy and at the end delete the copied database. Thus, executing the dual definition doesn't change the database.

Flexible Evidence Generation and Exploration. The segments of evidence generated during query evaluation in online justification can then be used to generate the full evidence graph. So online justification can be used as a tool to debug programs. However, it has a few fundamental differences with the traditional trace-based debuggers. Justification gives a declarative view of the logic program, displaying sufficient and necessary information to establish the truth or falsity of a query, whereas debuggers provide only the procedural view of the execution. Online justification gives flexibility in both generating and exploring the evidence generated during evaluation by allowing the user to explore the evidence as and when necessary, skipping over uninteresting portions and even revisiting the skipped portions later without restarting the debugging process.

6 Experimental Results

One of the primary goals of our work is to implement a *practical* tool for justification of tabled logic programs. To measure the overheads of time and space on such programs, we have used the justifier to automatically transform the XMC model checker [15] to construct the evidence for model checking. The entire implementation of the model checker consists of about 150 lines of XSB, most of which are definitions for the non-tabled predicate models(S,F), which checks if state S models the μ-calculus formula F, and one definition for a tabled predicate rec_models(S,F), which checks if state S models formula definition of F. A fragment of this model checker is given below:

```
:- table rec_models/2.
rec_models(State_s, X) :- fDef(X, Y), models(State_s, Y).
models(_State_s, tt).
models(State_s, fAnd(X_1, X_2))  :-
   models(State_s, X_1), models(State_s, X_2).
models(State_s, fOr(X_1, X_2))   :-
   models(State_s, X_1) ;  models(State_s, X_2).
models(State_s, fDiam(Act_a, X)) :-
   transition(State_s, Act_a, State_t), models(State_t, X).
```

Since models(S,F) is a non-tabled predicate, the justifier transforms it into two predicates: models_t(S,F,E), corresponding to the true-literal justification and models_f (S, F, E), corresponding to the false-literal justification, where E is the evidence. The rec_models is transformed so that the evidence of the definition and the evidence from models/2 are stored in the evidence database. The transformed model checker is then used on some examples from XMC test suite [6] with various system sizes: Iprotocol, Leader election, Java meta-lock and Sieve. The system sizes that we have tried are very large (requiring upto 1GB of system memory). Here we report the time and space performance of this model checker along with the original XMC model checker.

All the tests were performed on Intel Xeon 1.7GHz machine with 2GB RAM running RedHat Linux 7.2 and XSB version 2.5 (optimal mode, slg-wam with local scheduling) with the garbage collector turned off.

Figure 6(a) compares the query evaluation time of the original XMC (without justification) with the transformed XMC (with justification). In our experiments, the transformed program took at most 50% (and on average 23%) longer time compared to the original program. Note that all the graphs are drawn in logarithmic scale, with performance of XMC without justification on X-axis and the performance of transformed XMC with justification on Y-axis.

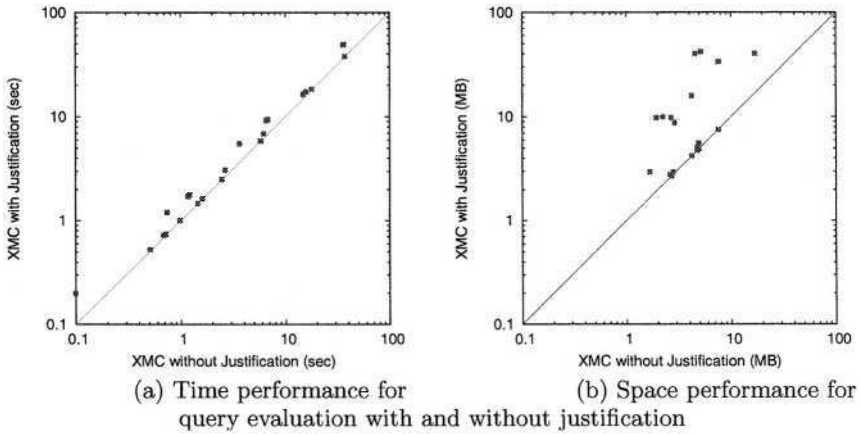


Fig. 4. Time and Space Performance of XMC with and without Justifier

Figure 6(b) shows the space overhead due to evidence generation. In our experiments, the transformed program has maximum overhead of 11 times when evidence is generated for every state along every path in the model (due to `forall`) in the case of Leader election protocol, and on average about 3.5 times the overhead. The comprehensive details about time and space performance of the online justifier tool can be found at `http://www.lmc.cs.sunysb.edu/~lmc/justifier`.

7 Conclusions

In this paper, we presented a new justification scheme using program transformation. In this scheme, a logic program is automatically translated such that the translated program builds evidence during query evaluation. The evidence so generated can be presented later to the user using an interactive interface. The extra overhead due to the evidence generation is so little that the tool we implemented using this scheme has been used in practice to generate evidence for model checking practical systems. We plan to extend our scheme to handle logic programs with side effects, and to integrate our implementation with the Evidence Explorer [5], so that the user can easily navigate the evidence.

References

[1] Samik Basu, Diptikalyan Saha, Yow-Jian Lin, and Scott A. Smolka. Generation of all counter-examples for push-down systems. In *FORTE,* 2003.

[2] M. Cameron, M. García de la Banda, K. Marriott, and P. Moulder. Vimer: a visual debugger for mercury. In *Proceedings of the 5th ACM SIGPLAN,* 2003.

[3] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of countterexamples and witnesses in symbolic model checking. In *32*nd *Design Automation Conference,* 1995.

[4] P. Deransart, M. Hermenegildo, and J. Maluszynski. *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging,* volume 1870 of *LNCS.* Springer, 2000.

[5] Y. Dong, C. R. Ramakrishnan, and S. A. Smolka. Evidence Explorer: A tool for exploring model-checking proofs. In *Proc. of 15*th *CAV,* LNCS, 2003.

[6] Yifei Dong and C. R. Ramakrishnan. Logic programming optimizations for faster model checking. In *TAPD,* Vigo, Spain, September 2000.

[7] J. Freire, T. Swift, and D.S. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. In *PLILP,* 1996.

[8] Hai-Feng Guo and Gopal Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *ICLP,* volume 2237 of *LNCS,* 2001.

[9] Hai-Feng Guo, C. R. Ramakrishnan, and I. V. Ramakrishnan. Speculative beats conservative justification. In *ICLP,* volume 2237 of *LNCS,* 2001.

[10] J.W. Lloyd. Declarative error diagnosis. In *New Generation Computing,* 1987.

[11] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[12] S. Mallet and M. Ducasse. Generating deductive database explanations. In *ICLP,* 1999.

[13] L. Naish, P.W. Dart, and J. Zobel. The NU-Prolog debugging environment. In *ICLP,* 1999.

[14] G. Puebla, F. Bueno, and M. Hermenegildo. A framework for assertion-based debugging in constraint logic programming. In *LOPSTR,* volume 1817 of *LNCS,* 1999.

[15] C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Yifei Dong, Xiaoqun Du, Abhik Roychoudhury, and V. N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *12*th *CAV,* volume 1855 of *LNCS.* Springer, 2000.

[16] P. Roop. *Forced Simulation: A Formal Approach to Component Based Development of Embedded Systems.* PhD thesis, Computer Science and Engineering, University of New South Wales, 2000.

[17] Abhik Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *Principles and Practice of Declarative Programming.* ACM Press, 2000.

[18] E. Shapiro. Algorithmic program diagnosis. In *Proceedings of POPL'82,* 1982.

[19] G. Specht. Generating explanation trees even for negations in deductive database systems. In *ILPS'93 Workshop on Logic Programming Environments,* 1993.

[20] T. Swift and D. S. Warren. An abstract machine for SLG resolution: definite programs. In *Proceedings of the Symposium on Logic Programming,* 1994.

[21] David S. Warren. *Programming in Tabled Prolog.* 1999. Early draft available at `http://www.cs.sunysb.edu/~warren/xsbbook/book.html.`

[22] XSB. The XSB logic programming system. Available at `http://xsb.sourceforge.net.`

Constructive Intensional Negation*

Susana Muñoz-Hernández, Julio Mariño, and Juan José Moreno-Navarro

Universidad Politécnica de Madrid
Dpto. LSIIS – Facultad de Informática
Campus de Montegancedo s/n, 28660, Madrid, SPAIN.
{susana,jmarino,jjmoreno}@fi.upm.es
voice: +34-91-336-7455, fax: +34-91-336-6595

Abstract. Although negation is an active area of research in Logic Programming, sound and complete implementations are still absent from actual Prolog systems. One of the most promising techniques in the literature is *Intensional Negation (IN),* which follows a transformational approach: for each predicate p in a program its negative counterpart $intneg(p)$ is generated. However, implementations of IN have not been included in Prolog environments due, in part, to the lack of details and explicit techniques, such as the treatment of universally quantified goals. In this paper, we describe a variant of IN, which we have called *Constructive Intensional Negation (CIN).* Unlike earlier proposals, CIN does not resort to a special resolution strategy when dealing with universally quantified formulae, which has been instrumental in having an effective implementation. Among the contributions of this work we can mention a full implementation being tested for its integration in the Ciao Prolog system and some formal results with their associated proofs.

Keywords: Negation, Constraint Logic Programming, Program Transformation, Logic Programming Implementation, Constructive Negation.

1 Introduction

Kowalski and Colmerauer's decision on the elements of first-order logic supported in Logic Programming was influenced by the availability of implementation techniques and efficiency considerations. Among those important aspects not included from the beginning we can mention *evaluable functions, negation* and *higher order features.* All of them have revealed themselves important for the expressiveness of Prolog as a programming language, but, while in the case of evaluable functions and higher order features considerable effort has been invested both in the semantic characterization and its efficient implementation we cannot say the same of negation. Many research papers do propose semantics to understand and incorporate negation into logic programming, but only a small subset of these ideas have their corresponding implementation counterpart.

In fact, systems such as XSB[4], DLV[1] or SMODELS[3] do implement the well-founded or the stable model semantics, but represent a departure from standard Prolog. On the other hand, the negation features incorporated in actual Prolog compilers are rather limited, namely: the (unsound) negation as failure rule, and the sound (but incomplete) delay technique of the language Gödel[14], or Nu-Prolog[20] (having the

* This research was partly supported by the Spanish MCYT project TIC2000-1632.

© Springer-Verlag Berlin Heidelberg 2004

risk of floundering.) The constructive negation of ECLiPSe[2], which was announced in earlier versions has been removed from recent releases due to implementation errors.

The authors have been involved in a project [17, 18, 19] to incorporate negation in a real Prolog system (up to now Ciao [8], but the techniques are easily applicable to any other system). This allows us to take advantage of state-of-the-art, WAM-based Prolog systems, as well as to reuse thousands of Prolog lines of code. The basis of our work is to combine and refine existing techniques to make them useful for practical application. Furthermore, we try to use the most efficient technique as possible in any particular case[19]. To help on this distinction, we need to use some tests to characterize the situation for efficiency reasons. To avoid the execution of dynamic tests, we suggest to use the results of a global analysis of the source code. For instance, the primary technique is the built-in *negation as failure* that can be used if a groundness analysis tells that every negative literal will be ground at call time [18].

In order to handle non-ground literals, a number of alternatives to the negation-as-failure rule have been proposed under the generic name of *constructive negation:* Chan's *constructive negation*[9, 10, 22, 13], *intensional negation*[5, 6, 7], fail substitutions, fail answers, *negation as instantiation*[21], etc. From a theoretical viewpoint Chan's approach is enough but it is quite difficult to implement and expensive in terms of execution resources. On the other hand, IN uses a transformational approach, so most of the work is performed at compile time and then (a variant of) the standard SLD resolution is used to execute the resulting program, so a significant gain in efficiency is expected. There are, however, some problems when transforming certain classes of programs.

In this paper we focus on the study of an efficient implementation of IN. As formulated in the original papers, it is difficult to derive an efficient transformation. On one hand, universally quantified goals generated are difficult to manage. On the other hand the operational behavior of the original program is modified computing infinitely many answers instead of compact results, unless disequality constraints are built into the Prolog system and this, and other details, have been missing so far.

The following paragraphs describe existing work on intensional negation.

1.1 Intensional Negation

In *Intensional Negation* [5, 6] a program transformation technique is used to add new predicates to the program in order to express the negative information. Informally, the *complement* of head terms of the positive clauses are computed and they are used later as the head of the negated predicate. We will denote the negated predicate of p as *intneg_p*. The following example is taken from [5]:

```
even(0) ←                        intneg_even(s(0)) ←
even(s(s(X))) ← even (X)         intneg_even(s(s(X))) ← intneg_even(X)
```

The predicate *intneg_even* succeeds when *even* fails and *vice versa.*

Our transformation approach basically follows the ideas of [6], but differs on some significant points. For the detailed description of the transformation, Barbuti and his co-authors apply the transformation to a restricted class of programs. Then they show that any program can be translated into a program of that class. Despite the elegance and simplicity of the presentation, this description is not very helpful for practical implementation.

There are two problems with this technique. The first one is that in the presence of logical variables on the right-hand side (rhs) of a clause, the new program needs to handle some kind of universal quantification construct. The second point affects the outcomes of the program: while the new program is semantically equivalent to the completed program, the operational behavior can differ. In the presence of logical variables, the new predicate can generate all the possible values one by one, even when a more general answer can be given in an expanded language (with disequality constraints). The predicate *p/2* defined by the single clause *p*(*X*,*X*) is negated by:

```
intneg(p)(X, Y) ← intneg(eq)(X, Y)
intneg(eq)(0, s(Y))
intneg(eq)(s(X), 0)
intneg(eq)(s(X), s(Y)) ← intneg(eq)(X, Y)
```

if the program only works with natural numbers constructed with 0 and succ/1. The query *intneg*(*p*)(*X, Y*)) will generate infinitely many answers, instead of the more general constraint $X \neq Y$. An answer like $X \neq Y$ can only be replaced by an infinite number of equalities.

1.2 Compilative Constructive Negation

To cope with the second problem it is possible to use explicitly the formulas in the program completion as rewrite rules with a suitable constraint extraction procedure. This forces us to interpret the transformed program with in a Constraint LP framework, working with equality and disequality constraints over the Herbrand domain. *Compilative Constructive Negation* of [7] follows the ideas of constructive negation but introduces constraints for the left-hand side of the predicate (and, therefore, for its negation). The description of the program transformation is in the following definition.

Definition 1 (Negation Predicate). *Given a predicate p defined by m clauses:*

$$C_1 : p(\bar{t}_1) \leftarrow A_1, B_1 \ \ldots \ C_m : p(\bar{t}_m) \leftarrow A_m, B_m$$

where each $B_j = G'_{j,1}, \ldots, G'_{j,r_j}$ *is a collection of literals and* $A_j = G_{j,1}, \ldots, G_{j,k_j}$ *is a collection of literals with no free variables (i.e.* $Vars(A_j) \in Vars(\bar{t}_j)$*), its* Negated Predicate *is defined as*

$$\neg p(\overline{X}) \iff (\forall \overline{Y}_1.\neg c_1 \vee \neg A_1 \vee \neg B_1) \wedge \cdots \wedge (\forall \overline{Y}_m.\neg c_m \vee \neg A_m \vee \neg B_m)$$

where $\overline{Y}_j = free\_var\ (C_j)$ *and* c_j *is the constraint* $\overline{X} = \bar{t}_j$.

We can rename the conjunction as $\neg p(\overline{X}) \iff F'_1 \wedge \cdots \wedge F'_m$. As for each j we have

$$F'_j = \forall \overline{Y}_j.(\neg c_j \vee \neg A_j \vee \neg B_j) \equiv \forall \overline{Y}_j.(\neg c_j) \vee \forall \overline{Y}_j.(\neg c_j \vee \neg B_j) \vee \neg A_j$$
$$\equiv \forall \overline{Y}_j.(\neg c_j) \vee \forall \overline{Y}_j.(\neg c_j \vee \neg G'_{j,1} \vee \cdots \vee \neg G'_{j,r_j}) \vee \neg G_{j,1} \vee \cdots \vee \neg G_{j,k_j}$$

then we compute the disjunctive normal form $(F_1 \vee \cdots \vee F_h)$ equivalent to the formula $(F'_1 \wedge \cdots \wedge F'_m)$ and replace in F_i the occurrences of $\neg q$ by *compneg_q*, obtaining F^*_i. The new program *CompNeg*(*P*) contains, for each *p*, the clauses:

$$compneg\_p(\overline{X}) \leftarrow F^*_1 \;\; \ldots \;\; compneg\_p(\overline{X}) \leftarrow F^*_h$$

Universally quantified subgoals resulting from the negation of clauses with free variables are evaluated by means of a new operational semantics called **SLD$^\forall$-resolution.** The authors prove the correctness and completeness of the new semantics w.r.t. the 3-valued program interpretation, that in our context means that the result is equivalent to the em Intensional Negation concept of definition 2. However, the introduction of a new operational mechanism contradicts the compilative nature of the method, makes difficult an efficient implementation and breaks our goal of the combination of negated goals with other Prolog modules already developed and tested. Furthermore, the compilation scheme is very naive: more than one clause can be created for the same goal, a lot of useless universally quantified goals are generated and the programs contains a lot of trivial constraints.

The rest of the paper is organized as follows. Section 2 introduces basic syntactic and semantic concepts needed to understand our method. Section 3 formally presents the transformation algorithm of the Constructive Intensional Negation technique. In section 4 we discuss our universal quantification definition and we provide implementation details in section 5. Finally, we conclude and discuss some future work (section 6). Proofs of relevant theorems and lemmata can be found in the appendix[1].

2 Preliminaries

In this section the syntax of (constraint) logic programs and the intended notion of correctness are introduced. Programs will be constructed from a signature $\Sigma = \langle FS_\Sigma, PS_\Sigma \rangle$ of function and predicate symbols. Provided a numerable set of variables V the set $Term(FS_\Sigma, V)$ of terms is constructed in the usual way.

A *constraint* is a first-order formula whose atoms are taken from $Term(FS_\Sigma, V)$ and where the only predicate symbol is the binary equality operator $= /2$. A formula $\neg(t_1 = t_2)$ will be abbreviated $t_1 \neq t_2$. The constants $\underline{t}$ and $\underline{f}$ will denote the neutral elements of conjunction and disjunction, respectively. A tuple $(x_1, \ldots, x_n)$ will be abbreviated by $\overline{x}$. The concatenation of tuples $\overline{x}$ and $\overline{y}$ is denoted $\overline{x} \cdot \overline{y}$. The existential closure of first-order formula φ is denoted $\exists\varphi$.

A (constrained) Horn clause is a formula $h(\overline{x}) \leftarrow b_1(\overline{y} \cdot \overline{z}), \ldots, b_n(\overline{y} \cdot \overline{z}) [] c(\overline{x} \cdot \overline{y})$ where $\overline{x}, \overline{y}$ and $\overline{z}$ are tuples from disjoint sets of variables[2]. The variables in $\overline{z}$ are called the *free* variables in the clause. The symbols "," and "[]" act here as aliases of logical conjunction. The atom to the left of the symbol "$\leftarrow$" is called the *head* or *left hand side* (lhs) of the clause. *Generalized* Horn clauses of the form $h(\overline{x}) \leftarrow B(\overline{y} \cdot \overline{z}) [] c(\overline{x} \cdot \overline{y})$ where

[1] Available at ftp://lml.ls.fi.upm.es/pub/users/susana/papers/intneg_appendix.ps. Extended examples are also included.

[2] The notation $p(\overline{x})$ expresses that $Vars(p(\overline{x})) \in \overline{x}$, not that it is identical to $p(x_1, \ldots, x_n)$

the body B can have arbitrary disjunctions, denoted by ";", and conjunctions of atoms will be allowed as they can be easily translated into "traditional" ones.

A Prolog program (in Σ) is a set of clauses indexed by $p \in PS_\Sigma$:

$$p(\overline{x}) \leftarrow B_1(\overline{y}_1 \cdot \overline{z}_1) \square c_1(\overline{x} \cdot \overline{y}_1) \quad \cdots \quad p(\overline{x}) \leftarrow B_m(\overline{y}_m \cdot \overline{z}_m) \square c_m(\overline{x} \cdot \overline{y}_m)$$

The set of defining clauses for predicate symbol p in program P is denoted $def_P(p)$. Without loss of generality we have assumed that the left hand sides in $def_P(p)$ are syntactically identical. Actual Prolog programs, however, will be written using a more traditional syntax.

Assuming the normal form, let $def_P(p) = \{p(\overline{x}) \leftarrow B_i(\overline{y}_i \cdot \overline{z}_i) \square c_i(\overline{x} \cdot \overline{y}_i) | i \in 1 \ldots m\}$. The *completed definition* of p, $cdef_P(p)$ is defined as the formula

$$\forall \overline{x}. \left[p(\overline{x}) \iff \bigvee_{i=1}^{m} \exists \overline{y}_i .\ c_i(\overline{x} \cdot \overline{y}_i) \wedge \exists \overline{z}_i . B_i(\overline{y}_i \cdot \overline{z}_i) \right]$$

The *Clark's completion* of the program is the conjunction of the completed definitions for all the predicate symbols in the program along with the formulas that establish the standard interpretation for the equality symbol, the so called *Clark's Equality Theory* or *CET*. The completion of program P will be denoted *Comp*(P). Throughout the paper, the standard meaning of logic programs will be given by the three-valued interpretation of their completion – i.e. its minimum 3-valued model, as defined in [15]. These 3 values will be denoted $\underline{t}$ (or `success`), $\underline{f}$ (or `fail`) and $\underline{u}$ (or `unknown`).

Example 1. Let us start with a sample program (over the natural numbers domain) to compute whether a number is even. It can be expressed as a set of normalized Horn clauses in the following way:

```
even(X) ← [] X=0.
even(X) ← even(N) [] X=s(s(N)).
```

Its completion is $CET \wedge \forall x. [even(x) \iff x = 0 \vee \exists n. x = s(s(n)) \wedge even(n)]$.

We are now able to specify IN formally. Given a signature $\Sigma = \langle FS_\Sigma, PS_\Sigma \rangle$, let $PS'_\Sigma \supset PS_\Sigma$ be such that for every $p \in PS_\Sigma$ there exists a symbol $neg(p) \in PS'_\Sigma \setminus PS_\Sigma$. Let $\Sigma' = \langle FS_\Sigma, PS'_\Sigma \rangle$.

Definition 2 (Intensional Negation of a Program). *Given a program P_Σ, its* intensional negation *is a program $P'_{\Sigma'}$ such that for every p in PS_Σ the following holds:*

$$\forall \overline{x}. [Comp(P) \models_3 p(\overline{x}) \iff Comp(P') \models_3 \neg(neg(p)(\overline{x}))]$$

3 Our Transformation

Our transformation differs from the above methods [5, 6, 7], as we have tried to keep the transformed program as close (syntactically) as possible to the original one (i.e. minimize the number of extra clauses, trivial constraints, universally quantified goals, etc.) in order to optimize its behaviour. In this sense, the translation scheme is not trivial and some simplifications can be obtained by taking into account some properties that are often satisfied by the source program. For instance, the clauses for a given predicate are usually mutually exclusive. The following definition formalizes this idea:

Definition 3. *A pair of constraints* c_1 *and* c_2 *is said to be* incompatible *iff their conjunction* $c_1 \wedge c_2$ *is unsatisfiable. A set of constraints* $\{c_i\}$ *is* exhaustive *iff* $\bigvee_i c_i = \underline{t}$. *A predicate definition* $\mathbf{def}_P(p) = \{p(\overline{x}) \leftarrow B_i(\overline{y}_i \cdot \overline{z}_i) \Box c_i(\overline{x} \cdot \overline{y}_i) | i \in 1 \dots m\}$ *is* nonoverlapping *iff* $\forall i, j \in 1 \dots m$ *the constraints* $\exists \overline{y}_i . c_i(\overline{x} \cdot \overline{y}_i)$ *and* $\exists \overline{y}_j . c_j(\overline{x} \cdot \overline{y}_j)$ *are incompatible (so,* $i \neq j$*) and* $\mathbf{def}_P(p)$ *is exhaustive if the set of constraints* $\{\exists \overline{y}_i . c_i(\overline{x} \cdot \overline{y}_i)\}$ *is exhaustive.*

In the following, the symbols "$\twoheadrightarrow$" and "$\|$" will be used as syntactic sugar for "$\wedge$" and "$\vee$" respectively, when writing completed definitions from nonoverlapping and exhaustive sets of clauses, to highlight their behaviour as a *case-like* construct. A nonoverlapping set of clauses can be made into a nonoverlapping and exhaustive set by adding an extra "default" case:

Lemma 1. *Let p be such that its definition* $\mathbf{def}_P(p) = \{p(\overline{x}) \leftarrow B_i(\overline{y}_i \cdot \overline{z}_i) \Box c_i(\overline{x} \cdot \overline{y}_i) | i \in 1 \dots m\}$ *is nonoverlapping. Then its completed definition is logically equivalent to*

$$\begin{aligned} \mathbf{cdef}_P(p) \equiv \forall \overline{x}.[p(\overline{x}) \iff & \exists \overline{y}_1 .\ c_1(\overline{x} \cdot \overline{y}_1) \twoheadrightarrow \exists \overline{z}_1 . B_1(\overline{y}_1 \cdot \overline{z}_1) \ \| \\ & \vdots \\ & \exists \overline{y}_m .\ c_m(\overline{x} \cdot \overline{y}_m) \twoheadrightarrow \exists \overline{z}_m . B_m(\overline{y}_m \cdot \overline{z}_m) \ \| \\ & \textstyle\bigwedge_{i=1}^m \neg \exists \overline{y}_i .\ c_i(\overline{x} \cdot \overline{y}_i) \twoheadrightarrow \underline{f}\,] \end{aligned}$$

The interesting fact about this kind of definitions is captured by the following lemma:

Lemma 2. *Given* $\{C_1, \dots, C_n\}$ *a set of exhaustive and nonoverlapping Herbrand constraints and* $\{B_1, \dots, B_n\}$ *a set of first-order formulas with no occurrences of* $\overline{x}$ *and* $\overline{y}_i$ *a set of variables included in the free variables of* C_i, $i \in \{1..n\}$ *the following holds:*

$$\forall \overline{x}(\neg[\exists \overline{y}_1(C_1 \twoheadrightarrow B_1) \| \cdots \| \exists \overline{y}_n(C_n \twoheadrightarrow B_n)] \iff \exists \overline{y}_1(C_1 \twoheadrightarrow \neg B_1) \| \cdots \| \exists \overline{y}_n(C_n \twoheadrightarrow \neg B_n))$$

This means that the overall structure of the formula is preserved after negation, which seems rather promising for our purposes.

The idea of the transformation to be defined below is to obtain a program whose completion corresponds to the negation of the original program, in particular to a representation of the completion where negative literals have been eliminated. We call this transformation *Negate*.

Definition 4 (Constructive Intensional Negation of a predicate). *Let p be a predicate of a program P and let* $\mathbf{def}_P(p) = \forall \overline{x}. [p(\overline{x}) \iff D]$ *be its completed definition. The* constructive intensional negation *of p, Negate(p), is the formula obtained after the successive application of the following transformation steps:*

1. *For every completed definition of a predicate p in* PS_Σ, $\forall \overline{x}. [p(\overline{x}) \iff D]$, *add the completed definition of its negated predicate,* $\forall \overline{x}. [\mathbf{neg\_p}(\overline{x}) \iff \neg D]$.
2. *Move negations to the right of "*$\twoheadrightarrow$*" using lemma 2.*
3. *If negation is applied to an existential quantification, replace* $\neg \exists \overline{z}.C$ *by* $\forall \overline{z}.\neg C$.
4. *Replace* $\neg C$ *by its* negated normal form[3] *NNF*($\neg$C).
5. *Replace* $\neg \underline{t}$ *by* $\underline{f}$, $\neg \underline{f}$ *by* $\underline{t}$, *for every predicate symbol p,* $\neg p(\overline{t})$ *by* $\mathbf{neg\_p}(\overline{t})$.

[3] The *negated normal form* of *C*, is obtained by successive application of the De Morgan laws until negations only affect atomic subformulas of *C*.

Definition 5 (Constructive Intensional Negation of a program). *Let P be a program, the* constructive intensional negation *of P will be the transformation of P into another program consisting of the constructive intensional negations of all predicates of P.*

Lemma 3. *Let the formula Negate(P) be the constructive intensional negation of a program P. Then, for every predicate symbol* $p \in PS_\Sigma$ *of P,*
$Negate(P) \models_3 (\forall \overline{x}.neg\_p(\overline{x}) \Longleftrightarrow \neg p(\overline{x}))$.

Definition 6. *The syntactic transformation negate_rhs is defined as follows:*

$$\begin{aligned}
negate\_rhs(P;Q) &= negate\_rhs(P), negate\_rhs(Q)\\
negate\_rhs(P,Q) &= negate\_rhs(P); negate\_rhs(Q)\\
negate\_rhs(\underline{t}) &= \underline{f}\\
negate\_rhs(\underline{f}) &= \underline{t}\\
negate\_rhs(p(\overline{t})) &= neg\_p(\overline{t})
\end{aligned}$$

Definition 7 (Constructive Intensional Negation). *For every predicate p in the original program, assuming* $def_P(p) = \{p(\overline{x}) \leftarrow B_i(\overline{y}_i \cdot \overline{z}_i) [] c_i(\overline{x} \cdot \overline{y}_i) | i \in 1 \ldots m\}$ *a nonoverlapping and exhaustive definition, the following clauses will be added to the negated program:*

– *If the set of constraints* $\{\exists \overline{y}_i.c_i(\overline{x}.\overline{y}_i)\}$ *is not exhaustive, a clause*

$$neg\_p(\overline{x}) \leftarrow [] \bigwedge_1^m \neg \exists \overline{y}_i.c_i(\overline{x}.\overline{y}_i)$$

– *If* $\overline{z}_j$ *is empty, the clause* $neg\_p(\overline{x}) \leftarrow negate\_rhs(B_j(\overline{y}_j)) [] c_j(\overline{x}.\overline{y}_j)$

– *If* $\overline{z}_j$ *is not empty, the clauses* $neg\_p(\overline{x}) \leftarrow forall(\overline{z}_j, p\_j(\overline{y}_j \cdot \overline{z}_j)) [] c_j(\overline{x}.\overline{y}_j)$
$p\_j(\overline{y}_j \cdot \overline{z}_j) \leftarrow negate\_rhs(B_j(\overline{y}_j \cdot \overline{z}_j))$

We can see that negating a clause with free variables introduces "universally quantified" goals by means of a new predicate *forall*/2 that is discussed in section 4 where soundness and completeness results are given. Implementation issues are given in section 4. In the absence of free variables the transformation is trivially correct, as the completion of the negated predicate corresponds to the negation-free normal form described above.

Theorem 1. *The result of applying the* Constructive Intensional Negation *transformation to a (nonoverlapping) program P is an intensional negation (as defined in def. 2).*

3.1 Overlapping Definitions

When the definition of some predicate has overlapping clauses, the simplicity of the above transformation is lost. Rather than defining a different scheme for overlapping rules, we will give a translation of general sets of clauses (that can be applied to any predicate definition) into nonoverlapping ones:

Lemma 4. *Let p be such that* $def_P(p) = \{p(\overline{x}) \leftarrow B_i(\overline{y}_i \cdot \overline{z}_i)[]c_i(\overline{x} \cdot \overline{y}_i) | i \in 1 \ldots m\}$ *and there exist* $j, k \in 1 \ldots m$ *such that* $\exists \overline{y}_j . c_j(\overline{x} \cdot \overline{y}_j)$ *and* $\exists \overline{y}_k . c_k(\overline{x} \cdot \overline{y}_k)$ *are compatible. Then the j-th and k-th clauses can be replaced by the clause*

$$p(\overline{x}) \leftarrow B_j(\overline{y}_j \cdot \overline{z}_j); B_k(\overline{y}_k \cdot \overline{z}_k)[]c_j(\overline{x} \cdot \overline{y}_j) \wedge c_k(\overline{x} \cdot \overline{y}_k)$$

and the following additional clauses (if the new constraints are not equivalent to $\underline{f}$*):*

$$p(\overline{x}) \leftarrow B_j(\overline{y}_j \cdot \overline{z}_j)[]c_j(\overline{x} \cdot \overline{y}_j) \wedge \neg \exists \overline{y}_k . c_k(\overline{x} \cdot \overline{y}_k)$$

$$p(\overline{x}) \leftarrow B_k(\overline{y}_k \cdot \overline{z}_k)[]\neg \exists \overline{y}_j . c_j(\overline{x} \cdot \overline{y}_j) \wedge c_k(\overline{x} \cdot \overline{y}_k)$$

without changing the standard meaning of the program. The process can be repeated if there are more than two overlapping clauses. It is clear that it is a finite process.

3.2 Examples

Let us show some motivating examples:

Example 2. The predicate `lesser/2` can be defined by the following nonoverlapping set of clauses

```
lesser(0,s(Y))
lesser(s(X),s(Y)) ← lesser(X,Y)
```

or, normalizing,

```
lesser(N,M) ← [] N=0, M=s(Y)
lesser(N,M) ← lesser(X,Y) [] N=s(X), M=s(Y)
```

By lemma 1, the completed definition of `lesser` is

$$\begin{aligned} cdef_P(lesser) \equiv \forall n, m.\ lesser(n, m) \iff & \exists y. n = 0 \wedge m = s(y) \twoheadrightarrow \underline{t} \ \| \\ & \exists x, y. n = s(x) \wedge m = s(y) \twoheadrightarrow lesser(x, y) \ \| \\ & m = 0 \twoheadrightarrow \underline{f} \end{aligned}$$

We have assumed that the constraint $\neg(\exists y.n = 0 \wedge m = s(y)) \wedge \neg(\exists x, y.n = s(x) \wedge m = s(y))$ has been simplified to $m = 0$ (if the only symbols in the program are 0 and $s/1$).

And the generated Prolog program for `neg_lesser` is:

```
neg_lesser(N,M) ← neg_lesser(X,Y) [] N=s(X), M=s(Y)
neg_lesser(N,M) ← [] M=0
```

Example 3. The second example, *family,* is also well-known, and includes free variables on the rhs of the definition of the predicate `ancestor/2`:

```
    parent(john, mary)      ancestor(X, Y) ← parent(X, Y)
    parent(john, peter)     ancestor(X, Y) ← parent(Z, Y) ∧
    parent(mary, joe)                        ancestor(X, Z)
    parent(peter, susan)
```

As `ancestor/2` is defined by a pair of overlapping clauses, we get:

```
ancestor(N,M) ← parent(X,Y) ; (parent(Z,Y), ancestor(Y,X)) [] N=X, M=Y
```

The corresponding Prolog program for `neg_ancestor` is:

```
neg_ancestor(N,M) ← neg_parent(X,Y),
                    forall([Z],neg_parent(Z,Y); neg_ancestor(Y,X)) [] N=X, M=Y
```

4 Universal Quantification

The main difference between our method and proposals like [10, 7] lies in a treatment of universal quantification that works by exploring the Herbrand Universe, making use of the following:

1. A universal quantification of the goal Q over a variable X succeeds when Q succeeds without binding (or constraining) X.
2. A universal quantification of Q over X is true if Q is true for all possible values for the variable X.

This results in a mutually recursive rule that can be expressed formally as follows:

$$\forall X.\ Q(X) \equiv Q(sk) \vee \left[\forall \overline{X_1}.\ Q(c_1(\overline{X_1})) \wedge \cdots \wedge \forall \overline{X_n}.\ Q(c_n(\overline{X_n}))\right]$$

where $FS_\Sigma = \{c_1 \ldots c_n\}$ is the set of function symbols and sk is a Skolem constant, that is, $sk \notin FS_\Sigma$. In practice, the algorithm proceeds by trying the Skolem case first and, if this fails, then it expands the variables in all possible constructors. The following definitions try to capture some tricky details of the procedure, which are necessary for ensuring its completeness. For reasons of simplicity, we will only consider quantifications of the form $\forall x.Q$ where Q has neither quantifiers nor free variables in the following (apart from x), but the results generalize to the case of quantification over several variables.

Definition 8 (Covering). *A* covering *of $Term(FS_\Sigma, V)$ is any finite sequence of terms $\langle t_1 \ldots t_n\rangle$ such that:*

- *For every i, j with $i \neq j$, t_i and t_j do not unify, i.e. there is no ground substitution σ with $t_i\sigma = t_j\sigma$.*
- *For every ground term s of the Herbrand Universe there exists i and a ground substitution σ with $s = t_i\sigma$.*

The simplest covering is a variable $C_1 = \langle X\rangle$. If a program uses only natural numbers ($FS_\Sigma = \{0, s/1\}$), for example, $C_2 = \langle 0, s(X)\rangle$, $C_3 = \langle 0, s(0), s(s(X))\rangle$ and $C_4 = \langle 0, s(0), s(s(0)), s(s(s(X)))\rangle$, etc., are also coverings. This example also suggests how to generate coverings incrementally. We start from the simplest covering X. From one covering we generate the next one, choosing one term and one variable of this term. The term is removed and then we add all the terms obtained replacing the variable by all the possible instances of that element.

Definition 9 (Next Covering). *Given a covering $C = \langle t_1, \ldots, t_m\rangle$, the* next covering *of C over the variable X is defined as $next(C, X) = \langle t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_m, t_{j1}, \ldots, t_{jn}\rangle$, where each $t_{jk} = t_j\sigma_k$ is obtained from the symbols in FS_Σ by applying, for each $c_k \in FS_\Sigma$, the substitution $\sigma_k = [X \mapsto c_k(\overline{X_k})]$, where $\overline{X_k} \cap Vars(t_j) = \emptyset$. We can say that a covering C is less instantiated that next(C,X) in the sense of [21].*

Definition 10 (Sequence of Coverings). $S = \langle C_1, \ldots, C_n\rangle$ *is a* sequence of coverings *if $C_1 = \langle X\rangle$, for some variable X and for every $i \in \{1 \ldots n-1\}$, $C_{i+1} = next(C_i, X_i)$, where X_i is the variable appearing in C_i at the leftmost position.*

Lemma 5. *Given an infinite sequence of coverings $S = \langle C_1, C_2, \ldots \rangle$ and a ground term $t \in Term_\Sigma$, then there exists a finite $n \in \mathbb{N}$ such that $t \in C_n$.*

The proof of this lemma is trivial for the definition of next covering.

To use coverings as successive approximations of a universal quantification, it is necessary to replace their variables by Skolem constants:

Definition 11 (Evaluation of a Covering). *Given a covering C, we define skolem(C) as the result of replacing every variable by a different Skolem constant. The* evaluation *of C for the quantified goal $\forall X.Q$ is the evaluation (meaning) of the three-valued conjunction of the results of evaluating Q with all the elements in skolem(C). That is*

$$eval(C, \forall X.Q) = I(Q[X \mapsto t_1] \wedge \cdots \wedge Q[X \mapsto t_m]) \in \{\underline{t}, \underline{f}, \underline{u}\}$$

where $skolem(C) = \langle t_1 \ldots t_m \rangle$.

After the evaluation of a covering C_i for a given quantified goal, several situations may arise:

1. $eval(C_i, \forall X.Q) = \underline{t}$. C_i is true for $\forall X.Q$, so we can infer that the universal quantification of Q succeeds.
2. $eval(C_i, \forall X.Q) = \underline{u}$. C_i is undefined for $\forall X.Q$, so we can infer that the universal quantification of Q loops (it is undefined).
3. $eval(C_i, \forall X.Q) = \underline{f}$. According to the three-valued semantics of conjunction this will happen whenever there exists some t_j in $skolem(C_i)$ such that $Q[X \mapsto t_j] \rightsquigarrow \underline{f}$. We can consider two subcases:
   – There is some t_j in $skolem(C_i)$ which does not contain Skolem constants such that $Q[X \mapsto t_j] \rightsquigarrow \underline{f}$. C_i is false, so we can infer that the universal quantification of Q fails.
   – Otherwise, for every t_j in $skolem(C_i)$ such that $Q[X \mapsto t_j] \rightsquigarrow \underline{f}$, t_j contains Skolem constants. We cannot infer that the universal quantification of Q fails, and $eval(C_{i+1}, \forall X.Q)$ must be considered. We will say that C_i is *undetermined* for $\forall X.Q$. Next covering should be considered: $eval(C_i, \forall X.Q) = eval(C_{i+1}, \forall X.Q)$

Definition 12 (forall). *The evaluation of a universally quantified goal Q over a set of variables Vars, forall(Vars, Q), is the process of evaluating the first covering C_1 for $\forall Vars.\ Q$, that is:*

$$forall(Vars, Q) = eval(C_1, \forall Vars.\ Q)$$

With this method, we construct a diagonalization to cover the domain of a set of variables, that is inspired by *Cantor diagonalization.* Cantor diagonalization is used to prove that the cartesian product of numerable factors ($\mathbb{N}^m$) is denumerable, e. g. it ensures that all elements are visited in a finite number of steps. So, it assures that the elements will be evaluated fairly during the generation covering and an infinite branch in the covering terms evaluation is impossible. It can be also applied in many decidability and computational results. This and other diagonalization methods are studied and used in [11].

It is important to point out some characteristics of Kunen's interpretation that are the key for getting completeness results for our approach.

Something can succeed or fail at infinity (I_ω) in Fitting semantics, but if there is no finite set in which we can check the failure or success of a goal in Kunen semantics, the goal is undefined. If we consider the simple predicate *nat*/1:

$$\begin{array}{l} \mathbf{nat}(0) \leftarrow \\ \mathbf{nat}(s(X)) \leftarrow \mathbf{nat}(X) \end{array}$$

we find that while $\forall X.\ \mathbf{nat}(X)$ succeeds in Fitting semantics (at I_ω), it is undefined in Kunen semantics, because it cannot be evaluated in a finite number of steps. That is, there is no $n \in \mathbb{N}$ such that $I_n \models \forall X.\mathbf{nat}(X)$. This is the Prolog interpretation. The reader is referred to Kunen's paper [15] for a wider discussion on the subject.

Similarly, $\forall X.\ \mathbf{nat}(X)$ will fail in Kunen semantics iff $\exists X.\ \neg\mathbf{nat}(X)$, that is, if there exists a counterexample that we are able to find in a finite process.

During this process, nontermination may arise either during the evaluation of one covering or because the growth of the sequence of coverings does not end. The following lemmata show that this does not affect completeness of the method:

Lemma 6. *Let $S = \langle C_1, C_2, \ldots \rangle$ be a sequence of coverings. $\forall X.Q$ is false iff there exists a finite $n \in \mathbb{N}$ such that $\mathbf{eval}(C_n, \forall X.Q) = \underline{f}$.*

Lemma 7. *Let $S = \langle C_1, C_2, \ldots \rangle$ be a sequence of coverings. If there is any C_i in S such that $\mathbf{eval}(C_i, \forall X.Q) = \underline{u}$, then $\mathbf{eval}(C_{i+1}, \forall X.Q) = \underline{u}$.*

Lemma 8. *Let $S = \langle C_1, C_2, \ldots \rangle$ be an infinite sequence of coverings such that for every $i > 0$, C_i is undetermined for $\forall X.Q$. Then it can be proved that, under the three-valued semantics of the program the quantification, $\forall X.Q$ is undefined.*

That is, loops introduced during the computation of coverings are consistent with the semantics of universal quantification. The case considered in the last lemma corresponds, essentially, to universal quantifications that hold at infinite iterations of Fitting's transformer but not at any finite instance.

Lemma 9. *Let $S = \langle C_1, C_2, \ldots \rangle$ be a sequence of coverings. If $\forall X.Q$ succeeds than there exists a finite $n \in \mathbb{N}$ such that $\mathbf{eval}(C_n, \forall X.Q) = \underline{t}$*

Theorem 2 (Completeness of forall). *When $\forall \overline{X}.Q$ succeeds/fails in Kunen semantics, then $\mathbf{forall}(\overline{X}, Q)$ succeeds/fails.*

Theorem 3 (Correctness of forall). *When $\mathbf{forall}(\overline{X}, Q)$ succeeds/fails in Kunen semantics, then $\forall \overline{X}.Q$ succeeds/fails.*

5 Implementation Issues

Disequality Constraints An instramental step in order to manage negation in a more advanced way is to be able to handle disequalities between terms such as $t_1 \neq t_2$. Prolog implementations typically include only the built-in predicate \== /2 can only work with disequalities if both terms are ground and simply succeeds in the presence of free variables. A "constructive" behavior must allow the "binding" of a variable with a disequality. On the other hand, the negation of an equation $X = t(\overline{Y})$ produces the universal quantification of the free variables in the equation, unless a more external quantification affects them. The negation of such an equation is $\forall\, \overline{Y}.\, X \neq t(\overline{Y})$. As we explained in [17], the inclusion of disequalities and constrained answers has a very low cost as a direct consequence of [9,10,22,13]. It incorporates negative normal form constraints instead of bindings and the decomposition step can produce disjunctions. More precisely, the normal form of constraints is:

$$\underbrace{\bigwedge_i (X_i = t_i)}_{\textbf{positive information}} \quad \wedge \quad \underbrace{(\bigwedge_j \forall\, \overline{Z}_j^1.\, (Y_j^1 \neq s_j^1) \vee \ldots \vee \bigwedge_l \forall\, \overline{Z}_l^n.\, (Y_l^n \neq s_l^n))}_{\textbf{negative information}}$$

where each X_i appears only in $X_i = t_i$, none s_k^r is Y_k^r and the universal quantification could be empty (leaving a simple disequality). Using some normalization rules we can obtain a normal form formula from any initial formula. It is easy to redefine the unification algorithm to manage constrained variables. This very compact way to represent a normal form was firstly presented in [16] and differs from Chan's representation where only disjunctions are used[4].

We have defined a predicate =/=/2 [17], used to check disequalities, in a similar way to explicit unification (=/2). Each constraint is a disjunction of conjunctions of disequalities that are implemented as a list of lists of terms as T_1/T_2 (that represents the disequality $T_1 \neq T_2$). When a universal quantification is used in a disequality (e.g., $\forall Y.\, X \neq c(Y)$) the new constructor *fA*/1 is used (e.g., *X*/*c*(*fA*(*Y*))).

Implementation of Universal Quantification We implement universal quantification by means of a predicate *forall*/2 that is the direct implementation of the predicate defined in definition 12. The execution of $\forall([X_1, ..., X_n], Q)$ implies a covering evaluation $eval(C_1, \forall\overline{X}.Q)$.

We have optimized the covering evaluation process to avoid repeating the evaluation of the same term in successive coverings[5].

We will keep a list with the content of the first covering and we will update the list with the elements of the second covering and so on. We will check at each step of the evaluation the value of the quantified goal for all elements of the covering at that step.

But all ground terms of the covering (e.g., terms without Skolem constants) and terms with universal variables are only evaluated once. To perform this optimization,

[4] Chan treats the disjunctions by means of backtracking. The main advantage of our normal form is that the search space is drastically reduced.

[5] Covering in this section refers to coverings with Skolem constants.

we keep a queue (implemented as a list). Expanded terms, from one covering to the next one, will be added to the bottom of the queue and only these new terms should be evaluated in the next step of the evaluation.

In the evaluation of a term t for the goal Q:

- If $Q[\overline{X} \mapsto t] = \underline{\mathrm{t}}$ then it will succeed in all the following coverings, so it is eliminated from the queue and we continue evaluating the other terms in the queue.
- If $Q[\overline{X} \mapsto t] = \underline{\mathrm{f}}$ then $\forall\overline{X}.Q$ fails and the evaluation process finishes.
- If $Q[\overline{X} \mapsto t] = \underline{\mathrm{u}}$ then $\forall\overline{X}.Q$ loops.

The key to the completeness of the process lies in the selection of the term of the queue to be evaluated first.

If we use the left-to-right Prolog selection rule, then the implementation is incomplete, although more efficient. This happens when we evaluate a term t and $Q[\overline{X} \mapsto t] = \underline{\mathrm{u}}$ and there is another term t' in the queue such that $Q[\overline{X} \mapsto t'] = \underline{\mathrm{f}}$. So, in these cases, $forall([X_1, ..., X_n], Q)$ loops instead of fails.

This is the reason why it is important to use a fair selection rule to assure that if there is a failure evaluation of a term, then it will be found before looping in a undefined evaluation of another term of the queue.

As we have seen in the definition of the quantifier, this implementation is correct and if we use a fair selection rule to chose the order in which the terms of the covering are visited then it is also complete. We can implement this because in Ciao Prolog is possible to ensure AND-fairness by goal shuffling [8].

If we implement this in a Prolog compiler using depth first SLD-resolution, the selection will make the process incomplete. When we are evaluating a covering $C = \{t_1, ..., t_m\}$ for a goal $Q(X)$ if we use the depth first strategy from the left to the right, we can find that the evaluation of a $Q(t_j)$ is unknown and is there exists one $k > j$ such that t_k is ground and $Q(t_k)$ fails, then we would obtain that $forall([X], Q(X))$ is unknown when indeed it is false.

6 Conclusion and Future Work

The experimental results are very encouraging. Table 1 includes some measurements of execution times (in milliseconds [6]) to get the first solution of a list of positive goals, their negation using negation as failure (*naf*) when applicable, their negation using our implementation of the constructive classical negation of Chan (*cneg*), and their negation using CIN (*intneg*). The ratios of the constructive negation and the CIN w.r.t. the positive goal and the ratio of the constructive negation w.r.t. CIN are shown.

We present three set of examples. The first one collects some examples where it is slightly worst or a little better to use CIN instead of negation as failure. The second set contains examples in which negation as failure cannot be used. CIN is very efficient in the sense that the execution time is similar as the time needed to execute the positive goal. The third set of examples is the most interesting because contains more

[6] All measurements were made using Ciao Prolog 1.5 on a Pentium II at 350 Mhz. Small programs were executed a sufficient number of times to get respectable data.

goals - G	**G**	**naf(G)**	**cneg(G)**	ratio	**intneg(G)**	ratio	**cneg/intneg**
boole(1)	780	829	1319	1.69	780	1.00	1.69
positive(500)	1450	1810	2090	1.44	3430	2.36	0.60
positive(700)	1450	1810	2099	1.44	5199	3.58	0.40
positive(1000)	2070	2370	2099	1.01	8979	4.33	0.23
less(0,50000)	1189	1199	23209	19.51	6520	5.48	3.55
less(50000,0)	1179	1140	4819	4.08	10630	9.01	0.45
average				**4.86**		**4.29**	**1.15**
boole(X)	829	–	1340	1.61	830	1.00	1.61
ancestor(peter,X)	820	–	1350	1.64	821	1.00	1.64
average				**1.63**		**1.00**	**1.62**
less(50000,X)	1430	–	28159	19.69	6789	4.74	4.14
less(100000,X)	1930	–	54760	28.37	13190	6.83	4.15
less(X,50000)	1209	–	51480	42.58	12210	10.09	4.21
less(X,100000)	1580	–	102550	64.90	24099	15.25	4.25
add(X,Y,100000)	2219	–	25109	11.31	4209	1.81	5.96
add(100000,Y,Z)	2360	–	12110	5.10	2659	1.12	4.55
add(X,100000,Z)	2160	–	23359	10.81	2489	1.15	9.38
average				**26.10**		**5.86**	**5.23**

Table 1. Runtime comparison

complex goals where negation as failure cannot be used and the speed-up of CIN over constructive negation is over 5 times better.

CIN is just a part of a more general project to implement negation where several other approaches are used: negation as failure possibly combined with dynamical goal reordering, IN, and constructive negation. The decision of what approach can be used is fixed by the information of different program analyses: naf in case of groundness of statically detected goal reordering, finite constructive negation in case of finiteness of the number of solutions, etc. See [18] for details. Notice that the strategy also ensures the soundness of the method [19]: if the analysis is correct, the precondition to apply a technique is ensured, so the results are always sound.

The main contributions of our CIN approach are:

- The formulation in terms of disequality constraints, solving some of the problems of IN [6] in a more efficient way than [7] because of the definition of the universal quantification without using the program code.
- The computation of universally quantified goals, that was sketched in [6], and we provide here an implementation and a discussion about its soundness and completeness.
- To our knowledge it is one of the first serious attempts to include intensional negation in a Prolog compiler.

As a future work, we plan to include this implementation into a compiler in order to produce a version of Ciao Prolog with negation. Our implementation of constructive CIN can be generalized to other simple Prolog Systems with the attributed variables extension that is the only requirement for implementing disequality constraints.

As Ciao allows CLP over various constraint domains, one might wonder if our method is directly applicable. One problem is that lemma 2 relies on a property of Herbrand constraints not enjoyed, in general, by other domains. Other than that, the only special requirement put by the transformation on the constraint domain is the ability of computing, from an *admissible* constraint c a constraint $\neg\exists\overline{x}.c$. This is, strictly speaking, a bit stronger than the *admissible closure* requirement of [22], as this does not imply the existence of disjunctive constraints, but obtaining a domain with constructive disjunction from one satisfying admissible closure is almost trivial. See [12] for an account of the necessity of admissible closure for constructive negation. The extension of the technique that implements the universal quantification to other constraint domains remains open.

Another field of work is the optimization of the implementation of the *forall using* different search techniques and more specialized ways of generating the coverings of the Herbrand Universe of our negation subsystem.

References

[1] The DLV home page. http://www.dbai.tuwien.ac.at/proj/dlv/.

[2] The ECLiPSe website at Imperial College. http://www-icparc.doc.ic.ac.uk/eclipse/.

[3] The SMODELS home page. http://www.tcs.hut.fi/Software/smodels/.

[4] The XSB home page. http://www.cs.sunysb.edu/~sbprolog/xsb-page.html.

[5] R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. Intensional negation of logic programs. *Lecture notes on Computer Science,* 250:96–110, 1987.

[6] R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *JLP,* 8(3):201–228, 1990.

[7] P. Bruscoli, F. Levi, G. Levi, and M.C. Meo. Compilative constructive negation in constraint logic programs. In Sophie Tyson, editor, *Proc. of the Nineteenth International Colloquium on Trees in Algebra and Programming, CAAP '94,* volume 787 of *LNCS,* pages 52–67, Berlin, 1994. Springer-Verlag.

[8] F. Bueno. *The CIAO Multiparadigm Compiler: A User's Manual,* 1995.

[9] D. Chan. Constructive negation based on the completed database. In *Proc. Int. Conference on LP'88,* pages 111–125. The MIT Press, 1988.

[10] D. Chan. An extension of constructive negation and its application in coroutining. In *Proc. NACLP'89,* pages 477–493. The MIT Press, 1989.

[11] N. J. Cutland. *Computability. An Introduction to Recursive Function Theory.* Cambridge University Press, 1980.

[12] Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. A necessary condition for constructive negation in constraint logic programming. *Information Processing Letters,* 74:147–156, 2000.

[13] F. Pages. Constructive negation by pruning. *Journal ofLogic Programming,* 32(2), 1997.

[14] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language.* The MIT Press, 1994.

[15] K. Kunen. Negation in logic programming. *Journal ofLogic Programming,* 4:289–308, 1987.

[16] J.J. Moreno-Navarro. Default rules: An extension of constructive negation for narrowing-based languages. In *Proc. ICLP'94,* pages 535–549. The MIT Press, 1994.

[17] S. Muñoz-Hernández and J.J. Moreno-Navarro. How to incorporate negation in a Prolog compiler. In V. Santos Costa E. Pontelli, editor, *2nd International Workshop PADL'2000,* volume 1753 of *LNCS,* pages 124–140, Boston, MA (USA), 2000. Springer.

[18] S. Muñoz-Hernández, J.J. Moreno-Navarro, and M. Hermenegildo. Efficient negation using abstract interpretation. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence and Reasoning*, La Habana (Cuba), 2001.

[19] S. Muñoz-Hernández. *A Negation System for Prolog.* PhD thesis, Facultad de Informática (Universidad Politécnica de Madrid), 2003.

[20] L. Naish. Negation and quantifiers in NU-Prolog. In *Proc. 3rd ICLP,* 1986.

[21] A. Di Pierro, M. Martelli, and C. Palamidessi. Negation as instantiation. *Information and Computation,* 120(2):263–278, 1995.

[22] P. Stuckey. Negation and constraint logic programming. In *Information and Computation*, volume 118(1), pages 12–33, 1995.

Analysing Definitional Trees: Looking for Determinism*

Pascual Julián Iranzo and Christian Villamizar Lamus

Departamento de Informática
Universidad de Castilla–La Mancha
Ciudad Real, Spain
Pascual.Julian@uclm.es, cvillami@inf-cr.uclm.es

Abstract. This paper describes how high level implementations of (needed) narrowing into Prolog can be improved by analysing definitional trees. First, we introduce a refined representation of definitional trees that handles properly the knowledge about the inductive positions of a pattern. The aim is to take advantage of the new representation of definitional trees to improve the aforementioned kind of implementation systems. Second, we introduce selective unfolding transformations, on determinate atom calls in the Prolog code, by examining the existence of what we call "deterministic (sub)branches" in a definitional tree. As a result of this analysis, we define some generic algorithms that allow us to compile a functional logic program into a set of Prolog clauses which increases determinism and incorporates some refinements that axe obtained by *ad hoc* artifices in other similar implementations of functional logic languages. We also present and discuss the advantages of our proposals by means of some simple examples.

Keywords: Functional logic programming, narrowing strategies, implementation of functional logic languages, program transformation.

1 Introduction

Functional logic programming [12] aims to implement programming languages that integrate the best features of both functional programming and logic programming. Most of the approaches to the integration of functional and logic languages consider term rewriting systems as programs and some narrowing strategy as complete operational mechanism. Laziness is a valuable feature of functional logic languages, since it increases the expressive power of this kind of languages: it supports computations with infinite data structures and a modular programming style. Among the different lazy narrowing strategies, needed narrowing [6] has been postulated optimal from several points of view. Needed narrowing addresses computations by means of some structures, namely definitional trees [2], which contain all the information about the program rules.

* Supported by CICYT TIC 2001-2705-C03-01, Acción Integrada Hispano-Italiana HI2000-0161, and Acción Integrada Hispano-Alemana HA2001-0059.

© Springer-Verlag Berlin Heidelberg 2004

These structures allow us to select a position of the term which is being evaluated and this position points out to a reducible subterm that is "unavoidable" to reduce in order to obtain the result of the computation. It is accepted that the framework for declarative programing based on non–deterministic lazy functions of [18] also uses definitional trees as part of its computational mechanism. In recent years, a great effort has been done to provide the integrated languages with high level implementations of this computational model into Prolog (see for instance [5,7,13,16] and [19]). This paper investigates how an analysis of definitional trees can introduce improvements in the quality of the Prolog code generated by these implementation systems.

The paper is organized as follows: Section 2 recalls some basic notions we use in the rest of the sections. In Section 3 we describe a refined representation of definitional trees and we give an algorithm for their construction in the style of [15]. Section 4 introduces two new translation techniques: Section 4.1 discusses how to take advantage of the new representation of definitional trees to improve (needed) narrowing implementations; Section 4.2 presents an algorithm, guided by the structure of a definitional tree, which is able to produce the same effect as if a determinate unfolding transformation was applied on the compiled Prolog code. Section 5 presents some experiments that show the effectiveness of our proposals. Section 6 discuses the relation of our techniques to other research on functional logic programming and logic programming. Finally, Section 7 contains our conclusions.

2 Preliminaries

We consider first order expressions or *terms* built from symbols of the set of variables $\mathcal{X}$ and the set of function symbols $\mathcal{F}$ in the usual way. The set of terms is denoted by $\mathcal{T}(\mathcal{F},\mathcal{X})$. We sometimes write $f/n \in \mathcal{F}$ to denote that f is a n–ary function symbol. If t is a term different from a variable, $\mathcal{R}oot(t)$ is the function symbol heading t, also called the *root symbol* of t. A term is *linear* if it does not contain multiple occurrences of the same variable. $\mathcal{V}ar(o)$ is the set of variables occurring in the syntactic object o. We write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$.

A *substitution* σ is a mapping from the set of variables to the set of terms, with finite *domain* $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$. We denote the identity substitution by *id*. We define the composition of two substitutions σ and θ, denoted $\sigma \circ \theta$ as usual: $\sigma \circ \theta(x) = \hat{\sigma}(\theta(x))$, where $\hat{\sigma}$ is the extension of σ to the domain of the terms. A *renaming* is a substitution ρ such that there exists the inverse substitution ρ^{-1} and $\rho \circ \rho^{-1} = \rho^{-1} \circ \rho = id$.

A term t is *more general* than s (or s is an *instance* of t), in symbols $t \leq s$, if $(\exists\sigma)\ s = \sigma(t)$. Two terms t and t' are *variants* if there exists a renaming ρ such that $t' = \rho(t)$. We say that t is *strictly* more general than s, denoted $t < s$, if $t \leq s$ and t and s are not variants. The quasi–order relation "$\leq$" on terms is often called *subsumption order* and "$<$" is called *strict subsumption order.*

Positions of a term t (also called *occurrences)* are represented by sequences of natural numbers used to address subterms of t. The concatenation of the sequences p and w is denoted by $p.w$. Two positions p and p' of t are *comparable* if $(\exists w)\ p' = p.w$ or $p = p'.w$, otherwise are *disjoint* positions. Given a position p of t, $t|_p$ denotes the subterm of t at position p and $t[s]_p$ denotes the result of replacing the subterm $t|_p$ by the term s. Let $\overline{p_n}$ be a sequence of disjoint positions of a term t, $t[s_1]_{p_1} \ldots [s_n]_{p_n}$ denotes the result of simultaneously replacing each subterm $t|_{p_i}$ by the term s_i, with $i \in \{1, \ldots, n\}$.

2.1 Term Rewriting Systems

We limit the discussion to unconditional term rewriting systems[1]. A *rewrite rule* is a pair $l \to r$ with $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. The terms l and r are called the *left–hand side* (lhs) and *right–hand side* (rhs) of the rewrite rule, respectively. A *term rewriting system* (TRS) $\mathcal{R}$ is a finite set of rewrite rules.

We are specially interested in TRSs whose associate signature $\mathcal{F}$ can be partitioned into two disjoint sets $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ where $\mathcal{D} = \{\mathcal{R}oot(l) \mid (l \to r) \in \mathcal{R}\}$ and $\mathcal{C} = \mathcal{F} \backslash \mathcal{D}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions* or *operations*. Terms built from symbols of the set of variables $\mathcal{X}$ and the set of constructors $\mathcal{C}$ are called *constructor terms*. A *pattern* is a term of the form $f(\overline{d_n})$ where $f/n \in \mathcal{D}$ and $\overline{d_n}$ are constructor terms. A term $f(\overline{x_n})$, where $\overline{x_n}$ are different variables, is called a generic pattern. A TRS is said to be *constructor–based* (CB) if the lhs of its rules are patterns. For CB TRSs, a term t is a *head normal form* (hnf) if t is a variable or $\mathcal{R}oot(t) \in \mathcal{C}$.

A TRS is said to be *left–linear* if for each rule $l \to r$ in the TRS, the lhs l is a linear term. We say that a TRS is *non–ambiguous* or *non–overlapping* if it does not contain critical pairs (see [9] for a standard definition of a critical pair). Left–linear and non–ambiguous TRSs are called *orthogonal* TRSs.

Inductively sequential TRSs are a proper subclass of CB orthogonal TRSs. The definition of this class of programs make use of the notion of *definitional tree*. For the sake of simplicity and because further complications are irrelevant for our study, in the following definition, we ignore the *exempt* nodes that appear in the original definition of [2] and also the *or*–nodes of [16] used in the implementation of Curry [15]. Note also, that or–nodes lead to parallel definitional trees and thus out of the class of inductively sequential systems.

Definition 1. [Partial definitional tree]
Given a CB TRS $\mathcal{R}$, $\mathcal{P}$ is a partial definitional tree *with pattern π if and only if one of the following cases hold:*

1. *$\mathcal{P} = rule(\pi, l \to r)$, where π is a pattern and $l \to r$ is a rewrite rule in $\mathcal{R}$ such that π is a variant of l.*
2. *$\mathcal{P} = branch(\pi, o, \overline{\mathcal{P}_k})$, where π is a pattern, o is a variable position of π (called* inductive position*), $\overline{c_k}$ are different constructors, for some $k > 0$, and for all $i \in \{1, \ldots, k\}$, $\mathcal{P}_i$ is a partial definitional tree with pattern $\pi[c_i(\overline{x_n})]_o$, where n is the arity of c_i and $\overline{x_n}$ are new variables.*

[1] This is not a true limitation for the expressiveness of a programming language relaying on this class of term rewriting systems [4].

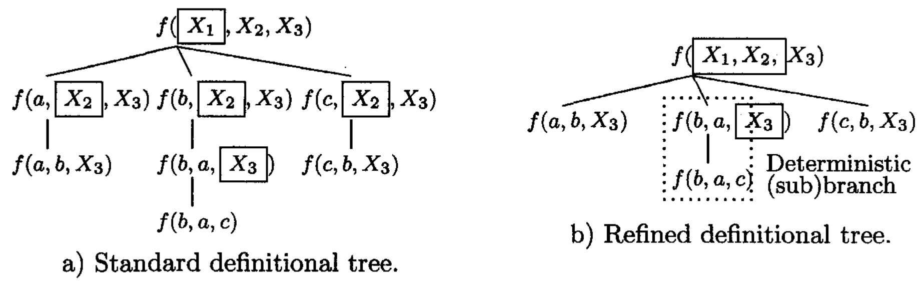


Fig. 1. Definitional trees for the function "f" of Example 1

From a declarative point of view, a partial definitional tree $\mathcal{P}$ can be seen as a set of linear patterns partially ordered by the strict subsumption order "<" [3]. Given a defined function f/n, a *definitional tree of* f is a partial definitional tree whose pattern is a generic pattern and its leaves contain variants of all the rewrite rules defining f.

Example 1. Given the rules defining the function $f/3$

$$R_1 : f(a,b,X) \to r_1, \quad R_2 : f(b,a,c) \to r_2, \quad R_3 : f(c,b,X) \to r_3.$$

a definitional tree of f is:

$$\begin{array}{l} branch(f(X_1,X_2,X_3),1, \\ \quad branch(f(a,X_2,X_3),2,rule(f(a,b,X_3),R_1)), \\ \quad branch(f(b,X_2,X_3),2,branch(f(b,a,X_3),2,rule(f(b,a,c),R_2))), \\ \quad branch(f(c,X_2,X_3),2,rule(f(c,b,X_3),R_3))) \end{array}$$

Note that there can be more than one definitional tree for a defined function. It is often convenient and simplifies understanding to provide a graphic representation of definitional trees, where each node is marked with a pattern and the inductive position in branches is surrounded by a box. Figure 1(a) illustrates this concept.

Definition 2. [Inductively Sequential TRS]
A defined function f is called inductively sequential *if it has a definitional tree. A rewrite system $\mathcal{R}$ is called inductively sequential if all its defined functions are inductively sequential.*

In this paper we are mainly interested in inductively sequential TRSs (or proper subclasses of them) which are called *programs.*

2.2 Definitional Trees and Narrowing Implementations into Prolog

Most of the relevant implementations of functional logic languages, which use needed narrowing as operational mechanism, are based on the compilation of the programs written in these languages into Prolog [7,13,16,17]. These implementation systems may be thought as a translation process that essentially consists in the following:

1. An algorithm to transform the program rules in a functional logic program into a set of definitional trees (See [16] and [15] for some of those algorithms).
2. An algorithm that takes the definitional trees as an input parameter and visits their nodes, generating a Prolog clause for each visited node. Since definitional trees contain all the information about the original program as well as information to guide the (optimal) pattern matching process during the evaluation of expressions, the set of generated Prolog clauses is able to simulate the intended narrowing strategy being implemented.

In the case of functional logic programs with a needed narrowing semantics, a generic algorithm for the translation of definitional trees into a set of clauses is given in [13]. When we apply that algorithm to the definitional tree of function f in Example 1, we obtain the following set of Prolog clauses:

```
% Clause for the root node: it exploits the first inductive position
f(X1, X2, X3, H) :- hnf(X1, HX1), f_1(HX1, X2, X3, H).

% Clauses for the remaining nodes:
f_1(a, X2, X3, H):- hnf(X2, HX2), f_1_a_2(HX2, X3, H).
f_1_a_2(b, X3, H):- hnf(r1, H).

f_1(b, X2, X3, H):- hnf(X2, HX2), f_1_b_2(HX2, X3, H).
f_1_b_2(a, X3, H):- hnf(X3, HX3), f_1_b_2_a_3(HX3, H).
f_1_b_2_a_3(c, H):- hnf(r2, H).

f_1(c, X2, X3, H):- hnf(X2, HX2), f_1_c_2(HX2, X3, H).
f_1_c_2(b, X3, H):- hnf(r3, H).
```

where hnf(T, H) is a predicate that is true when H is the hnf of a term T. For this example, the clauses defining the predicate hnf are:

```
% Evaluation to head normal form (hnf).
hnf(T, T) :- var(T), !.
hnf(f(X1, X2, X3), H) :- !, f(X1, X2, X3, H).
hnf(T, T). % otherwise the term T is a hnf;
```

The meaning of these set of clauses is as follows. For evaluating a term $t = f(t_1, t_2, t_3)$ to a hnf, first, it is necessary to evaluate (to a hnf) the subterms of t at the inductive positions of the patterns in the definitional tree associated with f (in the order dictated by that definitional tree — see Figure 1(a)). Hence, for our example: we compute the hnf of t_1 and then the hnf of t_2; if b is the hnf of t_1 and a is the hnf of t_2, we have to compute the hnf of t_3; if the hnf of t_3 is c then the hnf of t will be the hnf of r_2 else the computation fails (see the sixth clause). On the other hand, if the hnf of t_1 is a or c it suffices to evaluate t_2 to a hnf, disregarding t_3, in order to obtain the final value. This evaluation mechanism conforms with the needed narrowing strategy of [6], as it has been formally demonstrated in [1].

3 A Refined Representation of Definitional Trees

As we have just seen, building definitional trees is the first step of the compilation process in high level implementations of needed narrowing into Prolog. Therefore, providing a suitable representation structure for the definitional trees associated with a functional logic program may be an important task in order to improve those systems. In this section we give a refined representation of definitional trees that saves memory allocation and is the basis for further improvements.

It is noteworthy that the function f of Example 1 has two definitional trees: the one depicted in Figure 1(a) and a second one obtained by exploiting position 2 of the generic pattern $f(X_1, X_2, X_3)$. Hence, this generic pattern has two inductive positions. As we are going to show, we can take advantage of this situation if we "simultaneously" exploit these two inductive positions. The main idea of the refinement is as follows: when a pattern has several inductive positions, exploit them altogether. Therefore we need a criterion to detect inductive positions. This criterion exists and it is based on the concept of uniformly demanded position, which was introduced into the functional logic setting by J. Moreno-Navarro, and M. Rodríguez-Artalejo *et al.* (see, for instance, [16]).

Definition 3. [Uniformly demanded position]
Given a pattern π and a TRS $\mathcal{R}$, Let be $\mathcal{R}_\pi = \{l \to r | (l \to r) \in \mathcal{R} \wedge \pi \leq l\}$. A variable position p of the pattern π is said to be: (i) demanded *by a lhs l of a rule in $\mathcal{R}_\pi$ if $\mathcal{R}oot(l|_p) \in \mathcal{C}$. (ii)* uniformly demanded *by $\mathcal{R}_\pi$ if p is demanded by all lhs in $\mathcal{R}_\pi$.*

We write $\mathcal{UDP}os(\pi)$ to denote the set of uniformly demanded positions of the pattern π. The following proposition establishes a necessary condition for a position of a pattern to be an inductive position.

Proposition 1. *Let $\mathcal{R}$ be an inductively sequential TRS and let π be the pattern of a branch node of a definitional tree $\mathcal{P}$ of a function defined in $\mathcal{R}$. If o is an inductive position of π then o is uniformly demanded by $\mathcal{R}_\pi$.*

The converse proposition is more involved but not difficult to establish. In the following, given two partial definitional trees $\mathcal{P}_1$ and $\mathcal{P}_2$, we say $\mathcal{P}_1 \preceq \mathcal{P}_2$ if and only if $\mathcal{P}_1 = \mathcal{P}_2$ or $\mathcal{P}_1 \prec \mathcal{P}_2$, where $\mathcal{P}_1 \prec \mathcal{P}_2$ if $\mathcal{P}_1$ is a proper subtree of $\mathcal{P}_2$.

Proposition 2. *Let $\mathcal{R}$ be an inductively sequential TRS. Let $\mathcal{P}$ a partial definitional tree, with pattern π, and o a variable position of π. If o is uniformly demanded by $\mathcal{R}_\pi$ then there exists a partial definitional tree $\mathcal{P}' \preceq \mathcal{P}$, with pattern π', such that o is an inductive position of π'.*

Hence, the concept of uniformly demanded position and Proposition 1 give us a syntactic criterion to detect if a variable position of a pattern is an inductive position or not and, therefore, a guideline to built a definitional tree: (i) Given a branch node, select a uniformly demanded position of its pattern; fix it as an inductive position of the branch node and generate the corresponding child nodes. (ii) If the node doesn't have uniformly demanded positions then there are two possibilities: the node is a leaf node, if it is a variant of a lhs of the considered TRS, or it is a "failure" node, and it is impossible to build the definitional tree. The following algorithm, in the style of [15], uses this scheme to build a refined partial definitional tree $rpdt(\pi, \mathcal{R}_\pi)$ for a pattern π and rules $\mathcal{R}_\pi = \{l \to r \mid (l \to r) \in \mathcal{R} \wedge \pi \leq l\}$:

1. If $\mathcal{UDPos}(\pi) = \emptyset$ and there is only one rule $(l \to r) \in \mathcal{R}_\pi$ and a renaming ρ such that $\pi = \rho(l)$:

   $$rpdt(\pi, \mathcal{R}_\pi) = rule(\pi, \rho(l) \to \rho(r));$$

2. If $\mathcal{UDPos}(\pi) \neq \emptyset$ and for all $(c_{i_1}, \ldots, c_{i_m}) \in \mathcal{C}_\pi$, $\mathcal{P}_i = rpdt(\pi_i, \mathcal{R}_{\pi_i}) \neq \mathtt{fail}$:

   $$rpdt(\pi, \mathcal{R}_\pi) = branch(\pi, \overline{o_m}, \overline{\mathcal{P}_k});$$

   where $\overline{o_m}$ is the sequence of uniformly demanded positions in $\mathcal{UDPos}(\pi)$, $\mathcal{C}_\pi = \{(c_{i_1}, \ldots, c_{i_m}) | (l_i \to r_i) \in \mathcal{R}_\pi \wedge \mathcal{R}oot(l_i|_{o_1}) = c_{i_1} \wedge \ldots \wedge \mathcal{R}oot(l_i|_{o_m}) = c_{i_m}\}$, $k = |\mathcal{C}_\pi| > 0$, $\pi_i = \pi[c_{i_1}(\overline{x_{n_{i_1}}})]_{o_1} \ldots [c_{i_m}(\overline{x_{n_{i_m}}})]_{o_m}$ and $\overline{x_{n_{i_1}}}, \ldots, \overline{x_{n_{i_m}}}$ are new variables.
3. Otherwise, $rpdt(\pi, \mathcal{R}_\pi) = \mathtt{fail}$.

Given an inductively sequential TRS $\mathcal{R}$ and a n–ary defined function f in $\mathcal{R}$, the definitional tree of f is $rdt(f, \mathcal{R}) = rpdt(\pi_0, \mathcal{R}_{\pi_0})$ where $\pi_0 = f(\overline{x_n})$. Note that, for an algorithm like the one described in [15] the selection of the inductive positions of the pattern π is non–deterministic, if $\mathcal{UDPos}(\pi) \neq \emptyset$. Therefore, it is possible to build different definitional trees for an inductively sequential function, depending on the inductive position which is selected. On the contrary, our algorithm deterministically produces a single definitional tree for each inductively sequential function. Note also that it matches the more informal algorithm that appears in [15] when, for each branch node, there is only one inductive position.

For the defined function f in Example 1, the last algorithm builds the following definitional tree:

$$\begin{array}{l} branch(f(X_1, X_2, X_3), (1,2), \\ \quad rule(f(a, b, X_3), R_1), \\ \quad branch(f(b, a, X_3), (3), rule(f(b, a, c), R_2)), \\ \quad rule(f(c, b, X_3), R_3)) \end{array}$$

which is depicted in Figure 1(b). As we said, for this example, the standard algorithm of [15] may build two definitional trees for f (depending on whether position 1 or position 2 is selected as the inductive position of the generic pattern $f(X_1, X_2, X_3)$). Both of these trees have eight nodes, while the new representation cuts the number of nodes of the definitional tree to five nodes. We claim that the new representation reduces the number of nodes in general and, also, the number of possible definitional trees associated with a function (actually, there is only one refined definitional tree for each defined function).

As it has been proposed in [7], it is possible to obtain a simpler translation scheme of functional logic programs into Prolog if definitional trees are first compiled into *case expressions.* That is, functions are defined by only one rule where the lhs is a generic pattern and the rhs contains case expressions to specify the pattern matching of actual arguments. The use of case expressions doesn't invalidate our argumentation. Thus, we can transform the last refined definitional tree in the following case expression:

$$\begin{array}{ll} f(X_1, X_2, X_3) = \texttt{case} & (X_1, X_2)\ \texttt{of} \\ & (a, b) \to r_1 \\ & (b, a) \to \texttt{case}\ (X_3)\ \texttt{of}\ (c) \to r_2 \\ & (c, b) \to r_3 \end{array}$$

A case expression, like this, will be evaluated by reducing a tuple of arguments to their hnf and matching them with one of the patterns of the case expression.

4 Improving Narrowing Implementations into Prolog

This section discusses two improvements in the translation of non-strict functional logic programs into Prolog which are based on the analysis of definitional trees. These translation techniques can be applied jointly or separately.

4.1 Translation Based on Refined Definitional Trees

The refined representation of definitional trees introduced in Section 3 is very close to the standard representation of definitional trees, but it is enough to provide further improvements in the translation of functional logic programs into Prolog.

It is easy to adapt the translation algorithm that appears in [13] to use our refined representation of definitional trees as input. If we apply this slightly different algorithm to the refined definitional tree of Figure 1(b), we obtain the following set of clauses, where the inductive positions 1 and 2 are exploited simultaneously:

```
% Clause for the root node:
f(X1, X2, X3, H):- hnf(X1, HX1), hnf(X2, HX2), f_1_2(HX1, HX2, X3, H).

% Clauses for the remaining nodes:
f_1_2(a, b, X3, H):- hnf(r1, H).
f_1_2(b, a, X3, H):- hnf(X3, HX3), f_1_2_b_a(HX3, H).
f_1_2_b_a(c, H):- hnf(r2, H).
f_1_2(c, b, X3, H):- hnf(r3, H).
```

where we have cut the number of clauses with regard to the standard representation into Prolog (of the rules defining function f) presented in Section 2.2. The number of clauses is reduced in the same proportion as the number of nodes of the standard definitional tree for f were cut. As we are going to show in Section 5, this refined translation technique is able to improve the efficiency of the implementation system.

On the other hand, it is important to note that the kind of improvements we are mainly studying in this subsection can not be obtained by an unfolding transformation process applied to the set of clauses produced by the standard algorithm of [13]: In fact, it is not possible to obtain the previous set of clauses by an unfolding transformation of the set of clauses shown in Section 2.2.

4.2 Selective Unfolding Transformations

The analysis of definitional trees provides further opportunities for improving the translation of inductively sequential programs into Prolog. For instance, we can take notice that the definitional tree of function f in Example 1 has a "deterministic" (sub)branch, that is, a (sub)branch whose nodes have only one child (see Figure 1(b)). This knowledge can be used as a heuristic guide for applying determinate unfolding transformation steps selectively.

Note that, for the example we are considering, the clauses:

```
f_1_2(b, a, X3, H):- hnf(X3, HX3), f_1_2_b_a(HX3, H). %% (C1)
f_1_2_b_a(c, H):- hnf(r2, H).                         %% (C2)
```

can be merged into:

```
f_1_2(b, a, X3, H):- hnf(X3, c), hnf(r2, H).          %% (C')
```

by applying a safe unfolding transformation in the style of Tamaki and Sato [21] but restricting ourselves to determinate atoms [10] (i.e., an atom that matches exactly one clause head in the Prolog code): we get clause C1 (the unfolded clause) and we select the atom f_1_2_b_a(HX3, H) in its body; this atom call is unifiable with the head of clause C2 (the *unique* unfolding clause for this atom call), with most general unifier $\sigma = \{\texttt{HX3/c}\}$ (actually, a matcher); Therefore, we can perform a transformation step where C1 and C2 are instantiated applying σ, the atom call is unfolded and, afterwards, clauses C1 and C2 are replaced by C'.

This selective unfolding is preferable to a generalized (post–compilation) unfolding transformation process[2] which may degrade the efficiency of the compiled Prolog code. Moreover, this selective unfolding transformation can be easily integrated inside the compilation procedure described in [13]. It suffices to introduce an additional case in order to treat deterministic (sub)branches:

```
⋮
$Trans(branch(\pi, o, \mathcal{T}^1), p) :=$
    if $\mathcal{T}^n = branch(\pi_n, o_n, \mathcal{T}')$
    $produceCode$ :
        $f_p(t_1, \ldots, t_m, \mathtt{H})$ :-
        hnf$(X, \pi_1|_o)$, hnf$(\pi_1|_{o_1}, \pi_2|_{o_1})$, ..., hnf$(\pi_{n-1}|_{o_{n-1}}, \pi_n|_{o_{n-1}})$,
        hnf$(\pi_n|_{o_n}, \mathtt{Y})$, $f_{p \cup \{o, o_1, \ldots, o_n\}}(t'_1, \ldots, t'_m, \mathtt{H})$.
    $Trans(\mathcal{T}', p \cup \{o, o_1, \ldots, o_n\})$;
```

[2] That is, a transformation process where non determinate atom calls are unfolded too.

```
        else if $\mathcal{T}^n = rule(\pi_n, \pi_n \to r)$
        produceCode :
            $f_p(t_1, \ldots, t_m, \mathrm{H})$ :-
            hnf($X$, $\pi_1|_o$), hnf($\pi_1|_{o_1}$, $\pi_2|_{o_1}$), ..., hnf($\pi_{n-1}|_{o_{n-1}}$, $\pi_n|_{o_{n-1}}$),
            hnf($r$, H).
    where $\pi = f(t_1, \ldots, t_m)$, $\pi|_o = X$, $\mathcal{T}^1, \ldots, \mathcal{T}^n$ is the sequence of nodes
    in the deterministic (sub)branch with $\mathcal{T}^i = branch(\pi_i, o_i, \mathcal{T}^{i+1})$, and
    $\pi_n[\mathrm{Y}]_{o_n} = f(t'_1, \ldots, t'_m)$;
      ⋮
    $Trans(\overline{\mathcal{T}_n}, p) := Trans(\mathcal{T}_1, p), \ldots, Trans(\mathcal{T}_n, p)$;
```

Now, each function f is translated by $Trans(\mathcal{T}, \emptyset)$, where $\mathcal{T}$ is a definitional tree of f.

Roughly speaking, the new case in the algorithm of [13] can be understood as follows. If there exists a deterministic (sub)branch visit its nodes in descending order forcing that the evaluation (to hnf) of the subterms at the inductive position o of a term be the flat constructor at position o of the child node. Proceed in this way until: i) a non deterministic node is reached; or ii) a leaf node is reached and, in this case, evaluate the rhs of the rule to its hnf and stop the translation.

The last algorithm allows some improvements we have omitted for the sake of simplicity. First, it is possible to eliminate redundant arguments. Second, it is possible to exploit rule nodes (i.e., an atom call like **hnf**(r, **H**)) to perform an additional determinate unfolding step[3]. Having all this in consideration, the following example illustrates the algorithm.

Example 2. Given the rules defining the partial function *even* and its definitional tree:

$R_1 : even(0) \to true,$
$R_2 : even(s(s(X))) \to even(X).$

$$branch(even(X_1), 1, \; rule(even(0), R_1), \; branch(even(s(X_2)), 1.1, \; rule(even(s(s(X_3))), R_2))))$$

the Prolog code generated by the *Trans* algorithm is:

```
% Evaluation to head normal form (hnf).
hnf(even(X1), H) :- !, even(X1, H).

% Clause for the root node: it exploits the first inductive position
even(X1, H) :- hnf(X1, HX1), even_1(HX1, H).
even_1(0, true).
% Clause for the deterministic (sub)branch:
even_1(s(X2), H) :- hnf(X2, s(X3)), even(X3, H).
```

Note as the determinate call `hnf(even(X3), H)` has been unfolded (into the call `even(X3, H)` using the first rule for evaluating a hnf).

[3] These improvements are implemented in the `curry2prolog` compiler of PAKCS [8] for the standard cases.

Therefore, our *Trans* algorithm, guided by the structure of a definitional tree, is able to reproduce the effect of a post-compilation unfolding transformation when it is applied selectively on determinate atom calls in the standard compiled Prolog code.

5 Experiments

We have made some experiments to verify the effectiveness of our proposals. We have instrumented the Prolog code obtained by the compilation of simple Curry programs by using the `curry2prolog` compiler of PAKCS [8] (an implementation of the multi–paradigm declarative language Curry [15]). We have introduced our translation techniques in the remainder Prolog code. For our first translation technique, the one using the refined representation of definitional trees, the results of the experiments are shown in Table 1. Runtime and memory occupation were measured on a Sun4 Sparc machine, running Sicstus v3.8 under SunOS v5.7. The "`Speedup`" column indicates the percentage of execution time saved by our translation technique. The values shown on that column are the percentage of the quantity computed by the formula $(t_1 - t_2)/t_1$, where t_1 and t_2 are the average runtimes, for several executions, of the proposed terms (goals) and Prolog programs obtained when we don't use (t_1) and we use (t_2) our translation technique. The "`G. stack Imp.`" column reports the improvement of memory occupation for the computation. We have measured the percentage of global stack allocation. The amount of memory allocation measured between each execution remains constant. Most of the benchmark programs are extracted from

Table 1. Runtime speed up and memory usage improvements for some benchmark programs and terms.

Benchmark	Term	Speedup	G. stack Imp.
family	$grandfather(\_,\_)$	19.9%	0%
geq	$geq(100000, 99999)$	4.6%	16.2%
geq	$geq(99999, 100000)$	4.3%	16.2%
xor	$xor(\_,\_)$	18.5%	0%
zip	$zip(L1, L2)$	3.6%	5.5%
zip3	$zip3(L1, L2, L2)$	4.5%	10%
	Average	9.2%	7.9%

[15] and the standard prelude for Curry programs with slight modifications[4]. For the benchmark programs `family` and `xor` we evaluate all outcomes. The natural numbers are implemented in Peano notation, using `zero` and `succ` as

[4] For example, `zip` (resp. `zip3`) is adapted for combining two (resp. three) lists of elements of equal length into one list of pairs (resp. triples) of the corresponding elements. However, this function also may be useful in a practical context (see [14], page 280).

constructors of the sort. In the zip and zip3 programs the input terms $L1$ and $L2$ are lists of length 9.

Regarding the second translation technique, the one which implements selective unfolding transformations, for the benchmark program of Example 2 we obtain an average speedup of 11.7% and an improvement in memory usage of 14.7%.

More detailed information about the experiments and benchmark programs can be found in http://www.inf-cr.uclm.es/www/pjulian/publications.html.

6 Discussion and Related Work

In this section we discuss some important issues and we put them in relation to other research on functional logic programming and logic programming when it is convenient.

Elimination of *ad hoc* artifices. It is noteworthy that, in some cases, the benefits of our first translation scheme are obtained in an *ad hoc* way in actual needed narrowing into Prolog implementation systems. For instance, the standard definition of the strict equality used in non–strict functional logic languages is [11,19]:

$$\begin{aligned} c == c &\to true \\ c(\overline{X_n}) == c(\overline{Y_n}) &\to X_1 == Y_1 \&\& \ldots \&\& X_n == Y_n \end{aligned}$$

where c is a constructor of arity 0 in the first rule and arity $n > 0$ in the second rule. There is one of these rules for each constructor that appears in the program we are considering. Clearly, the strict equality has an associate definitional tree whose pattern $(X_1 == X_2)$ has two uniformly demanded positions (positions 1 and 2) and, therefore, it can be translated using our first technique, that produces a set of Prolog clauses similar to the one obtained by the curry2prolog compiler. Thus, the curry2prolog compiler produces an optimal representation of the strict equality which is treated as a special system function with an *ad hoc* predefined translation into Prolog, instead of using the standard translation algorithm which is applied for the translation of user defined functions.

Failing derivations. Our first contribution, as well as the overall theory of needed evaluation, is interesting for computations that succeed. However it is important to say that some problems may arise when a computation does not terminate or fails. For example, given the (partial) function $\{f(a, a) \to a\}$ the standard compilation into Prolog is:

```
f(A,B,C)     :- hnf(A,F), f_1(F,B,C).
f_1(a,A,B) :- hnf(A,E), f_1_a_2(E,B).
f_1_a_2(a,a).
```

while our first translation technique produces:

```
f(A,B,C) :- hnf(A,F), hnf(B,G), f_1(F,G,C).
f_1(a,a,a).
```

Now, if we want to compute the term `f(b, expensive_term)`, the standard implementation detects the failure after the computation of the first argument. On the other hand, the new implementation computes the expensive term (to `hnf`) for nothing. Of course, the standard implementation has problems too —e.g. if we compute the term `f(expensive_term, b)`, it also computes the expensive term (to hnf)—, but it may have a better behavior on this problem. Thus, in a sequential implementation, the performance of our first translation technique may be in danger when subterms, at uniformly demanded positions, are evaluated (to hnf) jointly with an other subterm whose evaluation (to `hnf`) produces a failure. An alternative to overcome this practical disadvantage is to evaluate these subterms in parallel, introducing monitoring techniques able to detect the failure as soon as possible and then to stop the streams of the computation.

Clause indexing and direct implementation into Prolog. Clause indexing is a technique, used in the implementation of Prolog compilers, that aims to reduce the number of clauses on which unification with a goal is performed. In general, indexing techniques are based on the inspection of the outermost function symbol of one or more arguments in a clause head. If the predicate symbol and the respective indexed symbols of the clause head and the goal coincide, then the clause is selected as part of the *filtered set.* Afterwards, the set of clauses in the filtered set (presumably smaller than the original one) is attempted to unify with the goal. More sophisticated indexing techniques such as those described in [20] perform indexing on all non variable symbols of a clause head (loosing no significant structural information). Also, these techniques are able to obtain the unifier during the indexing process. Although it seems to have some similarities between indexing techniques and the standard operational mechanism of functional logic languages, there is a big difference: in the context of pure logic languages terms are *dead* structures. However, in the context of this work, the concept of evaluation strategy relies on the existence and manipulation of nested *alive* terms. The needed narrowing strategy, as defined in [6], is an application from terms and partial definitional trees to sets of triples (position, rule, substitution), where each triple gives the position of a term, the rule of the program and the unifier substitution (not necessarily a most general one) used in a narrowing step. Our work is concerned in the optimization of certain implementation techniques of needed narrowing into Prolog.

On the other hand, a direct representation of a function into Prolog is possible, which is often more efficient, since term structures with nested functions calls are not generated. However, a direct implementation corresponds to a call-by-value strategy, that lacks some valuable properties (as the ability of handle infinite data structures or a good termination behavior) [13].

Determinate unfolding. Determinate unfolding [10] has been proposed as a way to ensure that the specialization of a logic program will never duplicate computations. The advantages of determinate unfolding transformations, in the context of the implementation of functional logic languages into Prolog, were suggested in [13] and [7]. They proposed to apply determinate unfolding as a post-compilation process but actually, in the `curry2prolog` compiler, determinate unfolding steps are only applied to unfold the atom calls produced by rule nodes. The novel of our proposal is that it exploits all opportunities for determinate unfolding in a systematic way and it is embedded inside the compilation process.

7 Conclusions

In this paper we have introduced a refined representation of definitional trees that eliminates the indeterminism in the selection of definitional trees in the context of the needed narrowing strategy (there is only one refined definitional tree for each inductively sequential function). We have defined two translation techniques based on the analysis of (refined) definitional trees. Although the results of the experiments section reveal a good behavior of these translation techniques, it is difficult to evaluate which may be their impact over the whole system, since the improvements appear when we can detect patterns that have several uniformly demanded positions or the existence of deterministic (sub)branches in a (refined) definitional tree. Nevertheless, our work shows that there is a potential for the improvement of actual (needed) narrowing implementation systems: we obtain valuable improvements of execution time and memory allocation when our translation techniques are relevant. For the case of inductively sequential functions without the features aforementioned, our translation schemes are conservative and don't produce runtime speedups or memory allocation improvements. Although failing derivations are rather a problematic case where the performance of our first translation technique may be in danger, we can deal with these problem by introducing concurrent computations, in order to guarantee that slowdowns, with regard to standard implementations of needed narrowing into Prolog, are not produced. Hence, the occurrence of several inductive position in a pattern can be considered as a signal for exploiting implicit parallelism.

On the other hand, our simple translation techniques are able to eliminate some *ad hoc* artifices in actual implementations of (needed) narrowing into Prolog, providing a systematic and efficient translation mechanism. Moreover, the ideas we have just developed can be introduced with a modest programming effort in standard implementations of needed narrowing into Prolog (such as the PAKCS [8] implementation of Curry) and in other implementations based on the use of definitional trees (e.g., the implementation of the functional logic language $\mathcal{TOY}$ [17]), since they don't modify their basic structures.

Acknowledgements

We gratefully acknowledge the anonymous referees for their valuable suggestions and Sergio Antoy for clarifying us some aspects of the theoretical/practical dimension of the failing derivations problem. Also, we thank Ginés Moreno for many useful discussions on the advantages of introducing selective unfolding transformations.

References

1. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Uniform Lazy Narrowing. *Journal of Logic and Computation,* 13(2):27, March/April 2003.
2. S. Antoy. Definitional trees. In *Proc. of ALP'92,* volume 632 of *LNCS,* pages 143–157. Springer-Verlag, 1992.
3. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of ALP'97,* volume 1298 of *LNCS,* pages 16–30. Springer-Verlag, 1997.
4. S. Antoy. Constructor-based conditional narrowing. In *Proc. of (PPDP'01).* Springer LNCS, 2001.
5. S. Antoy. Needed Narrowing in Prolog. In *Proc. of PLILP'96, LNCS,* pages 473–474. 1996.
6. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM,* 47(4):776–822, July 2000.
7. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. of FroCoS 2000,* pages 171–185. Springer LNCS 1794, 2000.
8. S. Antoy, M. Engelke, M. Hanus, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. Packs 1.5 : The Portland Aachen Kiel Curry System User Manual. Technical Report Version of May, 23, University of Kiel, Germany, 2003. Available from URL: http://www.informatik. uni-kiel.de/~pakcs/
9. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
10. J. Gallagher and M. Bruynooghe. Some Low-Level Source Transformations for Logic Programs. In *Proc. of 2nd Workshop on Meta-Programming in Logic,* pages 229–246. 1990.
11. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences,* 42:363–377, 1991.
12. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming,* 19&20:583–628, 1994.
13. M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. LOPSTR'95,* pages 252–266. Springer LNCS 1048, 1995.
14. M. Hanus and F. Huch. An open system to support web-based learning. In *Proc. of WFLP 2003,* pages 269–282. Universidad Politécnica de Valencia, 2003.
15. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at http://www. informatik.uni-kiel.de/~curry, 1999.
16. R. Loogen, F. López-Fraguas, and M. Rodríguez - Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of PLILP'93,* pages 184–200. Springer LNCS 714, 1993.
17. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99,* pages 244–247. Springer LNCS 1631, 1999.
18. J. G. Moreno, M. H. González, F. López-Fraguas, and M. R. Artalejo. An Approach to Declarative Programming Based on a Rewriting Logic. *Journal of Logic Programming,* 1(40):47–87, 1999.
19. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming,* 12(3):191–224, 1992.
20. R. Ramesh, I. Ramakrishnan, and D. Warren. Automata–Driven Indexing of Prolog Clauses. *Journal of Logic Programming,* 23(2):151–202, 1995.
21. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In *Proc. of ICLP,* pages 127–139, 1984.

$\mathcal{DDT}$: a Declarative Debugging Tool for Functional-Logic Languages*

Rafael Caballero and Mario Rodríguez-Artalejo

Dpto. de Sistemas Informáticos y Programación,
Universidad Complutense de Madrid
{rafa,mario}@sip.ucm.es

Abstract. We present a graphical tool for the declarative debugging of wrong answers in functional-logic languages. The tool, integrated in the system $\mathcal{TOY}$, can be used to navigate a computation tree corresponding to some erroneous computation. During the navigation the users can either follow a fixed strategy or move freely providing information about the validity of the nodes as they like. We show by means of examples how this flexibility can reduce both the number and the complexity of the questions that the user must consider w.r.t. the usual top-down navigation strategy. Moreover, the tool includes some extra features that can be used to automatically simplify the computation trees.

1 Introduction

The idea of *declarative debugging* was first proposed by E.Y. Shapiro [18] in the field of Logic Programming, and has been developed not only in this paradigm [11,7] but also in constraint-logic programming [20], functional programming [15,14,17], and functional-logic programming [5,4,6].

The overall idea in all the cases is the same, as pointed out by Lee Naish in [13]. The debugging starts when some erroneous computation, the so-called *initial symptom,* is found, and can be described as a two stages process:

- First, the declarative debugger builds a suitable *computation tree* for the initial symptom. Each node in this tree keeps the result of some subcomputation and can be associated to the fragment of code responsible for it. In particular, the root represents the (wrong) result of the main computation. The children of a given node must correspond to the intermediate subcomputations needed for obtaining the result at such node. This phase is automatically performed by the debugger; the user's assistance is only required to detect the initial symptom.

- Once the tree is obtained it is *navigated* looking for a *buggy* node, i.e. a node with an erroneous result whose children nodes have correct results. Such node is associated to a fragment of code that has produced an erroneous output from

* Work partially supported by the Spanish CYCIT (project TIC2002-01167 'MELODIAS').

© Springer-Verlag Berlin Heidelberg 2004

correct inputs, and will be pointed out by the debugger as one bug in the program. In order to check whether the nodes are correct or not, some external *oracle,* generally the user, is needed.

One of the main criticisms about the use of this technique w.r.t. other debugging methods such as abstract diagnosis [2,1], is the amount and complexity of the questions that the user must answer during the navigation phase. The problem has been considered in (constraint) logic programming [18,20], but has received little attention in functional and functional-logic programming, where most of the works are devoted to the definition of suitable computation trees and to devise mechanisms for their implementation. The declarative debuggers proposed in functional and functional-logic programs, to the best of our knowledge, perform a top-down traversal of the tree during the navigation phase. As we will see this is only satisfactory for trees containing a buggy node near the root; otherwise the number and complexity of the questions can make the debugging process unrealistic.

In this paper we 𝒟𝒟𝒯 (an acronym for 𝒟eclarative 𝒟ebugging 𝒯ool), a graphical declarative debugger included as part of the lazy functional-logic system 𝒯𝒪𝒴 [12]. However, the ideas and techniques presented here are also valid for the declarative debugging of wrong answers in other lazy functional-logic languages such as Curry [10] or lazy functional languages as Haskell [16].

𝒟𝒟𝒯 allows the user either to navigate freely the computation tree or to select one of the default strategies provided by the tool to guide the navigation. In the paper we show how these possibilities can be used to reduce both the number and the complexity of the questions that the user must consider during the debugging process. Moreover, 𝒟𝒟𝒯 also incorporates two techniques for simplifying the computation tree, the first one based on the notion of *entailment* proposed in [6], and the second one based on the use of another program as a (generally partial) correct specification of the intended program semantics. These two features can be used to determine the validity of some nodes of the computation tree in advance, thus simplifying the role of the user during the navigation phase.

The structure of the paper is as follows: next Section introduces some preliminary concepts and presents the general aspect of the tool. Section 3 explains by means of an example how the flexibility of the navigation in 𝒟𝒟𝒯 can be used to detect buggy nodes more easily. Section 4 discusses the strategies provided by the system, while Section 5 presents the two techniques used to simplify the computation tree mentioned above. Finally Section 6 concludes and points to some planned future work.

𝒟𝒟𝒯 is part of the distribution of the 𝒯𝒪𝒴 system, which is available at `http://titan.sip.ucm.es`.

2 Initial Concepts

As we said in the previous section, 𝒟𝒟𝒯 is integrated in the lazy FLP language 𝒯𝒪𝒴 [12]. In this section we first recall some basics about the language and illustrate them with an example. Then some basic notions and properties regarding computation trees are presented.

2.1 The $\mathcal{TOY}$ Language

Programs in $\mathcal{TOY}$ can include data type declarations, type alias, infix operators declarations, function type declarations, and defining rules for functions symbols. Before describing the structure of the defining rules we must define some initial notions such as expressions and patterns. A more detailed description of the syntax of the language can be found in [12].

The syntax of *partial expressions* $e \in Exp_\perp$ is $e ::= \perp \mid X \mid h \mid (e\ e')$ where X is a variable and h either a function symbol or a data constructor. Expressions of the form $(e\ e')$ stand for the application of expression e (acting as a function) to expression e' (acting as an argument). In the rest of the paper the notation $e\ e_1\ e_2 \ldots e_n$ (or even $e\ \overline{e}_n$) is used as a shorthand for $(\ldots((e\ e_1)\ e_2)\ldots)e_n)$. Similarly, the syntax of *partial patterns* $t \in Pat_\perp \subset Exp_\perp$ can be defined as $t ::= \perp \mid X \mid c\ t_1 \ldots t_m \mid f\ t_1 \ldots t_m$ where X represents a variable, c a data constructor of arity greater or equal to m, and f a function symbol of arity greater than m, while the t_i are partial patterns for all $1 \leq i \leq m$. We define the *approximation ordering* $\sqsubseteq$ as the least partial ordering over $Pat_\perp$ satisfying the two following properties:

- $\perp \sqsubseteq t$, for all $t \in Pat_\perp$.
- $h\ \overline{t}_m \sqsubseteq h\ \overline{s}_m$ if $h\ \overline{t}_m, h\ \overline{s}_m \in Pat_\perp$ and $t_i \sqsubseteq s_i$ for all $1 \leq i \leq m$.

Expressions and patterns without any occurrence of $\perp$ are called *total.*

The defining rules for a function f are composed of a *left-hand side,* a *right-hand side,* an optional *condition,* and some optional *local definitions*:

$$(R) \quad \underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \; = \; \underbrace{r}_{\text{right-hand side}} \; \Leftarrow \; \underbrace{C}_{\text{condition}} \quad where \quad \underbrace{LD}_{\text{local definitions}}$$

the condition has the form $C \equiv e_1 == e'_1, \ldots, e_k == e'_k$, while the local definitions are $LD \equiv \{s_1 \leftarrow a_1; \ \ldots; \ s_m \leftarrow a_m\}$, where e_i, e'_i, a_i and r are total expressions, the t_j, s_j are total patterns with no variable occurring more than once in different t_k, t_l or in different s_k, s_l, and no variable in s_i occurring in a_j for $1 \leq j < i \leq m$. Roughly, the intended meaning of a program rule like (R) is that a call to the function f can be reduced to r whenever the actual parameters match the patterns t_i, using the local definitions *LD,* and ensuring that the conditions $e_i == e'_i$ are satisfied. A condition $e == e'$ is satisfied by evaluating e and e' to some common total pattern.

A formal semantic calculus for $\mathcal{TOY}$ programs is described in [8,9], and has been adapted to the declarative debugging of wrong answers in [5,6]. As we proved in [5,6] a simplification of the proof trees in this semantic calculus can be employed as suitable computation trees for the declarative debugging of wrong answers in lazy functional-logic languages. The nodes of such computation trees

```
fib                  = [1, 1 | fibAux 1 1]
fibAux N M           = [N+M | fibAux N (N+M)]

goldenApprox         = (tail fib) ./. fib

infixr 20 ./.
[X | Xs]  ./. [Y | Ys] = [ X/Y | Xs ./. Ys]

tail [X|Xs]          = Xs

take 0 L             = []
take N [X|Xs]        = [X| take (N-1) Xs] <== N>0

main R               = true <== take 5 goldenApprox == R
```

Fig. 1. Approximating the Golden Ratio

are always *basic facts* of the form $f\ t_1 \ldots t_n \rightarrow t$, with f a function symbol of arity n and $t, t_1, \ldots t_n \in Pat_\perp$. The idea is that a basic fact $f\ t_1 \ldots t_n \rightarrow t$ can be proved w.r.t. some program *P* iff the pattern t approximates the value of the function call $(f\ t_1 \ldots t_n)$ in *P*. Since the value $\perp$ approximates all the values in our semantics, trivial basic facts of the form $f\ t_1 \ldots t_n \rightarrow \perp$ always can be proved. In [5,6] we also proved that any buggy node in these computation trees its associated with some *erroneous function rule* of the program, in fact with the function rule used to prove the basic fact labelling the node. The *intended model* of a program *P* is the set $\mathcal{I}$ of basic facts that the user expects to be provable w.r.t. *P*. A node of the computation tree is correct if its basic fact belongs to $\mathcal{I}$, and incorrect otherwise. Correct nodes are also called valid w.r.t. $\mathcal{I}$.

We will assume that every program *P* includes a special program rule *main* of the form $main\ X_1 \ldots X_k = true \Leftarrow\!= C$, where $\{X_1, \ldots, X_k\}$, $k \geq 0$, is the set of variables occurring in the condition *C*. The system will compute substitutions σ, called *answers,* of patterns for variables such that $dom(\sigma) \subseteq \{ X_1, \ldots, X_k\}$, meaning that the basic fact $(main\ X_1 \ldots X_k)\sigma \rightarrow true$ can be proved w.r.t. *P*. Notice that this notion of goal, suitable for this work, is compatible with actual goals in $\mathcal{TOY}$ which are of the form: $e_1 == e'_1, \ldots, e_k == e'_k$, simply by assuming that the goal is the condition of an implicit program rule for *main*.

2.2 An Example

Figure 1 shows a $\mathcal{TOY}$ program whose purpose is to approximate the number $\frac{1+\sqrt{5}}{2}$, known as the *golden ratio,* by using the Fibonacci sequence 1, 1, 2, 3, 5,…, where each term in the sequence (after the second) is the sum of the two that immediately precede it. If we call $fib(i)$ to the i-th term of this sequence, the following property holds:

$$\lim_{n\to\infty} \frac{fib(n+1)}{fib(n)} = \frac{1+\sqrt{5}}{2}$$

The program contains a function fib that represents an infinite list containing the Fibonacci sequence. This function uses an auxiliary function fibAux. Given two integer values X_0 and X_1, the function call fibAux X_0 X_1 is expected to compute the infinite list $[X_2, X_3, \ldots]$, such that $X_k = X_{k-2} + X_{k-1}$, for any $k \geq 2$. Function goldenApprox computes the infinite list [fib(2)/fib(1), fib(3)/fib(2), ...] using the infix operator ./., that returns the result of dividing two infinite lists term by term. The meaning of the rest of the functions should be clear from the context. Some basic facts included in the intended model $\mathcal{I}$ of the program are:

$$\begin{aligned}\mathcal{I} = \{ & \ldots, \text{fib} \to \bot, \ldots, \text{fib} \to [1 \mid \bot], \ldots, \text{fib} \to [1,1,2,3,5,8 \mid \bot], \ldots, \\ & \ldots, \text{fibAux 1 1} \to [2,3,5, \mid \bot], \ldots, \text{fibAux 10 20} \to [30,50,80,120 \mid \bot], \ldots, \\ & \ldots, \text{main } [1, 2, 1.5, 1.66, 1.6] \to \text{true}, \ldots \}\end{aligned}$$

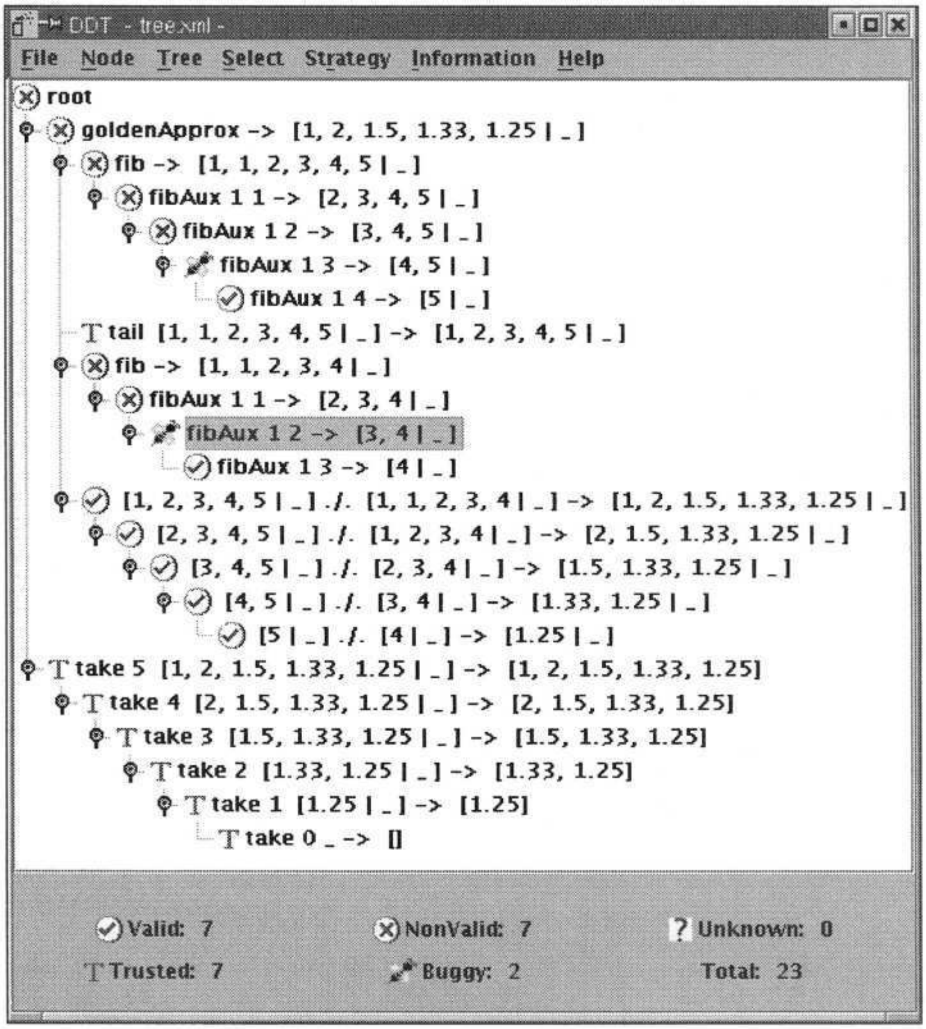
among others. In particular, the basic fact main [1, 2, 1.5, 1.66, 1.6] $\to$ true is expected since [1/1, 2/1, 3/2, 5/3, 8/5] = [1, 2, 1.5, 1.66, 1.6] (rounding to two decimals for simplicity) is the list of the fifth first approximations to the golden ratio using the Fibonacci sequence. However, the system computes the answer $\sigma = \{\text{R} \mapsto [1, 2, 1.5, 1.33, 1.25\ ]\}$. This is a *wrong answer,* since main R$\sigma \to$ true is not in the intended model of the program, and constitutes the initial symptom showing that there is some bug in the program.

The computation tree for this wrong computation can be seen in Figure 2, as displayed by $\mathcal{DDT}$. The root of the tree corresponds to the initial symptom and the children of each node correspond to the function calls needed for computing the basic fact at the node. For instance the root has two children:

$$\begin{array}{lll}(1) & \text{goldenApprox} & \to [1,2,1.5,1.33,1.25|\_\ ] \\ (2) & \text{take 5 } [1,2,1.5,1.33,1.25|\_\ ] & \to [1,2,1.5,1.33,1.25\ ]\end{array}$$

corresponding to the two function calls in the condition of main instantiated with the values used during the computation. The character _ in the display represents the symbol $\bot$, and stands in place of some value whose evaluation was not needed during the computation. The basic fact (2) is valid in the intended model, but the basic fact (1) is not, since the fourth and fifth members of the list at the right-hand side of the basic fact should be $\frac{fib(5)}{fib(4)} = 1.66$ and $\frac{fib(6)}{fib(5)} = 1.6$ respectively.

In the debugging session of the figure we have provided information about the validity of all the nodes, although usually this is not necessary as we will see in sections 3 and 4. At the bottom of the display $\mathcal{DDT}$ shows data about the amount of different kinds of nodes, including *unknown* nodes, corresponding to basic facts whose validity has not yet determined, and *trusted* nodes, which correspond to basic facts associated to trusted functions. In this example the user has decided that functions take and tail are trusted and hence all the basic facts corresponding to calls of these functions will be considered valid by the



```
(x) root
  (x) goldenApprox -> [1, 2, 1.5, 1.33, 1.25 | _]
    (x) fib -> [1, 1, 2, 3, 4, 5 | _]
      (x) fibAux 1 1 -> [2, 3, 4, 5 | _]
        (x) fibAux 1 2 -> [3, 4, 5 | _]
          fibAux 1 3 -> [4, 5 | _]
            ✓ fibAux 1 4 -> [5 | _]
    T tail [1, 1, 2, 3, 4, 5 | _] -> [1, 2, 3, 4, 5 | _]
    (x) fib -> [1, 1, 2, 3, 4 | _]
      (x) fibAux 1 1 -> [2, 3, 4 | _]
        fibAux 1 2 -> [3, 4 | _]
          ✓ fibAux 1 3 -> [4 | _]
    ✓ [1, 2, 3, 4, 5 | _] ./. [1, 1, 2, 3, 4 | _] -> [1, 2, 1.5, 1.33, 1.25 | _]
      ✓ [2, 3, 4, 5 | _] ./. [1, 2, 3, 4 | _] -> [2, 1.5, 1.33, 1.25 | _]
        ✓ [3, 4, 5 | _] ./. [2, 3, 4 | _] -> [1.5, 1.33, 1.25 | _]
          ✓ [4, 5 | _] ./. [3, 4 | _] -> [1.33, 1.25 | _]
            ✓ [5 | _] ./. [4 | _] -> [1.25 | _]
  T take 5 [1, 2, 1.5, 1.33, 1.25 | _] -> [1, 2, 1.5, 1.33, 1.25]
    T take 4 [2, 1.5, 1.33, 1.25 | _] -> [2, 1.5, 1.33, 1.25]
      T take 3 [1.5, 1.33, 1.25 | _] -> [1.5, 1.33, 1.25]
        T take 2 [1.33, 1.25 | _] -> [1.33, 1.25]
          T take 1 [1.25 | _] -> [1.25]
            T take 0 _ -> []

✓ Valid: 7          (x) NonValid: 7     ? Unknown: 0
T Trusted: 7        Buggy: 2            Total: 23
```

Fig. 2. Computation Tree for the program of Figure 1

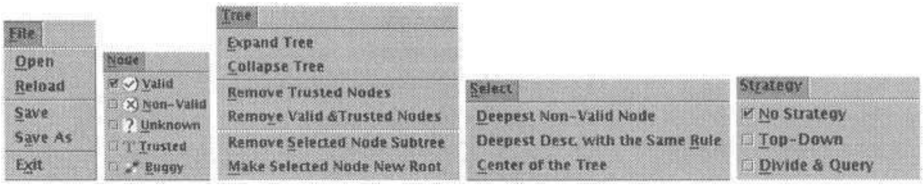


Fig. 3. Some menu options in 𝒟𝒟𝒯

debugger. The computation tree has two buggy nodes (one appears selected in the figure), both of them corresponding to the application of the single function rule for fibAux, which is therefore a wrong rule and will be pointed out by the debugger as the cause of the bug. The bug in this rule is in the first argument of the recursive call, that should be *M* instead of *N*. The correct rule is then: fibAux N M = [N+M | fibAux $\underline{\text{M}}$ (N+M)].

2.3 Computation Trees

Next we present some definitions and auxiliary notation about computation trees (shortly CT's). Notice that although some basic facts can occur repeatedly in a CT, any node can be identified by its path to the root. Given a CT *T* and a node $N \in T$, we will use the notation *root*(*T*) to represent the root of the tree, *subtree*(*T*, *N*) for the subtree of *T* whose root is *N*, and *children*(*T*, *N*) to represent a list with the children nodes of *N* in *T*. If *N* is valid w.r.t. the intended interpretation $\mathcal{I}$ we will write *valid*(*N*), assuming that $\mathcal{I}$ is clear from the context. Analogously *nonvalid*(*N*) will represent a non-valid node, while *buggy*(*T*, *N*) will mean that *nonvalid*(*N*) and $valid(N')$ for all $N' \in children(T, N)$. Finally, *buggy*(*T*) will indicate that there exists a node $N \in T$ such that *buggy*(*T*, *N*).

The number of nodes in a computation tree *T* will be represented as $|T|$. Let *N* be a node in *T* such that $N \neq root(T)$, and let *P* be the parent of *N* in *T*. Then the notation *T* – *N* represents the new tree obtained by removing *N* from *T* and letting the children of *N* become children of *P*. Hence, if $N_1, \ldots, N_{i-1}, N, N_{i+1}, \ldots N_m$ are the children of *P* in *T* and $N'_1, \ldots, N'_m$ are the children of *N* in *T*, the children of *P* in *T* – *N* will be $N_1, \ldots, N_{i-1}, N'_1$, $\ldots, N'_m, N_{i+1} \ldots N_m$. With these definitions two interesting properties of computation trees can be proved. Given a computation tree *T*:

P1 If $N \in T$ and *nonvalid*(*N*), then there is some node $N' \in subtree(T, N)$ such that $buggy(T, N')$ and the path from *N* to N' only has non-valid nodes.
P2 if $N \in T$ and *valid*(*N*), then *buggy*(*T*) iff *buggy*(*T* – *N*), and for every $N' \in T - N$ such that $buggy(T - N, N')$, $buggy(T, N')$ holds.

At several places in the rest of the paper we will use these properties for justifying the correctness of various $\mathcal{DDT}$ features.

3 Free Navigation

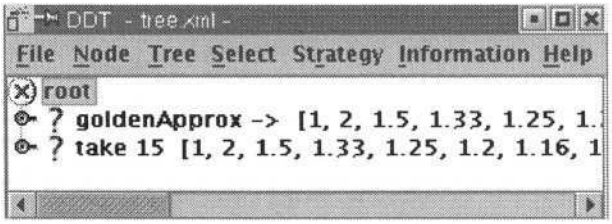
The $\mathcal{TOY}$ system includes currently two declarative debugging navigators: a textual top-down navigator similar to those of *Buddha* [17] and *Freja* [14], and the graphical navigator $\mathcal{DDT}$. This section shows by means of an example how the flexibility allowed by $\mathcal{DDT}$ can reduce the number and complexity of the nodes that the user must examine in comparison to the top-down navigators.

Let us consider again the program of Figure 1, but replacing the rule of the function main by main R = true <== take 15 goldenApprox == R. The answer computed by the system is again wrong and therefore the declarative debugger can be employed. By using the top-down navigator of 𝒯𝒪𝒴 we can obtain the following debugging session, where we have replaced part of the lists by dots for the sake of saving space in this presentation.

```
Consider the following facts:
1: goldenApprox → [1, 2, 1.5, 1.33, 1.25, 1.2, ... | _ ]
2: take 15 [1, 2, 1.5, 1.33, 1.25, 1.2, ... | _ ] → [1, 2, 1.5, 1.33, 1.25, 1.2, ...]
Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: fib → [1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 | _ ]
2: tail [1, 1, 2, 3, 4, 5, ... | _ ] → [1, 2, 3, 4, 5, ... | _ ]
3: fib → [1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 | _ ]
4: [1, 2, 3, 4, ... | _ ] ./. [1, 1, 2, 3, ... | _ ] → [1, 2, 1.5, 1.33, 1.25, ... | _ ]
Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: fibAux 1 1 → [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 | _ ]
Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: fibAux 1 2 → [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 | _ ]
.....(12 similar questions more involving fibAux )
Rule number 1 of the function fibAux is wrong.
```

We have omitted 12 additional questions about the validity of basic facts involving fibAux. Even assuming that the user knows the answer to these questions, the process can be a bit boring. With greater values in the argument of take the use of the top-down textual navigator will become unrealistic. It could be argued that the user should stop after a few questions about fibAux and try some easier goal, but in general is not always feasible to replace the goal which has produced an error symptom by a simpler one.

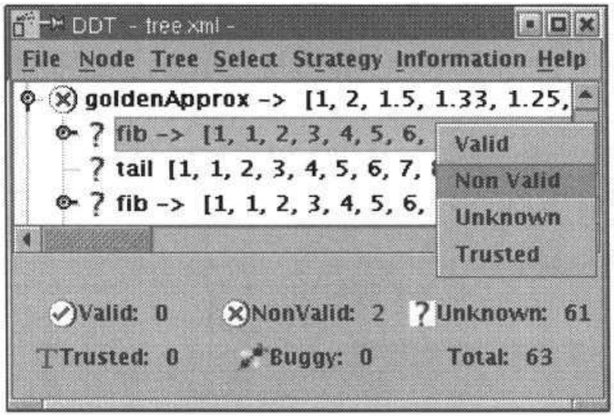
Let us examine now a possible debugging session for the same program using 𝒟𝒟𝒯. The display is not completely shown in some of the images due to the lack of space. In the initial state of the debugger only the first level of the tree, with the two children of the root, is expanded:



```
root
? goldenApprox -> [1, 2, 1.5, 1.33, 1.25, 1.
? take 15 [1, 2, 1.5, 1.33, 1.25, 1.2, 1.16, 1
```

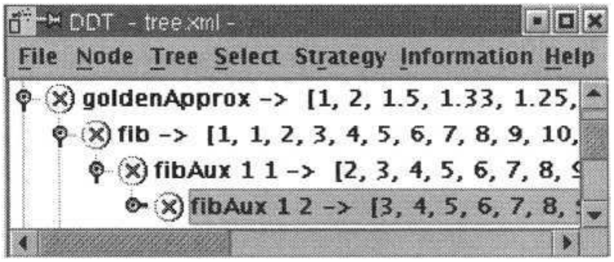
The user observes that the node corresponding to goldenApprox is not valid, changes its state to *nonvalid* and expands the node to examine its children. The state of a node can be changed in 𝒟𝒟𝒯 either by using the option menu

"Nodes" (see Figure 3), or by right-clicking over the node and selecting the state from the menu that appears, as shows the next image:



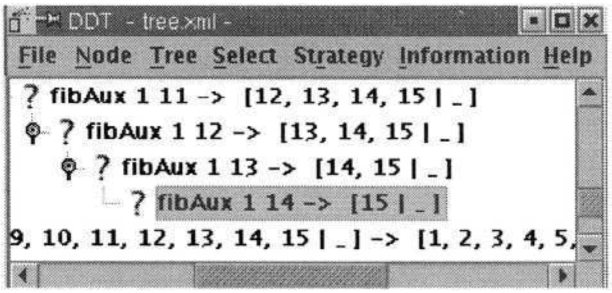
```
(x) goldenApprox -> [1, 2, 1.5, 1.33, 1.25,
  ? fib -> [1, 1, 2, 3, 4, 5, 6,
    ? tail [1, 1, 2, 3, 4, 5, 6, 7,
  ? fib -> [1, 1, 2, 3, 4, 5, 6,

Valid: 0      NonValid: 2    Unknown: 61
Trusted: 0    Buggy: 0       Total: 63
```

The next image shows the debugging session after descending in this way four levels in the tree:



```
(x) goldenApprox -> [1, 2, 1.5, 1.33, 1.25,
  (x) fib -> [1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    (x) fibAux 1 1 -> [2, 3, 4, 5, 6, 7, 8,
      (x) fibAux 1 2 -> [3, 4, 5, 6, 7, 8,
```

At this point the user, maybe getting bored, can check that the next nodes in the same path correspond to the function fibAux, and seem to be non-valid. Then the option "Deepest Descendant with the Same Rule" of the menu "Select" can be used. This option looks automatically for the deepest use of fibAux in the subtree whose root is selected. In this example, the system finds out and selects the node containing the basic fact **fibAux 1 14 → [15 | _]**:



```
? fibAux 1 11 -> [12, 13, 14, 15 | _]
 ? fibAux 1 12 -> [13, 14, 15 | _]
   ? fibAux 1 13 -> [14, 15 | _]
     ? fibAux 1 14 -> [15 | _]
9, 10, 11, 12, 13, 14, 15 | _] -> [1, 2, 3, 4, 5,
```

The user can check readily that this node is valid and change its state consequently. Moving now bottom-up, the user detects that the parent of the selected node contains the non-valid basic fact fibAux 1 13 → [14,15 | _] (the second element of the list should be 14 + 13 = 27 instead of 15) and changes its state accordingly. In this moment *DDT* detects that this node is buggy and shows an message reporting to the user that the only program rule for fibAux, used at the buggy node, has been detected as incorrect.

Due to the recursive structure of function definitions, the option "Deepest Descendant with the Same Rule" will often render a new current node *N* whose basic fact is smaller and thus simpler to analyze. Navigation can then proceed by moving down to *N*'s children in case that *N* is non-valid, and moving up to *N*'s ancestors otherwise. In both cases, property P1 guarantees that a buggy node can be eventually found, because *N* is known to have an invalid ancestor.

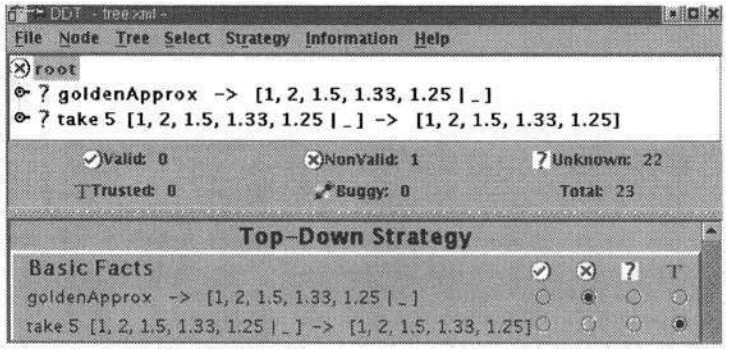
Of course the textual debugger could be enhanced with more sophisticated options, similar to those of *DDT* mentioned here. However we think that the graphical displaying provides a more general perspective of the CTs that allows, in many cases, a quicker detection of the buggy nodes.

4 Strategy-Guided Navigation: *Top-Down* Versus *Divide-and-Query*

Although free navigation can be used to reduce the number of nodes considered during the debugging process, *DDT* also includes the possibility of using a strategy-guided navigation and provides two possibilities: the *top-down* and the *divide-and-query* strategies.

The top-down strategy behaves essentially like the textual debugger presented in the previous section. The process starts with a computation tree whose root is considered non-valid. Then the children of the root are examined looking for some non-valid child. If such child is found the debugging continues examining its corresponding subtree. Otherwise all the children are valid and the root of the tree is pointed out as buggy, finishing the debugging.

The next display shows the starting point of the a debugging session using the top-down strategy, where the user has marked the first node as non-valid and the second one as trusted:



```
root
? goldenApprox -> [1, 2, 1.5, 1.33, 1.25 | _]
? take 5 [1, 2, 1.5, 1.33, 1.25 | _] -> [1, 2, 1.5, 1.33, 1.25]
```

Valid: 0 NonValid: 1 Unknown: 22
Trusted: 0 Buggy: 0 Total: 23

Top-Down Strategy

Basic Facts	✓	⊗	?	T
goldenApprox -> [1, 2, 1.5, 1.33, 1.25 | _]	○	●	○	○
take 5 [1, 2, 1.5, 1.33, 1.25 | _] -> [1, 2, 1.5, 1.33, 1.25]	○	○	○	●

	G_1	G_2	G_3	G_4	G_5	G_6
Nodes	403	6725	963	600	257	731
Top-Down	102 (106)	83 (165)	3 (3)	102 (204)	10 (33)	39 (71)
Divide-and-Query	10 (10)	13 (13)	10 (10)	10 (10)	8 (8)	7 (7)

Fig. 4. Number of steps and nodes examined with the two strategies

Notice that after each subsequent step the selected subtree has a smaller size and an invalid root. Hence, as a consequence of property P1, a buggy node is eventually reached.

The divide-and-query strategy was presented in [18] and has been included also in the system *TkCalypso* [20]. As in the top-down strategy, debugging starts with a computation tree whose root is not valid. The idea is to choose a node *N* such that the number of nodes inside and outside of the subtree rooted by *N* are the same. Although such node (called the *center* of the tree) does not exist in most of the cases, the system looks for the node that better approximates the condition. Then the user is queried about the validity of the basic fact labelling this node. If the node is non-valid its subtree will be considered at the next step. If it is valid then its subtree is deleted from the tree and debugging continues. The process ends when the subtree considered has been reduced to a single non-valid node.

Is easy to observe that, as in the top-down strategy, the number of nodes in the tree *T* considered is reduced after each step, and that *nonvalid*(*root*(*T*)) holds. To check that the strategy will find some buggy node we must examine the two actions that it can perform depending on the validity of the selected node *N*. First, if *nonvalid*(*N*), substituting the whole tree by *subtree*(*T,N*) is safe due to property P1. If *valid*(*N*) we must ensure that the deleting of *subtree*(*T,N*) will not delete all the buggy nodes. This holds again by Property P1, since the tree must have a buggy node *B* with a path of non-valid nodes from *root*(*T*) to *B*. Therefore *N* cannot be part of this path and *B* is not in *subtree*(*T, N*).

Since these strategies modify the structure of the tree, $\mathcal{DDT}$ includes options to save and load computation trees (see the options of the menu “File”). The files are stored in XML format. These options can be also used to restore a previous version of the debugging session if the user realizes after some steps that she or he made a mistake when providing information about the validity of the nodes, a situation that often arises.

Figure 4 shows a comparison of the number of steps and the number of nodes examined (between round brackets) during some debugging sessions with the two strategies. The first row of the table shows the total number of nodes of the computation tree considered in each example. Goal G_1 corresponds to our example of Figure 1 taking the 100 first approximations of the golden ratio. Goal G_2 uses a buggy program for computing prime numbers presented in [6]. G_3 uses a program with arithmetic in Peano’s representation. G_4 corresponds to a program for sorting numbers using a functional-logic programming technique called “lazy generate-and-test” in [8]. Goal G_5 uses a program for the symbolic derivation of expressions, while G_6 corresponds to a program implementing a queue.

The table shows a clear advantage of the divide-and-query strategy w.r.t. the top-down strategy. In general, the number of steps in a debugging session with a CT of size n is $O(log\ n)$ when using the divide and query and $O(n)$ when using the top-down strategy. However, these are worst-case estimations. The top-down strategy can behave more efficiently whenever there is a buggy node close to the root as it is the case for goal G_3.

The source code $\mathcal{DDT}$ consists of 2900 lines of Java code. We have used the Java language for two reasons: firstly, Java provides several libraries for designing graphical interfaces, and in particular some specific classes for representing trees graphically, and secondly because the Prolog system in which $\mathcal{TOY}$ is based, SICStus Prolog [19], includes an interface for interacting with Java.

5 Simplification of Computation Trees

Next we present two techniques incorporated in $\mathcal{DDT}$ that can provide automatically information about the validity of some nodes in the computation tree.

5.1 Entailment

In [5,6] we presented an *entailment* relation between basic facts based on the *approximation ordering* $\sqsubseteq$ defined in Section 2.1. A basic fact $f\ \bar{t}_n \to t$ *entails* another basic fact $f\ \bar{s}_n \to s$ (written as $f\ \bar{t}_n \to t\ \succeq\ f\ \bar{s}_n \to s$) iff there is some total substitution $\theta \in Subst$ such that $t_1\theta \sqsubseteq s_1, \ \ldots\ , t_n\theta \sqsubseteq s_n, s \sqsubseteq t\theta$.

Notice that the entailment property is covariant in the arguments but contravariant in the result. For instance, considering the basic facts of Figure 2, it can be easily proved that

$$\begin{array}{c} \mathsf{fib} \to [1,1,2,3,4,5 \mid \bot\,] \succeq \mathsf{fib} \to [1,1,2,3,4 \mid \bot\,] \\ \mathsf{fibAux}\ 1\ 2 \to [3,4,5 \mid \bot] \ \succeq \mathsf{fibAux}\ 1\ 2 \to [3,4 \mid \bot] \end{array}$$

with θ as the identity substitution in both cases. As shown in [6], entailment between basic facts is a decidable relation, and intended program models are closed under entailment, i.e. if $f\ \bar{t}_n \to t\ \succeq\ f\ \bar{s}_n \to s$ and $(f\ \bar{t}_n \to t) \in \mathcal{I}$ then $(f\ \bar{s}_n \to s) \in \mathcal{I}$. This means that if $f\ \bar{t}_n \to t$ is valid then $f\ \bar{s}_n \to s$ will be also valid, and conversely that if $f\ \bar{s}_n \to s$ is not valid then $f\ \bar{t}_n \to t$ is not valid.

$\mathcal{DDT}$ uses this property for changing automatically the state of some nodes when the user provides information about others. For instance when the node containing $\mathsf{fib} \to [1,1,2,3,4,5 \mid \bot\,]$ is marked as *valid,* the state of the node containing $\mathsf{fib} \to [1,1,2,3,4 \mid \bot\,]$ will be changed accordingly to *valid,* while marking the node corresponding to $\mathsf{fibAux}\ 1\ 2 \to [3,4 \mid \bot]$ as *nonvalid* will automatically change to *nonvalid* of the state of the node containing $\mathsf{fibAux}\ 1\ 2 \to [3,4,5 \mid \bot]$.

Every basic fact obviously entails itself. Therefore, any user-given change in the state of a node propagates automatically to all the other nodes containing the same basic fact.

5.2 Trusted Specifications

As explained in previous sections, $\mathcal{DDT}$ users can mark some CT nodes as *trusted* during a debugging session. All the nodes whose basic facts correspond to the function used at the trusted node are automatically considered as valid. A more automatic way of declaring trusted functions is by means of trusted specifications. This idea is not new and was introduced in the seminal work of E.Y. Shapiro [18].

We say that a $\mathcal{TOY}$ program *S* is a *trusted specification* if the user assumes as valid all the basic facts that can be derived from *S*. Let *P* be a buggy program, *T* a CT corresponding to some initial symptom for *P*, and S_P some trusted specification for *some* of the functions occurring in *P*. Then the following procedure can be used to provide some information about the validity of the nodes in *T*:

```
For each basic fact $\varphi$ : $f\ \bar{t}_n \rightarrow t$ labelling some node $N$ of $T$:
    If valid?($S_p$, $\varphi$) = yes          then delete $N$.
    If valid?($S_p$, $\varphi$) = no           then mark $N$ as non-valid.
    If valid?($S_p$, $\varphi$) = don't-know   then mark $N$ as unknown.
```

$\mathcal{DDT}$ incorporates an algorithm for computing *valid?*(S_p, φ) in a correct way. If the result is *yes* then φ can be derived from S_p and deleting *N* is safe because of Property P2. If the result is *no* then φ cannot be derived from S_p and marking *N* as non-valid is correct. Otherwise *N* is marked as unknown. There are two possible situations where the algorithm returns *don't-know*:

- If the function used in φ is not defined in S_p.
- If the time required for deciding if φ can be derived from S_p exceeds a certain time-out constant. This is done to avoid possible problems of non-termination, since the set of basic facts derivable from a given program is undecidable in general.

In each debugging session $\mathcal{DDT}$ asks the user whether this simplification should be performed. If the answer is affirmative the tool asks for the name of the $\mathcal{TOY}$ program which contains the trusted specification, and simplifies the tree before the navigation phase.

For instance, the program of Figure 5 is a trusted specification for the example program of the golden ratio, using a different method for generating the infinite list with the Fibonacci sequence. For saving space we don't include the definitions of take, tail, goldenApprox, ./. and main which are the same of the Figure 1.

We can imagine this program as the first, naive, solution for the problem of the golden ratio approximations programmed by the user. It works correctly, but the generation of Fibonacci numbers is quite inefficient. Then the example of Figure 1 could be an attempt of improving the efficiency of this program. After trying the new version the user could observe that it returns a different answer, and decide that the first naive version was more likely to be correct. Then the declarative debugger could be started using this first version of Figure 5 as a trusted specification. The simplification will delete 12 nodes of the computation

```
fibN 1          = 1
fibN 2          = 1
fibN N          = if N>2 then (fibN (N-1))+(fibN (N-2))

fib             = map fibN (from 0)

map F []        = []
map F [X|Xs]    = [F X | map F Xs]

from N          = [N | from N+1]
```

Fig. 5. A Trusted Specification for the program of Figure 1

tree and mark 3 nodes more as non valid, hence reducing the number of *unknown* nodes in the initial CT from 23 to only 8.

6 Conclusions

In this paper we have described the declarative debugging tool $\mathcal{DDT}$, which is part of the functional-logic system $\mathcal{TOY}$. In comparison to the traditional top-down declarative debuggers, $\mathcal{DDT}$ gives more support for avoiding the complexity of oracle questions. This can be achieved either by skillful free CT navigation, or by using a divide-and-query navigation strategy. Additionally, $\mathcal{DDT}$ also offers two useful techniques for simplifying the CT prior to navigation.

In contrast to other debugging tools (as e.g. the recent visual debugger for Mercury [3]) $\mathcal{DDT}$ is an off-line tool: the computation tree must be completely generated before it can be displayed and navigated. Unfortunately, complete CT generation causes a considerable overhead w.r.t. to the original computation which led to the debugging session, both in terms of time and space resources. Related works on the implementation of declarative debuggers for lazy functional languages [14,17] have proposed techniques for reducing the computational overhead caused by debugging. As far as we know, this kind of techniques have been worked out only for the top-down navigation strategy. They mainly rely on a lazy generation of the CT as demanded by navigation.

In spite of the computational overhead, we still believe that $\mathcal{DDT}$ offers better facilities for CT simplification and navigation, which means a crucial advantage in CTs with a large number of nodes, where top-down navigation produces too many (maybe complex) questions. As future work, we plan to revise the implementation of the $\mathcal{DDT}$ tool, looking for incremental CT simplification and navigation methods that can be made compatible with lazy CT generation.

Acknowledgements The authors are grateful to Wolfgang Lux and Francisco J. López-Fraguas for their useful comments about the paper, as well as to the anonymous referees for their constructive remarks.

References

1. M. Alpuente, F.J. Correa, and M. Falaschi. *A Debugging Scheme for Funcional Logic Programs.* Electronic Notes in Theoretical Computer Science. Vol. 64. Elsevier, 2002.
2. M. Comini, G. Levi, M.C. Meo y G. Vitello. *Abstract Diagnosis.* J. of Logic Programming 39, 43–93, 1999.
3. M. Cameron, M. García de la Banda, K. Marriott, and P. Moulder. *ViMer: A Visual Debugger for Mercury.* In Proc. PPDP03, ACM Press, 56–66, 2003.
4. R. Caballero and W. Lux. *Declarative Debugging of Encapsulated Search.* Electronic Notes in Theoretical Computer Science. Vol. 76. Elsevier, 2002.
5. R. Caballero, F.J. López-Fraguas, and M. Rodríguez-Artalejo. *Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs.* In Proc. FLOPS'01, Springer LNCS 2024:170–184, 2001.
6. R. Caballero and M. Rodríguez Artalejo. *A Declarative Debugging System for Lazy Functional Logic Programs.* Electronic Notes in Theoretical Computer Science. Vol. 64. Elsevier, 2002.
7. G. Ferrand. *Error Diagnosis in Logic Programming, an Adaptation of E.Y. Shapiro's Method.* The Journal of Logic Programming 4(3):177–198, 1987.
8. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez- Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic,* Journal of Logic Programming 40(1) 44-87, 1999.
9. J.C. González-Moreno, M.T. Hortalá-González y M. Rodríguez-Artalejo. *Polymorphic Types in Functional Logic Programming.* FLOPS'99 special issue of the Journal of Functional and Logic Programming, 2001. Available at: `http://danae.uni-muenster.de/lehre/kuchen/JFLP`
10. M.Hanus. *Curry: An Integrated Functional Logic Language.* Version 0.7.1, June 2000. Available at http://www.informatik.uni-kiel.de/curry/report.html.
11. J.W. Lloyd. *Declarative Error Diagnosis.* New Generation Computing 5(2):133–154, 1987.
12. F.J. López-Fraguas, and J. Sánchez-Hernández. *$\mathcal{TOY}$ a Multiparadigm Declarative System,* In Proc. RTA'99, LNCS 1631, Springer Verlag, 244-247, 1999.
13. L. Naish. *A Declarative Debugging Scheme.* Journal of Functional and Logic Programming, 1997-3.
14. H. Nilsson. *How to look busy while being lazy as ever: The implementation of a lazy functional debugger.* Journal of Functional Programming 11(6):629–671, 2001.
15. H. Nilsson and J. Sparud. *The Evaluation Dependence Tree as a basis for Lazy Functional Debugging.* Automated Software Engineering, 4(2):121–150, 1997.
16. S.L. Peyton Jones (ed.), J. Hughes (ed.), et al. *Report* on *the programming language Haskell 98: a non-strict, purely functional language.* Available at `http://www.haskell.org/onlinereport/`, 2002.
17. B. Pope and L. Naish, *Practical Aspects of Declarative Debugging in Haskell 98,* In Proc. PPDP03, ACM Press, 230–240, 2003.
18. E.Y.Shapiro. *Algorithmic Program Debugging.* The MIT Press, Cambridge, 1982.
19. SICStus Prolog homepage: `http://www.sics.se/sicstus/`.
20. A. Tessier and G. Ferrand. *Declarative Diagnosis in the CLP Scheme.* In P. Deransart, M. Hermenegildo and J. Małuszynski (Eds.), Analysis and Visualization Tools for Constraint Programming, Chapter 5,151–174. Springer LNCS 1870, 2000.

LIX: an Effective Self-applicable Partial Evaluator for Prolog

Stephen-John Craig and Michael Leuschel

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
sjc02r,mal@ecs.soton.ac.uk

Abstract. This paper presents a self-applicable partial evaluator for a considerable subset of full Prolog. The partial evaluator is shown to achieve non-trivial specialisation and be effectively self-applied. The attempts to self-apply partial evaluators for logic programs have, of yet, not been all that successful. Compared to earlier attempts, our LIX system is practically usable in terms of efficiency and can handle natural logic programming examples with partially static data structures, built-ins, side-effects, and some higher-order and meta-level features such as `call` and `findall`. The LIX system is derived from the development of the LOGEN compiler generator system. It achieves a similar kind of efficiency and specialisation, but can be used for other applications. Notably, we show first attempts at using the system for deforestation and tupling in an offline fashion. We will demonstrate that, contrary to earlier beliefs, declarativeness and the use of the ground representation is not the best way to achieve self-applicable partial evaluators. **Keywords:** Partial Evaluation, Self-application, Logic Programming, Partial Deduction, Deforestation, Tupling.

1 Introduction and Summary

Partial evaluation has received considerable attention over the past decade both in functional (e.g. [16]), imperative (e.g. [2]) and logic programming (e.g. [9, 18, 25]). In the context of pure logic programs, partial evaluation is often referred to as *partial deduction,* the term partial evaluation being reserved for the treatment of impure logic programs. We will adhere to this convention in this paper.

Guided by the *Futamura projections* (see e.g. [16]) a lot of effort, especially in the functional partial evaluation community, has been put into making systems self-applicable. A partial evaluation or deduction system is called *self-applicable* if it is able to effectively[1] specialise itself. The practical interests of such a capability are manifold. The most well-known are related to the second and third *Futamura projections* [7]. The first Futamura projection consists of specialising an *interpreter* for a particular *object program,* thereby producing a specialised

[1] This implies some efficiency considerations, e.g. the system has to terminate within reasonable time constrains, using an appropriate amount of memory.

© Springer-Verlag Berlin Heidelberg 2004

version of the interpreter which can be seen as a *compiled* version of the object program. If the partial evaluator is self-applicable then one can specialise the partial evaluator for performing the first Futamura projection, thereby obtaining a *compiler* for the interpreter under consideration. This process is called the second Futamura projection. The third Futamura projection now consists of specialising the partial evaluator to perform the second Futamura projection. By this process we obtain a *compiler generator* (*cogen* for short).

History of Self-application for Logic Programming Not surprisingly, writing an effectively self-applicable specialiser is a non-trivial task — the more features one uses in writing the specialiser the more complex the specialisation process becomes, as the specialiser then has to handle these features as well. For a long time it was believed that in order to develop a self-applicable specialiser for logic programs one needed to write a clean, pure and simple specialiser. In practice, this meant using few (or even no) impure features in the implementation of the specialiser. For this the *ground representation* [14] was believed to be key, in which variables of the source program are represented by ground constants within the specialiser. Indeed, the ground representation allows one to freely manipulate the source program to be specialised in a declarative manner. The *non-ground representation,* where source-level variables are represented as variables in the program specialiser, can suffer from semantical problems [23] and requires some non-declarative features (such as `findall/3`) in order to perform the specialisation.

Some early[2] attempts at self-application [6] used the non-ground representation, but the self-applying led to incorrect results as the specialiser did not properly handle the non-declarative constructs that were employed in its implementation . Other specialisers like MIXTUS [27], PADDY [26] and ECCE [22] use the non-ground representation, but none of them are able to effectively specialise themselves.

The ground representation approach towards self-application was pursued in [3], [19], [24], and [4, 12, 13] leading to some self-applicable specialisers:

- SAGE [12], a self-applicable partial evaluator for Gödel. While the speedups obtained by self-application are respectable, the process takes a very long time (several hours) and the obtained specialised specialisers are still extremely slow. This is probably due to the explicit unification algorithm required by the ground representation. To effectively specialise this much more powerful specialisation techniques would be required to obtain reasonably efficient specialisers. Similar performance problems were encountered in the earlier work [3].
- LOGIMIX [16, 24], a self-applicable partial evaluator for a subset of Prolog, including *if-then-else,* side-effects and some built-in's. LOGIMIX uses a meta-interpreter (sometimes called *InstanceDemo*) for the ground representation in which the goals are "lifted" to the non-ground representation for resolution. This avoids the use of an explicit unification algorithm, at the expense

[2] A problem mentioned in [3], see also [24, 19].

of some power[3]. Unfortunately, LOGIMIX gives only modest speedups (when compared to results for functional programming languages, see [24]), but it was probably the first practical self-applicable specialiser for a logic programming language.

Given the problem in developing a truly practical self-applicable specialiser for logic programs, the attention shifted to the *cogen approach* [15]: instead of trying to write a partial evaluation system which is neither too inefficient nor too difficult to self-apply, one simply writes a compiler generator directly. Indeed, the actual creation of the cogen according to the third Futamura projection is in general not of much interest to users since the cogen can be generated once and for all when a specialiser is given. This approach was pursued in [17, 21] leading to the LOGEN system, which can produce specialised specialisers much more efficiently than any of the self-applicable systems mentioned above. The resulting specialisers themselves are also much more efficient.

A New Attempt at Self-application In a sense the cogen approach has closed the practical debate on self-application for logic programming languages: one can get most of the benefits of self-application without writing a self-applicable specialiser. Still, there is the question of academic curiosity: is it really impossible to derive the cogen written by hand in [17, 21] by self-application? Also, having a self-applicable specialiser is sometimes more flexible as we may generate different cogen's for different purposes (such as one with debugging enabled). One may produce more or less optimised cogen's by tweaking the specialisation process, and better control the tradeoff between specialisation time and quality of the optimised code. Maybe there are other situations where a self-applicable partial evaluation system is preferrable to a cogen: Glück's specialiser projections [10] and the semantic modifiers of Abramov and Glück [1] may be such a setting.

This paper aims to answer some of these questions. Indeed, after the development of LOGEN we realised that one could translate LOGEN into a classical partial evaluator without too much difficulty. Furthermore, using new annotation facilities developed for the second version of LOGEN [21], one can actually make this partial evaluator (henceforth called LIX) self-applicable. By self-applying LIX we obtain generating extensions via the second Futamura projection which are very similar to the ones produced by LOGEN and the cogen obtained via the third Futamura projection also has lot of similarities to the code of LOGEN. The performance of this self-applicable partial evaluator is (after self-application) on par with LOGEN, and is thus much faster than any of the previous self-applicable logic programming specialisers. In the paper we also show some potential practical applications of this self-applicable specialiser.

The code of the specialiser itself is also surprisingly simple, but uses a few non-declarative features and does not use the ground representation. So, contrary to earlier belief, declarativeness and the ground representation were not the best way to climb the mountain of self-application. Indeed, the use of the non-ground representation makes our partial evaluator much more efficient and avoids all

[3] This idea was first used by Gallagher in [8, 9] and then later in [20] to write a declarative meta-interpreter for integrity checking in databases.

the complications related to specialising an explicit unification algorithm. The only drawback is that to safely deal with the non-ground representation, our partial evaluator needs to use some non-declarative features such as `findall`, and hence also has to be able to specialise them. Fortunately, this turned out to be less of a problem than anticipated.

In summary, Futamura's insight was that a cogen could be derived by a self-applicable specialiser. The insight in [15] was that a cogen is just a simple extension of a binding-time analysis, while our insight is that an effective self-applicable specialiser can be derived by transforming a cogen.

2 The Partial Evaluator

LOGEN and LIX are both offline partial evaluators. An offline partial evaluator works on an annotated version of the source program, these annotations are used to guide the specialisation process. There are two kinds of annotations:

- **filter declarations**, indicating whether arguments to predicates are **static** or **dynamic.** This influences the global control.
- **clause annotations**, indicating how every call in the body should be treated during unfolding. These influence the local control.

2.1 The Basic Annotations

A common annotation format is used for both the LIX and LOGEN systems. Each call in the program is annotated using `logen/2` and arguments are annotated using filter declarations. The head of a clause is annotated with an identifier. The format of the annotations is demonstrated in the following append example:

```
:- filter append(static,dynamic,dynamic).
logen(app, append([],L,L)).
logen(app, append([H|T], L, [H|T1])) :- logen(unfold, append(T,L,T1)).
```

The first argument to append has been marked as **static**, it will be known at specialisation time, and the other arguments have been marked **dynamic**. The recursive call to append is annotated for unfolding, the first argument is known thus guaranteeing termination at specialisation time. Some of the basic annotations are:

- **unfold** for reducible predicates, they will be unravelled during specialisation.
- **memo** for non-reducible predicates, they will be added to the memoisation table and replaced with a generalised residual predicate.
- **call** fully static call will be made during specialisation.
- **rescall** the call will be kept and will appear in the final specialised code.

2.2 The Source Code

We now present the main body of the LIX partial evaluator[4] . An atom A is specialised by calling `lix(`A`, Res)`. The `memo/2` and `memo_table/2` predicates

[4] The LIX system can be downloaded from:
`http://www.ecs.soton.ac.uk/~sjc02r/lix/lix.html`.

return in their second argument a call to a new specialised predicate where static arguments have been removed and dynamic ones generalised. Generalisation and filtering are performed by `generalise_and_filter/3`. It returns in its second argument the generalised call (to be unfolded) and in its third argument the call to the specialised predicate. It uses the annotations defined by `filter/2` to perform its task. The predicate `gensym/2` is used to create unique names for the specialised predicates. The predicate `unfold/2` computes the bodies of specialised predicates. A call annotated as `memo` is replaced by a call to the specialised version. If it does not already exist it is created by `memo/2`. A call annotated as `unfold` is further unfolded; a call annotated as **call** is completely evaluated; finally, a call annotated as `rescall` is added to the residual code without modification (for built-ins that cannot be evaluated or code that is defined elsewhere). All clauses defining the new predicate are collected using `findall/3` and pretty printed.

Note the use of the global side effect, `assert(memo_table(GCall, RCall))`, to maintain the list of previously specialised calls. The `univ` operator =.. can be used either to decompose a term into a list containing its functor and arguments or else construct a term from such a list. For example the term `f(X,Y)` can be deconstructed into `[f,X,Y]`.

To save space the definition of `pretty_print_clauses/1` is not given.

```
:- dynamic memo_table/2,flag/2.
lix(CallToSpecialise, ResidualCall) :-
        print(':- dynamic flag/2, memo_table/2.\n'),
        print(':- use_module(library(lists)).\n'),
        memo(CallToSpecialise, ResidualCall).
memo(Call, Residual) :-
        (   memo_table(Call, Residual) -> true
        ;   generalise_and_filter(Call, GenCall, ResidualPred),
            assert(memo_table(GenCall,ResidualPred)),
            findall((ResidualPred:-Body), unfold(GenCall,Body), Clauses),
            format('/*~k=~k*/~n', [ResidualPred,GenCall]),
            pretty_print_clauses(Clauses), memo_table(Call, Residual)
        ).
unfold(Head, Residual) :- ann_clause(_, Head, Body),pe(Body, Residual).
pe(true, true).
pe((A,B), (ResA,ResB)) :- pe(A, ResA), pe(B, ResB).
pe(logen(call,Call), true) :- call(Call).
pe(logen(rescall,Call), Call).
pe(logen(memo,Call), Residual) :- memo(Call, Residual).
pe(logen(unfold,Call), Residual) :- unfold(Call, Residual).
generalise_and_filter(Call, GenCall, ResidualPred) :-
        filter(Call, Filter), Call=..[Head|Args],
        gen_filter(Filter, Args, GenArgs, ResArgs),
        GenCall=..[Head|GenArgs],
        gensym(Head, ResHead), ResidualPred =..[ResHead|ResArgs].
gen_filter([], [], [], []).
gen_filter([static|A], [B|C], [B|D], E) :- gen_filter(A, C, D, E).
gen_filter([dynamic|A], [_|B], [C|D], [C|E]) :- gen_filter(A, B, D, E).
```

```
/* code for unique symbol generation, using dynamic flag/2 */
oldvalue(Sym, Value) :- flag(gensym(Sym), Value), !.
oldvalue(_, 0).
set_flag(Sym, Value) :-
   nonvar(Sym), retract(flag(Sym,_)), !, asserta(flag(Sym,Value)).
set_flag(Sym, Value) :- nonvar(Sym), asserta(flag(Sym,Value)).
gensym(Head, ResidualHead) :-
   var(ResidualHead), atom(Head), oldvalue(Head, OldVal),
   NewVal is OldVal+1, set_flag(gensym(Head), NewVal),
   name(A__, "__"), string_concat(Head, A__, Head__),
   string_concat(Head__, NewVal, ResidualHead).
append([], A, A).
append([A|B], C, [A|D]) :-  append(B, C, D).
string_concat(A, B, C) :- name(A, D), name(B, E),
                          append(D, E, F), name(C, F).
/* Clause Database: automatically created from annotated file */
ann_clause(1, app([],A,A), true).
ann_clause(2, app([A|B],C,[A|D]), logen(memo,app(B,C,D))).
filter(app(_,_,_), [dynamic,static,dynamic]).
```

2.3 Deriving LIX from LOGEN

The LIX partial evaluator was created by transforming the LOGEN compiler generator. The basic insight was that it is possible to create a classical partial evaluator that when specialised would produce similar generating extensions. Let us compare a small extract of code from both LOGEN and LIX, dealing with the **call** and **rescall** annotations:

`body(logen(call,Call),Call,true).`	`pe(logen(call,Call), true) :- call(Call).`
`body(logen(rescall,Call),true,Call).`	`pe(logen(rescall,Call), Call) :- true.`
LOGEN	LIX

The `body` predicate is explained in detail in [21]. Basically, the first argument is an annotated call, the second argument is the code that will appear in the generating extension and the third argument denotes the specialised code. We can see that the middle argument from `body/3` in LOGEN has been transformed into a call in the LIX version. This call is annotated as **residual** for self-application, and will hence appear in the generating extension produced by self-application. A more detailed comparison of the generating extensions and the produced cogen can be found in Section 5.

2.4 Specialised Code

To specialise code we use the `lix/2` entry point. Calling `lix(app(A,[b],C),Res)` specialises the append predicate to append `[b]` to the end of a list:

```
app__1([], [b]).
app__1([A|B], [A|C]) :- app__1(B, C).
```

The generation of the above code took 0.318 ms[5]. This is a very simple example to demonstrate the partial evaluator. The specialisation of a non-trivial Vanilla debugging interpreter and other examples can be found on the `lix` homepage[6].

3 Towards Self-application

We have presented the main body of the code for the LIX system. For a partial evaluator to be self-applicable it must be able to effectively handle all of the features it uses. The system we have presented so far uses a few non-declarative features and does not use the ground representation. In this section will shall introduce the required extension to make LIX self-applicable.

3.1 The Nonvar Binding-Type

We now present a new feature derived from LOGEN which is useful when specialising interpreters. This annotation will be the key for effective self-application.

In addition to marking arguments to predicates as **static** or **dynamic**, it is also possible to use the binding-type **nonvar.** This means that this argument is not a free variable and will have at least a top-level function symbol, but it is not necessarily ground. For example `f(X)`, `f(a)` and `f` are all nonvar but the variable `X` is not. During generalisation, the top level function symbol is kept but all its sub-arguments are replaced by fresh variables. For filtering, every sub-argument becomes a new argument of the residual predicate.

A small example will help to illustrate this annotation:

```
:- filter p(nonvar).
p(f(X)) :- p(g(a)).                 p(g(X)) :- p(h(X)).
p(h(a)).                            p(h(X)) :- p(f(X)).
```

If we mark no calls as unfoldable, we get the following specialised program for the call `p(f(Z))`:

```
%%% entry point:  p(f(Z)) :- p__0(Z)
p__0(B) :-   p__1(a).               p__1(B) :-   p__2(B).
p__2(a).                            p__2(B) :-  p__0(B).
```

If we mark everything except the last call as unfoldable we obtain:

```
p__0(B).
p__0(B) :- p__0(a).
```

The `gen_filter/2` predicate in the LIX source code is extended to handle the **nonvar** annotation:

```
gen_filter([nonvar|A], [B|C], [D|E], F) :-
        B=..[G|H], length(H, I), length(J, I),
        D=..[G|J], gen_filter(A, C, E, K), append(J, K, F).
```

[5] Benchmarks performed using SICStus Prolog 3.10.1 for Linux on a Pentium 2.4GHz with 512MB RAM.

[6] `http://www.ecs.soton.ac.uk/~sjc02r/lix/lix.html`

3.2 Treatment of findall

In LIX `findall` is used to collect the clauses when unfolding a call; hence we have to be able to treat this feature during specialisation.

Handling `findall` is actually not much different from handling negation in [21]. There is a static version (`findall`), in which the call is executed at specialisation time, and a dynamic version (`resfindall`), where it is executed at runtime. In both cases, the second argument must be annotated. For `resfindall`, much like `resnot` in [21], the annotated argument should be deterministic and should not fail (which can be ensured by wrapping the argument into a `hide_nf` annotation, see [21]). Also, if a `findall` is marked as static then the call should be sufficiently instantiated to fully determine the list of solutions. The following code is used in the subsequent examples:

```
:- filter all_p(static,dynamic).
all_p(X,Y) :- findall(X,p(X),Y).
:- filter p(static).
p(a). p(b).
```

If the `findall` is marked as residual and we `memo` `p(X)` inside it then the specialised program for `all_p(a,Y)` is:

```
all_p__0(A) :- findall(a,p__1,A).
p__1.
```

If we mark `p(X)` as `unfold` we get:

```
all_p__0(A) :- findall(a,true,A).
```

For self-application, only `resfindall` is actually required. The `pe/2` predicate is extended as follows:

```
pe(resfindall(Vars,G2,Sols), findall(Vars,VS2,Sols)) :-
    pe(G2,VS2).
```

3.3 Treatment of if

In the LIX code an `if-then-else` is used in `memo/2`. In this case the `if` is dynamic, the body of the conditional will be computed, along with those of the branches and an`if` statement will be constructed in the residual code. LIX is also extended to handle a static `if` which is performed at specialisation time.

```
pe(resif(A,B,C), (D->E;F)) :- pe(A, D), pe(B, E), pe(C, F).
pe(if(A,B,C), D) :- (pe(A, _) -> pe(B, D) ; pe(C, D)).
```

3.4 Handling the Cut

This is actually very easy to do, as with careful annotation the cut can be treated as a normal built-in call. The cut must be annotated using **call,** where it is performed at specialisation time, or **rescall,** where it is included in the residual code. It is up to the annotator to ensure that this is sound, i.e. LIX assumes that:

- if a cut marked **call** is reached during specialisation then the calls to the left of the cut will never fail at runtime.
- if a cut is marked as **rescall** within a predicate p, then no calls to p are unfolded.

These conditions are sufficient to handle the cut in a sound, but still useful manner.

4 Self-application

Using the features introduced in Section 3 and the basic annotations from Section 2.1, LIX can be successfully annotated for self-application. Self-application allows us to achieve the Futamura projections mentioned in the introduction.

4.1 Generating Extensions

In Section 2.4 we specialised `app/3` for the call `app(A,[b],C)`. If a partial evaluator is fully self-applicable then it can specialise itself for performing a particular specialisation, producing a *generating extension.* This process is the second Futamura projection. When specialising an interpreter the generating extension is a compiler.

A generating extension for the append predicate can be created by calling `lix(lix(app(A,B,C),R),R1)`, creating a specialised specialiser for append.

```
/*Generated by Lix*/
:- dynamic flag/2, memo_table/2.
/* oldvalue__1(_5557,_5586) = oldvalue(_5557,_5586) */
oldvalue__1(A, B) :- flag(gensym(A), B), !.
oldvalue__1(_, 0).

/* set_flag__1(_7128,_7153) = set_flag(gensym(_7128),_7153) */
set_flag__1(A, B) :- retract(flag(gensym(A),_)), !,
                     asserta(flag(gensym(A),B)).
set_flag__1(A, B) :- asserta(flag(gensym(A),B)).

/* gensym__1(_4392) = gensym(app,_4392) */
gensym__1(A) :- var(A), oldvalue__1(app, B),
                C is B+1,set_flag__1(app, C),
                name(C, D), name(A, [97,112,112,95,95|D]).
/* Printing and Flatten Clauses removed to save space */
/* unfold__1(_6925,_6927,_6929,_6956) =
                                    unfold(app(_6925,_6927,_6929),_6956) */
unfold__1([], A, A, true).
unfold__1([A|B], C, [A|D], E) :- memo__1(B, C, D, E).

/* memo__1(_2453,_2455,_2457,_2484) =
                                      memo(app(_2453,_2455,_2457),_2484) */
memo__1(A, B, C, D) :-
        (   memo_table(app(A,B,C), D) -> true
        ;   gensym__1(E), F=..[E,G,H],
            assert(memo_table(app(G,B,H),F)),
            findall((F:-I), unfold__1(G,B,H,I), J),
            format('/*~k=~k*/~n', [F,app(G,B,H)]),
            pretty_print_clauses__1(J),
            memo_table(app(A,B,C), D)
        ).
/* lix__1(_1288,_1290,_1292,_1319) = lix(app(_1288,_1290,_1292),_1319) */
lix__1(A, B, C, D) :- memo__1(A, B, C, D).
```

This is almost entirely equivalent to the proposed specialised unfolders in [17, 21]. It is actually slightly better as it will do flow analysis and only generate unfolders for those predicates that are reachable from the query to be specialised. Note the `gensym/2` predicate is specialised to produce only symbols of the form `app__N`. Generation of the above took 3.3 ms.

The generating extension for append can be used to specialise the append predicate for different sets of static data. Calling the generating extension with `lix--1(A,[b],C,R)` creates the same specialised version of the append predicate as in seciton 2.4:

```
app__1([], [b]).
app__1([A|B], [A|C]) :- app__1(B, C).
```

However using the generating extension is faster, for this small example 0.212 ms instead of 0.318 ms. Using a larger benchmark, unfolding (as opposed to memoising) the append predicate for a 10,000 item list produces more dramatic results. To generate the same code the generating extension takes 40 ms compared to 990 ms for LIX. The overhead of creating the generating extension for the larger benchmark is only 10 ms. Generating extensions can be very efficient when a program is to be specialised multiple times with different static data.

4.2 Lix Compiler Generator

The third Futamura projection is realised by specialising the partial evaluator to perform the second Futamura projection. By this process we obtain a *compiler generator* (*cogen* for short), a program that transforms interpreters into compilers. By specialising LIX to create generating extensions we create LIX-COGEN, a self-applied compiler generator. This can be achieved with the query `lix(lix(lix(Call,R),R1),R2)`. An extract from the produced code is now given:

```
/*unfold__13(Annotation, Generated Code, Specialisation Time) */
unfold__13(true, true, true).
unfold__13((A,B), (C,D), (E,F)) :-
        unfold__13(A, C, E),
        unfold__13(B, D, F).
unfold__13(logen(call,A), true, call(A)).
unfold__13(logen(rescall,A), A, true).
...
```

This has basically re-generated the 3-level cogen described in [17, 21]. In the **rescall** annotation for example, the call *(A)* will become part of the residual program, and nothing *(true)* is performed at specialisation time.

This code extract demonstrates the importance of the **nonvar** annotation. The annoated version of the original `unfold/2` is now shown.

```
:- filter unfold(nonvar,dynamic):unfold.
logen(unfold, unfold(X,Code)) :-
        logen(unfold, ann_clause(_,X,B)),
        logen(unfold, pe(B,Code)).
```

Without the **nonvar** annotation the first argument would be annotated **dynamic** as the arguments to the call being unfold may not be known at specialisation time. This would produce a single generic unfolder predicate much like the original `lix`. The **nonvar** annotation is needed to generate the specialised unfolders.

The generated LIX-COGEN will transform an annotated program directly into a generating extension, like the one found in section 4.1. However LIX-COGEN is faster: to create the same generating extension from an input program of 1,000 predicates LIX-COGEN takes only 3.9 s compared to 100.9 s for LIX.

5 Comparison

Logen The LOGEN system is an offline partial evaluation system using the cogen approach. Instead of using self-application to achieve the third Futamura projection, the LOGEN compiler generator is hand written. LIX was derived from LOGEN by rewriting it into a classical partial evaluation system. Using the second Futamura projection and self-applying LIX produces almost identical generating extensions to those produced by LOGEN (and both systems can in principle treat full Prolog). Apart from the predicate names the specialised unfolders generated by the two systems are the same:

`app__u([],A,A,true).`	`unfold__1([], A, A, true).`
`app__u([A|B],C,[A|D],E) :-`	`unfold__1([A|B], C, [A|D], E) :-`
`   app__m(B,C,D,E).`	`  memo__1(B, C, D, E).`
`...`	`...`
LOGEN Generating Extension	LIX-COGEN Generating Extension

While LOGEN is a hand written compiler generator, LIX must be self-applied to produce the same result as in Section 4.2. If we compare the LOGEN source code to the output in Section 4.2 we find very similar clauses in the form of `body/3` (note however, that the order of the last two arguments is reversed).

```
body(true,true,true).                       | unfold__13(true, true, true).
body((G,GS),(G1,GS1),(V,VS)) :-             | unfold__13((A,B), (C,D), (E,F)) :-
  body(G,G1,V),                             |       unfold__13(A, C, E),
  body(GS,GS1,VS).                          |       unfold__13(B, D, F).
body(logen(call,Call),Call,true).           | unfold__13(logen(call,A), true, call(A)).
body(logen(rescall,Call),true,Call).        | unfold__13(logen(rescall,A), A, true).
LOGEN                                       | LIX-COGEN
```

Unlike LIX, LOGEN does not perform flow analysis. It produces unfolders for all predicates in the program, regardless of whether or not they are reachable.

Logimix and Sage Comparisons of the initial *cogen* with other systems such as LOGIMIX, PADDY, and SP can be found in [17]. In essence, LOGEN was was 50 times faster than LOGMIX at producing the generating extensions (0.02 s instead of 1.10 s or 0.02 s instead of 0.98 s) and the specialisation times were about 2 times faster. It is likely that a similar relationship holds between LIX and LOGIMIX given that LIX and LOGEN have similar performance. Unfortunately LOGIMIX no longer runs on current versions of SICStus Prolog and we were thus unable to compare LIX and LOGIMIX directly. Similarly, Gödel no longer runs on current versions of SICStus Prolog, and hence we could not produce any timings for SAGE. However, timings from [12] indicate that the use of the ground representation means that SAGE is far too slow to be practical. Indeed, generating the compiler generator took about 100 hours and creating a generating extension for the examples in [12] took at least 7.9 hours. The speedups from using the generating extension instead of the partial evaluator range from 2.7 to 3.6 but the execution times for the generating extensions still ranged from 113 s to 447 s.

Multi-level Languages Our annotation scheme (for both LIX and LOGEN) can be viewed as a two-level language. Contrary to MetaML [28] our annotations are not part of the programming language itself (as we treat classical Prolog). It would be interesting to investigate to what extent one could extend our scheme for multiple levels of specialisation [11].

6 New Applications

Apart from the academic satisfaction of building a self-applicable specialiser, we think that there will be practical applications as well. We elaborate on a few in this section.

Several Versions of the Cogen In the development of new annotation and specialisation techniques it is often useful to have a debugging specialisation environment without incurring any additonal overhead when it is not required. Using LIX we can produce a debugging or non-debugging specialiser from the same base code, the overhead of debugging being specialised away when it is not required. By augmenting LIX with extra options we can produce several versions of the cogen depending on the requirements:

– a debugging cogen, useful if the specialisation does not work as expected
– a profiling cogen
– a simple cogen, whose generating extensions produce no code but which can be fed into termination analysers or abstract interpreters to obtain information to check the annotations.

We could also play with the annotations of LIX to produce more or less aggressive specialisers, depending on the desired tradeoff between specialisation time, size of the specialised code and the generating extensions, and quality of the specialised code. This would be more flexible and maintainable than re-writing LOGEN to accomodate various tradeoffs.

Extensions for Deforestation/Tupling LIX is more flexible than LOGEN: we do not have to know beforehand which predicates are susceptible to being unfolded or memoised. Hence, LIX can handle a potentially unbounded number of predicates. Using this allows LIX to perform a simple form of conjunctive partial deduction [5].

For example, the following is the well known double append example where conjunctive partial deduction can remove the unnecessary intermediate datastructure XY (this is *deforestation*):

```
doubleapp(X,Y,Z,XYZ) :- append(X,Y,XY), append(XY,Z,XYZ).
append([],L,L).
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

When annotating this example for LIX we can now simply annotate a conjunction as memo (which is not allowed in LOGEN):

```
ann_clause(1,doubleapp(A,B,C,D), (memo((append(A,B,E),append(E,C,D))))).
```

Running LIX on this will produce a result where the intermediate datastructure has been removed (after post-processing, as in [5]):

```
doubleapp(A,B,C,D) :- doubleapp__0(A,B,C,D).
append__2([],B,B).
append__2([C|D],E,[C|F]) :- append__2(D,E,F).
conj__1([],[],B,B).
conj__1([],[C|D],E,[C|F]) :- append__2(D,E,F).
conj__1([G|H],I,J,[G|K]) :- conj__1(H,I,J,K).
doubleapp__0(B,C,D,E) :- conj__1(B,C,D,E).
```

For this example to work in LOGEN we would need to declare every possible conjunction skeleton beforehand, as a specialised unfolder predicate has to be generated for every such conjunction. LIX is more flexible in that respect, as it can unfold a conjunction even if it has not been declared before.

We have also managed to deal with the rotate-prune example from [5], but more research will be needed into the extent that the extra flexibility of LIX can be used to do deforestation or tupling in practice. It should be possible, for example, to find out whether there is a bounded number of conjunction skeletons simply by self-application.

7 Conclusions and Future Work

We have presented an implemented, effective and surprisingly simple, self-applicable partial evaluation system for Prolog and have demonstrated that the ground representation is not required for a partial evaluation system to be self-applicable. The LIX system can be used for the specialisation of non-trivial interpreters, and we hope to extend the system to use more sophisticated binding types developed for LOGEN.

While LIX and LOGEN essentially perform the same task, there are some situations where a self-applicable partial evaluation system is preferrable. LIX can potentially produce better generating extensions, using specialised versions of `gensym` and performing some of the generalisation and filtering beforehand. We have shown the potential for the use of LIX in deforestation, and in producing multiple cogens from the same code. Tweaking the annotation of LIX allows the cogen generation to be controlled. The overhead of a debugging cogen can be removed or a more aggressive specialiser can be generated.

At present the annotations for LIX and LOGEN are placed by hand. We are still working on a fully automatic binding time analysis (bta). The automatic bta will be used with a graphical interface allowing the user to tweak the annotations.

References

[1] S. M. Abramov and R. Glück. Semantics modifiers: an approach to non-standard semantics of programming languages. In M. Sato and Y. Toyama, editors, *Third Fuji International Symposium on Functional and Logic Programming,* page to appear. World Scientific, 1998.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[3] A. Bondorf, F. Frauendorf, and M. Richter. An experiment in automatic self-applicable partial evaluation of Prolog. Technical Report 335, Lehrstuhl Informatik V, University of Dortmund, 1990.

[4] A. F. Bowers and C. A. Gurr. Towards fast and declarative meta-programming. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming,* pages 137–166. MIT Press, 1995.

[5] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming,* 41(2 & 3):231–277, 1999.

[6] H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing,* 6(2 & 3):91–118, 1988.

[7] Y. Futamura. Partial evaluation of a computation process — an approach to a compiler-compiler. *Systems, Computers, Controls,* 2(5):45–50, 1971.

[8] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

[9] J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93,* pages 88–98. ACM Press, 1993.

[10] R. Glück. On the generation of specialisers. *Journal of Functional Programming,* 4(4):499–514, 1994.

[11] R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In S. Swierstra and M. Hermenegildo, editors, *Proceedings PLILP'95,* LNCS 982, pages 259–278, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[12] C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel.* PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[13] C. A. Gurr. Specialising the ground representation in the logic programming language Gödel. In Y. Deville, editor, *Proceedings LOPSTR'93,* Workshops in Computing, pages 124–140, Louvain-La-Neuve, Belgium, 1994. Springer-Verlag.

[14] P. Hill and J. Gallagher. Meta-programming in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming,* volume 5, pages 421–497. Oxford Science Publications, Oxford University Press, 1998.

[15] C. K. Holst. Syntactic currying: yet another approach to partial evaluation. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1989.

[16] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[17] J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar,* LNCS 1110, pages 238–262, Schloß Dagstuhl, 1996. Springer-Verlag.

[18] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92,* LNCS 649, pages 49–69. Springer-Verlag, 1992.

[19] M. Leuschel. Partial evaluation of the "real thing". In L. Fribourg and F. Turini, editors, *Proceedings of LOPSTR'94 and META'94,* LNCS 883, pages 122–137, Pisa, Italy, June 1994. Springer-Verlag.

[20] M. Leuschel and D. De Schreye. Towards creating specialised integrity checks through partial evaluation of meta-interpreters. In *Proceedings PEPM'95,* pages 253–263, La Jolla, California, June 1995. ACM Press.

[21] M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming,* 4(1):139–191, 2004.

[22] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems,* 20(1):208–258, January 1998.

[23] B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. *The Journal of Logic Programming,* 22(1):47–99, 1995.

[24] T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *Proceedings of LOPSTR'92,* pages 214–227. Springer-Verlag, 1992.

[25] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming,* 19& 20:261–320, May 1994.

[26] S. Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.

[27] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing,* 12(1):7–51, 1993.

[28] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science,* 248:211–242, 2000.

Multivariant Non-failure Analysis via Standard Abstract Interpretation

Francisco Bueno[1], Pedro López-García[1], and Manuel Hermenegildo[1,2]

[1] School of Computer Science, Technical University of Madrid (UPM)
{bueno,pedro.lopez,herme}@fi.upm.es
http://www.cliplab.org/
[2] Depts. of Comp. Science and El. and Comp. Eng., U. of New Mexico (UNM)

Abstract. Non-failure analysis aims at inferring that predicate calls in a program will never fail. This type of information has many applications in functional/logic programming. It is essential for determining lower bounds on the computational cost of calls, useful in the context of program parallelization, instrumental in partial evaluation and other program transformations, and has also been used in query optimization. In this paper, we re-cast the non-failure analysis proposed by Debray et al. as an abstract interpretation, which not only allows to investigate it from a standard and well understood theoretical framework, but has also several practical advantages. It allows us to incorporate non-failure analysis into a standard, generic abstract interpretation engine. The analysis thus benefits from the fixpoint propagation algorithm, which leads to improved information propagation. Also, the analysis takes advantage of the multi-variance of the generic engine, so that it is now able to infer separate non-failure information for different call patterns. Moreover, the implementation is simpler, and allows to perform non-failure and covering analyses alongside other analyses, such as those for modes and types, in the same framework. Finally, besides the precision improvements and the additional simplicity, our implementation (in the Ciao/CiaoPP multiparadigm programming system) also shows better efficiency.

1 Introduction

Non-failure analysis involves detecting at compile time that, for any call belonging to a particular (possibly infinite) class of calls, a predicate will never fail. As an example, consider a predicate defined by the following two clauses:

```
abs(X, Y) :- X >= 0, Y is X.
abs(X, Y) :- X < 0, Y is -X.
```

and assume that we know that this predicate will always be called with its first argument bound to an integer, and the second argument a free variable. Obviously, for any particular call, one or the other of the tests `X >= 0` and `X < 0` may fail; however, taken together, one of them will always succeed. Thus, we can infer that calls to the predicate will never fail.

Being able to determine statically that a predicate will not fail has many applications. It is essential for determining lower bounds on the computational

© Springer-Verlag Berlin Heidelberg 2004

cost of goals since without such information a lower bound of almost zero (corresponding to an early failure) must often be assumed [10]. Detecting non-failure is also very useful in the context of parallelism because it allows avoiding unnecessary speculative parallelism and ensuring no-slowdown properties for the parallelized programs (in addition to using the lower bounds mentioned previously to perform granularity control) [11]. Non-failure information is also instrumental in partial evaluation and other program transformations, such as reordering of calls, and has also been used in query optimization in deductive databases [8]. It is also useful in program debugging, where it allows verifying user assertions regarding non-failure of predicates [12,13]. Finally, similar techniques can be used to detect the absence of errors or exceptions when running particular predicates.

A practical non-failure analysis has been proposed by Debray *et al.* [9]. In a similar way to the example above, this approach relies on first inferring mode and type information, and then testing that the constraints in the clauses of the predicate are entailed by the types of the input arguments, which is called a *covering* test. Covering cannot be inferred by examining the constraints of each clause separately: it is necessary to collect them together and examine the behavior of the predicate as a whole. Furthermore, non-failure of a given predicate depends on non-failure of other predicates being called and also possibly on the constraints in such predicates.

While [9] proposed the basic ideas behind non-failure analysis, only a simple, monovariant algorithm was proposed for propagating the non-failure information. In our experience since that proposal, we have found a need to improve it in several ways. First, information propagation needs to be improved, which leads us to a fixpoint propagation algorithm. Furthermore, the analysis really needs to be *multi–variant,* which means that it should be able to infer separate non-failure (and covering) information for different call patterns for a given predicate in a program. This is illustrated by the following example which, although simple, captures the very common case where the same (library) procedure is called from a program (in different points) for different purposes:

Example 1 Consider the (exported) predicate `mv/3` (which uses the library predicate `qsort/2`), defined for the sake of discussion as follows:

```
mv(A,B,C):- qsort(A,B), !, C = B.
mv(A,B,C):- append(A,B,D), qsort(D, C).
```

Assume the following entry assertion for `mv/3`:

```
:- entry mv(A,B,C) : (list(A, num), list(B, num), var(C)).
```

which means that the predicate `mv(A,B,C)` will be called with `A` and `B` bound to lists of numbers, and `C` a free variable. A multi–variant non-failure analysis would infer two call patterns for predicate `qsort/2`:

1. The call pattern `qsort(A,B): (list(A,num), list(B,num))`, for which the analysis infers that it *can fail* and is *not covered,* and
2. the call pattern `qsort(A,B): (list(A,num), var(B))`, for which the analysis infers that it will *not fail* and *is covered.*

This in turn allows the analysis to infer that the predicate `mv/3` will *not fail* and *is covered* (for the call pattern expressed by the entry assertion).

However, a monovariant analysis only considers one call pattern per predicate. In particular, for predicate `qsort/2`, the call pattern used is `qsort(A,B):(list(A,num), term(B))`[3] (which is the result of "collapsing" all call patterns which can appear in the program, so that precision is lost), for which it infers that `qsort/2` *can fail* and *is not covered.* This causes the analysis to infer that the predicate `mv/3` *can fail* (since the calls to `qsort/2` in both clauses of predicate `mv/3` are detected as failing) and *is covered.* □

In order to address the different shortcomings of [9] in this paper we start by casting the ideas behind non-failure and covering analysis as an abstract interpretation [5]. This then allows us to incorporate non-failure analysis into a (somewhat modified) standard, generic abstract interpretation engine. This has several advantages. First of all, the analysis is now based on a standard and well studied theoretical framework. But, most importantly, being able to take advantage of standard and well developed analysis engines allows us to obtain a simpler and more efficient implementation, with better propagation of information, performing an efficient fixpoint. The non-failure and covering analyses can be performed alongside other abstract interpretation based analyses, such as those for modes and types, in the same framework. Furthermore, the analysis that we obtain is *multi–variant* (on calls and successes) thus inferring separate non-failure (and covering) information for different call patterns for a given predicate in a program. Finally, the abstract domain for non-failure can be easily enhanced to define a domain for determinacy of predicates.

Abstract Interpretation [5] is often proposed as a means for inferring properties of programs at compile–time. It was shown by Bruynooghe [2], Jones and Sondergaard [15], Debray [7], and Mellish [17] that this technique can be extended to flow analysis of programs in logic programming languages, and several frameworks or particular analyses have evolved since (e.g. [16,20,21,22]). Abstract interpretation formalizes the relation between analysis and semantics, and, therefore, it is inherently semantics sensitive, different semantic definition styles yielding different approaches to program analysis. For logic programs we distinguish between two main approaches, namely *bottom-up* analysis and *top–down* analysis. We also distinguish between *goal dependent* and *goal independent* analyses. In this paper we use a goal dependent framework, since non-failure analysis is inherently goal dependent. In [3], Bruynooghe describes a framework for the goal-dependent, top–down abstract interpretation of logic programs. We use the PLAI/CiaoPP framework [12,13], which follows [3], but incorporates a number of optimizations and efficient fixpoint algorithms, described in [18,19,14].

[3] `term(B)` means that argument `B` can be bound to any term.

2 Preliminaries

We will denote $\mathcal{C}$ the universal set of constraints. We let $\theta\downarrow_L$ be the constraint θ restricted to the variables of the syntactic object L. We denote constraint entailment by $\models$, so that $c_1 \models c_2$ denotes that c_1 entails c_2.

An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and the t_i are terms. A *literal* is either an atom or a constraint. A *goal* is a finite sequence of literals. A *rule* is of the form H :- B where H, the *head*, is an atom and B, the *body*, is a possibly empty finite sequence of literals. A *constraint logic program*, or *program*, is a finite set of rules. The *definition* of an atom A in program P, $defn_P(A)$, is the set of variable renamings of rules in P such that each renaming has A as a head and has distinct new local (but not head) variables.

The operational semantics of a program is in terms of its "derivations" which are sequences of reductions between "states". A *state* $\langle G \mid \theta\rangle$ consists of a goal G and a constraint store (or *store* for short) θ. A state $\langle L :: G \mid \theta\rangle$, where L is a literal and :: denotes concatenation of sequences, can be *reduced* as follows:

1. If L is a constraint and $\theta \wedge L$ is satisfiable, it is reduced to $\langle G \mid \theta \wedge L\rangle$.
2. If L is an atom, it is reduced to $\langle B :: G \mid \theta\rangle$ for some rule $(L$:-$B) \in defn_P(L)$.

assuming for simplicity that the underlying constraint solver is complete. We use $S \leadsto_P S'$ to indicate that in program P a reduction can be applied to state S to obtain state S'. Also, $S \leadsto_P^* S'$ indicates that there is a sequence of reduction steps from state S to state S'. A *derivation* from state S for program P is a sequence of states $S_0 \leadsto_P S_1 \leadsto_P ... \leadsto_P S_n$ where S_0 is S and there is a reduction from each S_i to S_{i+1}. Given a non-empty derivation D, we denote by *curr_goal(D)* and *curr-store(D)* the first goal and the store in the last state of D, respectively. E.g., if D is the derivation $S_0 \leadsto_P^* S_n$ with $S_n = \langle g :: G \mid \theta\rangle$ then $curr\_goal(D) = g$ and $curr\_store(D) = \theta$. A *query* is a pair (L, θ) where L is a literal and θ a store of an initial state $\langle L \mid \theta\rangle$. The set of all derivations from Q for P is denoted *derivations(P, Q)*. We will denote sets of queries by $\mathcal{Q}$. We extend *derivations* to $\mathcal{Q}$ as follows: $derivations(P, \mathcal{Q}) = \bigcup_{Q \in \mathcal{Q}}$ *derivations(P,Q)*.

The observational behavior of a program is given by its "answers" to queries. A finite derivation from a query (L, θ) for program P is *finished* if the last state in the derivation cannot be reduced. A finished derivation from a query (L, θ) is *successful* if the last state is of the form $\langle nil \mid \theta'\rangle$, where *nil* denotes the empty sequence. The constraint $\theta'\downarrow_L$ is an *answer* to (L, θ). We denote by *answers(P, Q)* the set of answers to query Q. A finished derivation is *failed* if the last state is not of the form $\langle nil \mid \theta\rangle$. Note that $derivations(P, \mathcal{Q})$ contains not only finished derivations but also all intermediate derivations. A query Q *finitely fails* in P if *derivations(P, Q)* is finite and contains no successful derivation.

Abstract Interpretation. Abstract interpretation [5] is a technique for static program analysis in which execution of the program is simulated on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain (D)*. For this study, we restrict to complete lattices over sets both for the concrete $\langle 2^D, \subseteq\rangle$ and abstract $\langle D_\alpha, \sqsubseteq\rangle$ domains.

Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle\alpha,\gamma\rangle$: *abstraction* $\alpha : 2^D \rightarrow D_\alpha$, and *concretization* $\gamma : D_\alpha \rightarrow 2^D$, such that $\forall x \in 2^D : \gamma(\alpha(x)) \supseteq x$ and $\forall y \in D_\alpha : \alpha(\gamma(y)) = y$. In general $\sqsubseteq$ is defined so that the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^D in a precise sense:

$$\forall\lambda,\lambda' \in D_\alpha : \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$$
$$\forall\lambda_1,\lambda_2,\lambda' \in D_\alpha : \lambda_1 \sqcup \lambda_2 = \lambda' \Leftrightarrow \gamma(\lambda_1) \cup \gamma(\lambda_2) = \gamma(\lambda')$$
$$\forall\lambda_1,\lambda_2,\lambda' \in D_\alpha : \lambda_1 \sqcap \lambda_2 = \lambda' \Leftrightarrow \gamma(\lambda_1) \cap \gamma(\lambda_2) = \gamma(\lambda')$$

Goal dependent abstract interpretation takes as input a program P, an abstract domain D_α, and a description Q_α of the possible initial queries to P, given as a set of abstract queries. An *abstract query* is a pair (L,λ), where L is an atom (for one of the exported predicates) and $\lambda \in D_\alpha$ describes the initial stores for L. A set Q_α represents the set of queries $\gamma(Q_\alpha)$, which is defined as $\gamma(Q_\alpha) = \{(L,\theta) \mid (L,\lambda) \in Q_\alpha \wedge \theta \in \gamma(\lambda)\}$. Such an abstract interpretation computes a set of triples $Analysis(P, Q_\alpha, D_\alpha) = \{\langle L_p, \lambda^c, \lambda^s\rangle \mid p$ is a predicate of $P\}$, where L_p is a (program) atom for predicate p. Note that, the analysis being multivariant (on calls), it may compute several tuples of the form $\langle L_p, \lambda^c, \lambda^s\rangle$ for different call patterns $\langle L_p, \lambda^c\rangle$ of each predicate p (including different program atoms L_p). If p is detected to be dead code then $\lambda^c = \lambda^s = \bot$. As usual in abstract interpretation, $\bot$ denotes the abstract constraint such that $\gamma(\bot) = \emptyset$, whereas $\top$ denotes the most general abstract constraint, i.e., $\gamma(\top) = D$.

3 The Abstract Interpretation Framework

PLAI is an analysis system based on the abstract interpretation framework of Bruynooghe [3] with the optimizations described in [18]. The framework works on an abstraction of the (SLD) AND-OR trees of the execution of a program for given entry points. The abstract AND-OR graph makes it possible to provide information at each program point, a feature which is crucial for many applications (such as, e.g., reordering, automatic parallelization, or garbage collection).

Program points and abstract substitutions are related as follows. Consider a clause h:- $p_1,\ldots,p_n$. Let λ_i and λ_{i+1} be the abstract substitutions to the left and right of the subgoal p_i, $1 \leq i \leq n$ in this clause. Then λ_i and λ_{i+1} are, respectively, the *abstract call substitution* and the *abstract success substitution* for the subgoal p_i. For this same clause, λ_1 is the *abstract entry substitution* and λ_{n+1} is the *abstract exit substitution.* Entry and exit substitutions are denoted respectively β_{entry} and β_{exit} when projected on the variables of the clause head.

Computing the *success* substitution from the *call* substitution is done as follows (see Figure 1(a)). Given a call substitution λ_{call} for a subgoal p, let $h_1,\ldots,h_m$ be the heads of clauses which unify with p. Compute the entry substitutions $\beta 1_{entry},\ldots,\beta m_{entry}$ for these clauses. Compute their exit substitutions $\beta 1_{exit},\ldots,\beta m_{exit}$ as explained below. Compute the success substitutions $\lambda 1_{success},\ldots,\lambda m_{success}$ from the corresponding exit substitutions. At this point, all different success substitutions can be considered for the rest of the analysis, or a single success substitution $\lambda_{success}$ for subgoal p computed by means of an aggregation operation for $\lambda 1_{success},\ldots,\lambda m_{success}$. This aggregator is usually the

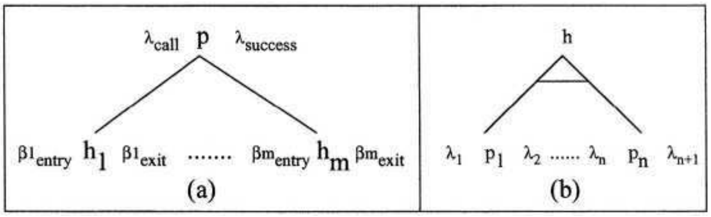

λ_{call} p $\lambda_{success}$
$\beta 1_{entry}$ h_1 $\beta 1_{exit}$ .... βm_{entry} h_m βm_{exit}
(a)
h
λ_1 p_1 λ_2 ...... λ_n p_n λ_{n+1}
(b)

Fig. 1. Illustration of the Top–Down Abstract Interpretation Process

LUB (least upper bound) of the abstract domain. In the first case the analysis is *multi-variant* on successes, in the second case it is not.

Computing the *exit* substitution from the *entry* substitution is straightforward (see Figure 1(b)). Given a clause h:- $p_1, \ldots, p_n$ and an entry substitution β_{entry} for the clause head h, λ_1 is the call substitution for p_1. This one is computed simply by adding to β_{entry} an abstraction for the free variables in the clause. The success substitution λ_2 for p_1 is computed as explained above (essentially, by repeating this same process for the clauses which match p_1). Similarly, $\lambda_3, \ldots, \lambda_{n+1}$ are computed. The exit substitution B_{exit} for this clause is precisely the projection onto h of λ_{n+1}.

If, from a different subgoal in the program, a different entry substitution is computed for an already analyzed clause, different call substitutions will appear (for p_1 and possibly the other subgoals). These substitutions can be collapsed using the LUB operation, or a different node in the graph can be computed. In the latter solution, different nodes exist in the graph for each call substitution and subgoal, thus yielding an analysis which is *multi–variant* on calls.

Note that the framework itself is domain independent. To instantiate it, a particular analysis needs to define an abstract domain and abstract unification, and the $\sqsubseteq$ relation, which in turn defines $\sqcup$ (LUB). Abstract unification is divided into two in the framework, so that it is required to define: (1) how to compute the entry substitution for a clause C given a subgoal p (which unifies with the head of C) and its call substitution; and (2) how to compute the success substitution for a subgoal p given its call substitution and the exit substitution for a clause C whose head unifies with p. We formalize this with functions *entry_to_exit* and *call_to_success* in Figure 2. The domain dependent functions used there are:

- *call_to_entry*$(p(\bar{u}), C, \lambda)$ which gives an abstract substitution describing the effects on *vars(C)* of unifying $p(\bar{u})$ with *head(C)* given an abstract substitution λ describing $\bar{u}$,
- *exit_to_success*$(\lambda, p(\bar{u}), C, \beta)$ which gives an abstract substitution describing $\bar{u}$ accordingly to β (which describes *vars(head(C)))* and the effects of unifying $p(\bar{u})$ with *head(C)* under the abstract substitution λ describing $\bar{u}$,
- *extend*(λ, λ') which extends abstract substitution λ to incorporate the information in λ' in a way that it is still consistent,
- *project_in*$(\bar{v}, \lambda)$ which extends λ so that it refers to all of the variables $\bar{v}$,
- *project_out*$(\bar{v}, \lambda)$ which restricts λ to only the variables $\bar{v}$.

```
$entry\_to\_exit(C, \beta_{entry}) \equiv$
    $A_1 := project\_in(vars(C), \beta_{entry})$;
    For $i := 1$ to $length(C)$ do
        $A_{i+1} := call\_to\_success(q_i(\bar{u}_i), A_i))$;
    return $project\_out(vars(head(C)), A_{n+1})$;

$call\_to\_success(p(\bar{u}), \lambda_{call}) \equiv$
    $\lambda := project\_out(\bar{u}, \lambda_{call})$; $\lambda' := \bot$;
    For each clause $C$ which matches $p(\bar{u})$ do
        $\beta_{exit} := entry\_to\_exit(C, call\_to\_entry(p(\bar{u}), C, \lambda))$;
        $\lambda' := \lambda' \sqcup exit\_to\_success(\lambda, p(\bar{u}), C, \beta_{exit})$;
    od;
    return $extend(\lambda_{call}, \lambda')$;
```

Fig. 2. The Top–Down Framework

In the presence of recursive predicates, analysis requires a fixpoint computation. In [18,19] a fixpoint algorithm was proposed for the framework that localizes fixpoint computations to only the strongly connected components of (mutually) recursive predicates. Additionally, an initial approximation to the fixpoint is computed from the non-recursive clauses of the recursive predicate. Fixpoint convergence is accelerated by updating this value with the information from every clause analyzed in turn. The algorithm is (schematically) shown in Figure 3. For a complete description see [18,19].

4 Abstract Framework, Domain, and Operations for Non-failure Analysis

In the non-failure analysis, the covering test is instrumental. In fact, covering can be seen as a notion that characterizes the fact that execution of a query will not finitely fail, i.e., if it has finished derivations then at least one is successful. Note that, as in [9], non-failure does not imply success: a predicate that is non-failing may nevertheless not produce an answer because it does not terminate.

Definition 1 (Covering). *Given computation state $\langle g :: G \mid \theta \rangle$ in the execution of program P, define the* global answer constraint *of goal g in store θ as:*

$$c = \vee\{\ curr\_store(D_i') \mid D_i' \in derivations(P, \langle g, \theta \rangle) \text{ and is maximal } \}$$

Let $\bar{u}$ denote the variables of g already constrained in θ, call them the input *variables. We say that g is* covered *in θ iff $\theta \downarrow_{\bar{u}} \models c \downarrow_{\bar{u}}$.*

It is not difficult to show that, in a pure language, where failure can only be caused by constraint store inconsistency, covering is a sufficient condition for non-failure. Indeed, if g is covered in θ, i.e., $\theta \downarrow_{\bar{u}} \models c \downarrow_{\bar{u}}$, then one of the disjunctions in (the projection of) c is entailed. This corresponds to a (maximal)

```
$call\_to\_success\_recursive(p(\bar{u}), \lambda_{call}) \equiv$
    $\lambda := project\_out(\bar{u}, \lambda_{call})$; $\lambda' := \perp$;
    For each non-recursive clause $C$ which matches $p(\bar{u})$ do
        $\beta_{exit} := entry\_to\_exit(C, call\_to\_entry(p(\bar{u}), C, \lambda))$;
        $\lambda' := \lambda' \sqcup exit\_to\_success(\lambda, p(\bar{u}), C, \beta_{exit})$;
    od;
    $\lambda'' := fixpoint(p(\bar{u}), \lambda, \lambda')$;
    return $extend(\lambda_{call}, \lambda'')$;

$fixpoint(p(\bar{u}), \lambda, \lambda') \equiv$
    $\lambda'' := \lambda'$;
    For each recursive clause $C$ which matches $p(\bar{u})$ do
        $\beta_{exit} := entry\_to\_exit(C, call\_to\_entry(p(\bar{u}), C, \lambda))$;
        $\lambda'' := \lambda'' \sqcup exit\_to\_success(\lambda, p(\bar{u}), C, \beta_{exit})$;
    od;
    If $\lambda'' = \lambda'$ then return $\lambda''$
    else return $fixpoint(p(\bar{u}), \lambda, \lambda'')$;
```

Fig. 3. The Fixpoint Computation

derivation of $\langle g, \theta \rangle$, and this derivation cannot be failed, since, if it were, it would be inconsistent, and no inconsistent constraint can be entailed by a consistent one. Therefore, either such derivation is infinite, or, if finite, it is successful. Thus

If g is *covered* in θ then $\langle g, \theta \rangle$ does not finitely fail.

A key issue in non-failure analysis will thus be how to approximate the current store and the global answer constraint so that covering can be effectively and accurately approximated. In [9] such an approximation is defined in the following terms: A goal is non-failing if there is a subset of clauses of the predicate which do not fail and which match the input types of the goal. This "matching" is the so-called *covering test,* and basically amounts to the analysis being able to gather, for each such clause, enough constraints on the input variables of the goal to be able to prove that, for each of the variables, any element in the corresponding type satisfies at least the constraint gathered for one clause. An analysis for non-failure thus needs to traverse the clauses of a predicate to check non-failure of the clause body goals, collect constraints that approximate the global answer constraint, and finally check that they cover the input types of the original goal. In the rest of this section, we show how to accommodate the abstract interpretation based framework of the previous section to perform these tasks, and define an abstract domain suitable for them.

4.1 Abstract Domain

The abstractions for non-failure analysis are made of four components. The first two are (abstractions of) constraints that represent the current store and the global answer constraint for the current goal. This is the core part of the domain. The other two components carry the results of the covering test, specifying if the current constraint store covers the global answer constraint, and if this implies that the computation may fail or not. The covering and non-failure information is represented by values of the set $\mathcal{B} = \{\top, \bar{0}, \bar{1}, \bot\}$, where $\bar{0}$ and $\bar{1}$ are not comparable in the ordering. For covering, $\bar{0}$ is interpreted as "not covered" and $\bar{1}$ as covered. For non-failure, $\bar{0}$ is interpreted as "not failing" and $\bar{1}$ as failing.

Definition 2 (Abstract Domain). *Let $\mathcal{C}^{\alpha_1}$ and $\mathcal{C}^{\alpha_2}$ be abstract domains for $\mathcal{C}$. The abstract domain for non-failure is the set*

$$\mathcal{F} = \{(s,c,o,f) \mid s \in \mathcal{C}^{\alpha_1}, c \in \mathcal{C}^{\alpha_2}, o \in \mathcal{B}, f \in \mathcal{B}\}$$

The ordering in domain $\mathcal{F}$ is induced from that in $\mathcal{B}$, so that (overloading $\sqsubseteq$):

$$(s_1,c_1,o_1,f_1) \sqsubseteq (s_2,c_2,o_2,f_2) \text{ iff } f_1 \sqsubseteq f_2$$

In an element $(s,c,o,f) \in \mathcal{F}$, components s and c are abstractions α_1 and α_2 of the constraint domain $\mathcal{C}$. The usual approximations used (e.g., in [9]) are types (and modes) for s, and a finite set of (concrete) constraints for c.

Definition 3 (Abstraction Function). *The abstraction of a derivation D in the execution of program P, such that curr_store(D) $= \theta$ and curr_goal(D) $= g$, and the input variables and global answer constraint of g in θ are respectively $\bar{u}$ and c, is $\alpha(D) = (\theta^{\alpha_1}, c^{\alpha_2}, o, f)$, where:*

$$f = \begin{cases} \bar{1} & \text{if } D \text{ is failed} \\ \bar{0} & \text{otherwise} \end{cases} \quad \text{and } o = \begin{cases} \bar{1} & \text{if } \theta^{\alpha_1}\downarrow_{\bar{u}} \models^{\alpha} c^{\alpha_2}\downarrow_{\bar{u}} \\ \bar{0} & \text{otherwise} \end{cases}$$

It is easy to show that such an abstraction is correct, provided that α_1 and α_2 are also correct abstractions, and that the corresponding abstract covering test ($\models^{\alpha}$) correctly approximates Definition 1. For α_1 we have already mentioned the use of type and mode information. One possibility for α_2 is to use only those constraints appearing explicitly in the clause bodies of the predicate whose covering test is to be performed (the current goal g in the derivation).

Example 2 Consider the following (contrived) predicates:
```
p(X,Y,Z):- X =< Y, q(X,Z).
q(X,Y):- X =< Y.
```
The global answer constraint for `p(X,Y,Z)` is `X =< Y` $\wedge$ `X =< Z`, but it can be approximated simply by `X =< Y`, the only constraint in the definition of `p/3`. □

One rationale for the above choice might be that collecting all constraints in derivations may not be possible during a compile-time analysis (since such constraints are only known during execution), or may lead to non-termination of the analysis. However, the first problem can be alleviated by proper abstractions of the tests (such as a depth-k abstraction, in a way similar to [6]), and the second problem only occurs for recursive predicates. Thus, the most simple solution to the termination problem is to avoid collecting constraints in recursive calls.[4]

[4] Note that this does not imply that recursive calls are simply ignored. They need to be considered to check that they are indeed non-failing, even though their global answer constraint is not computed.

Example 3 The global answer constraint for the predicate sorted/1 defined below includes a constraint for each two elements in the input list, the length of which is not in general known at compile-time.

```
sorted([]).
sorted([_]).
sorted([X,Y|L]):- X =< Y, sorted([Y|L]).
```
□

Our solution to this problem[5] is to collect only constraints that refer literally to the predicate arguments in the program clause head, which also excludes in general (but not always) the constraints arising from recursive calls.

Example 4 Consider again the predicate sorted/1 defined in the previous example. We collect constraints only for the clause head argument [X,Y|L], which amounts to only one constraint: X =< Y (since the recursive call does not provide constraints for the head arguments that appear literally in the program).

Consider, on the other hand, predicate p/3 of Example 2. In this case the complete global answer constraint for p(X,Y,Z) will be collected: X =< Y $\wedge$ X =< Z, since the two single constraints can be "projected" onto the clause head. □

Note that such a solution yields an under-approximation of the global answer constraints. Given the use of type and mode information, which are in general over-approximations, we have that, for any element $(s, c, o, f) \in \mathcal{F}$, given current constraint store θ and global answer constraint ω, $s = \theta^{\alpha_1}$ is an over-approximation of θ, and $c = \omega^{\alpha_2}$ is an under-approximation of ω. In this situation, it is not difficult to prove that $\theta^{\alpha_1} \downarrow_{\bar{u}} \models^{\alpha} \omega^{\alpha_2} \downarrow_{\bar{u}}$ correctly approximates covering: $\theta \downarrow_{\bar{u}} \models \omega \downarrow_{\bar{u}}$.

4.2 Abstract Operations

Abstract values $(s, c, o, f) \in \mathcal{F}$ are built during analysis in the following way: f is carried along during the abstract computation by the abstract operations below, o is computed from the covering test, c is collected as explained above, and for s, type and mode analysis is performed. Thus, our analysis is in fact three-fold: it carries on mode, type, and non-failure analyses simultaneously. We focus now on the abstract operations for non-failure, given that those for types and modes are standard:

- $call\_to\_entry(p(\bar{u}), C, \lambda)$ solves head unification $p(\bar{u}) = head(C)$, and checks that it is consistent with the c component of λ. If it is not, it returns $\perp$, otherwise, the resulting abstraction.
  If $p(\bar{u}) \in C$, i.e., if it happens to be a constraint itself, then no clause C exists, and $p(\bar{u})$ itself is added to the c component. In this case the following *exit_to_success* function is not called.

[5] However, we plan to investigate other solutions. In particular, the use of a depth-k abstraction seems to be a very promising one.

- $exit\_to\_success(\lambda, p(\bar{u}), C, \beta)$ adds the equations resulting from unification $p(\bar{u}) = head(C)$ to the c component of β and projects it onto $vars(\bar{u})$.
  It is the projection performed here that gets rid of useless constraints, like in the case of Example 4. Constraints that cannot be projected onto the (goal) variables $\bar{u}$ are simply dropped in the analysis.
- $\lambda \uplus \lambda'$ adds abstraction λ to the set λ' if λ is non-failing.
- $extend(\lambda, \lambda')$ performs the covering test for λ' (a set of abstractions); if it is successful, the c component of λ' is merged with that of λ.
  This operation uses the covering algorithm described in [9], which takes the global answer constraint c and a *type, assignment* for the input variables appearing in c. Given a finite set of variables V, a type assignment over V is a mapping from V to a set of types. This is computed from the type information in the first component of λ. Input variables are determined from the mode information in that same component. The global answer constraint is obtained as the disjunction of the c components of each abstraction in λ'.

4.3 Adapting the Analysis Framework

The framework described in the previous section is not adequate for non-failure analysis. The main reason for this is that the aggregation function for the successive exit abstractions of the different clauses is not the LUB anymore. In non-failure analysis, the constraints for each clause need to be gathered together, and a covering test on the set of constraints needs to be performed. Another difference is that the covering test should only consider constraints from clauses that are not guaranteed to fail altogether;[6] therefore the aggregator must be able to discriminate abstract substitutions on this criterion.

We have adapted the definition of the *call_to_success* function to reflect the aggregation operator. The adapted definition is shown in Figure 4. Note that, as a result of this, λ' in the algorithm is not anymore an abstract substitution, but a set of them. This is input to *extend,* which is in charge of the covering test.

```
$call\_to\_success(p(\bar{u}), \lambda_{call}) \equiv$
    $\lambda := project\_out(\bar{u}, \lambda_{call})$; $\lambda' := \emptyset$;
    For each clause $C$ which matches $p(\bar{u})$ do
        $\beta_{exit} := entry\_to\_exit(C, call\_to\_entry(p(\bar{u}), C, \lambda))$;
        $\lambda' := \lambda' \uplus exit\_to\_success(\lambda, p(\bar{u}), C, \beta_{exit})$;
    od;
    return $extend(\lambda_{call}, \lambda')$;
```

Fig. 4. The Top–Down Framework for Non-Failure Analysis

When fixpoint computation is required, adapting the framework is a bit more involved. Basically, since the aggregation operator is not LUB, fixpoint detection

[6] Note how this information could be used to improve the results of other analyses.

cannot be performed right after the success substitution has been computed. Normally, it is the LUB that is used for updating the successive approximations to the fixpoint value, and fixpoint detection works by simply comparing the initial and the final values for the success substitution. In non-failure analysis, the covering test must be performed first, and only after this one has been performed, the test for the fixpoint can be done. The resulting algorithm is shown in Figure 5. It is basically a simpler fixpoint iterator over the function *call_to_success* abandoning the sophisticated fixpoint computation of Figure 3.

```
$call\_to\_success\_recursive(p(\bar{u}), \lambda_{call}) \equiv$
    $\lambda := project\_out(\bar{u}, \lambda_{call})$;
    return $fixpoint(p(\bar{u}), \lambda, \bot)$;

$fixpoint(p(\bar{u}), \lambda, \lambda') \equiv$
    $\lambda'' := call\_to\_success(p(\bar{u}), \lambda)$;
    If $\lambda'' = \lambda'$ then return $\lambda''$
    else return $fixpoint(p(\bar{u}), \lambda, \lambda'')$;
```

Fig. 5. The Fixpoint Computation for Non-Failure Analysis

A Running Example We now illustrate our analysis by means of a detailed example on how it will proceed. Consider the program (fragment) below:

```
qsort(As,Bs):- qsort(As,Bs,[]).

qsort([X|L],R,R2) :-
    partition(L,X,L1,L2), qsort(L2,R1,R2), qsort(L1,R,[X|R1]).
qsort([],R,R).

partition([],_,[],[]).
partition([E|R],C,[E|Left1],Right):- E < C,  partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):- E >= C, partition(R,C,Left,Right1).
```

Let the abstract call pattern for atom qsort(As, Bs, []) be

$$(\{list(As, num), var(Bs)\}, true, \bar{1}, \bar{0}).$$

Upon entering the first clause defining qsort/3, the result of *call_to_entry* (restricted to the head variables) is

$$(\{num(X), list(L, num), var(R), [](R2)\}, true, \bar{1}, \bar{0})$$ [7]

plus, additionally, $\{var(R1), var(L1), var(L2)\}$ for the free variables in the clause. Once projected, this gives the call pattern for the first literal in that clause:

$$(\{list(L, num), num(X), var(L1), var(L2)\}, true, \bar{1}, \bar{0}).$$

[7] To be concise, we denote with [] (*A*) that the type of *A* is that of the empty lists.

We omit the analysis of the partition predicate. After the fixpoint computation for this predicate, however, we will have a set of three abstract elements corresponding to the abstraction of the three clauses. For brevity, we express such set as a single abstraction where it is the c component that is a set, instead.[8] Note that this is possible because all other components (types, modes, covering, non-failure) of the abstractions in the set are the same. Thus, we have:

$$(\{ list(L,num), num(X), list(L1,num), list(L2, num) \}, \\ \{ L = [] \wedge L1 = [] \wedge L2 = [], \quad L = [E|\_] \wedge E < X \wedge L1 = [E|\_], \\ L = [E|\_] \wedge E >= X \wedge L2 = [E|\_] \}, \quad \overline{1}, \overline{0})$$

This is now extended (by abstract function *extend*) to the corresponding program point of the clause of `qsort`. First, the covering test is performed, and it succeeds, since *list*(*L*, *num*), *num*(*X*) covers indeed the global answer constraint projected onto the input variables:

$$(L = [E|\_] \wedge (E < X \vee E >= X)) \vee L = [].$$

Therefore, computation is still covered and non-failing. This, together with the projection of the c component onto the variables of the first clause of `qsort`, yields success abstraction for partition:

$$(\{ num(X), list(L,num), var(R), var(R1), [](R2), \\ list(L1,num), list(L2,num) \}, \quad true, \overline{1}, \overline{0})$$

where the c component is still *true* since the projection onto the clause variables factors out the previously computed global answer constraint. Now, analysis will proceed into call `qsort(L2,R1,R2)` with

$$(\{list(L2,num), var(R1), [](R2)\}, true, \overline{1}, \overline{0}).$$

Since this is basically the same call pattern that we started with, no new fixpoint computation is started in this case.[9] On the other hand, a new fixpoint computation is started for the second recursive call `qsort(L1,R,[X|R1])` with

$$(\{list(L1,num), var(R), num(X), list(R1,num)\}, true, \overline{1}, \overline{0}).$$

This is a new call pattern for the `qsort` predicate, which initiates a new fixpoint computation. The fixpoint value obtained in this computation is the same abstraction, except for the type of R which on output is a list. Finally, *exit_to_success* now lifts this result to the original goal `qsort(As,Bs,[])` giving:

$$(\{list(As,num), list(Bs,num)\}, As = [\_|\_], \overline{1}, \overline{0}).$$

The analysis of the non-recursive clause immediately gives:

$$(\{[](As), [](Bs)\}, As = [] \wedge Bs = [], \overline{1}, \overline{0}),$$

and *extend* computes the covering test for the set of the above two abstractions with the initial input abstraction, in which the input types are *list*(*As*,*num*). Certainly, this type covers the (projected) global answer constraint $As = [\_|\_] \vee As = []$. Thus, the goal is still covered and non-failing.

Finally, since the abstraction now computed is only the result of a first iteration of the fixpoint computation, a new iteration is started. The result in this case is the same, and fixpoint computation finishes with that very same result.

[8] This very same "trick" is used in the implementation.

[9] Here, we save the reader from some more fixpoint iterations that will be taking place. However, the results are as indicated.

5 Implementation Results

We have constructed a prototype implementation in (Ciao) Prolog by adapting the framework of the PLAI implementation and defining the abstract operations for non-failure analysis that we have described in this paper. Most of these abstract operations have been implemented by reusing code of the implementation in [9], such as for example, the covering algorithm. We have incorporated the prototype in the Ciao/CiaoPP multiparadigm programming system [12,13,4] and tested it on the benchmarks used in the non-failure analysis of Debray *et al.* [9], plus some benchmarks exhibiting paradigmatic behaviours, plus a last group with those used in the cardinality analysis of Braem *et al.* [1]. These two analyses are the closest related previous work that we are aware of. Some relevant results of these tests for non-failure analysis are presented in Table 1. **Program** lists the program names, **N** the number of predicates in the program, **F** and **C** are the number of non-failing predicates detected by the non-failure analysis in [9], and the cardinality analysis in [1], respectively.

Note that our multi–variant analysis can infer several variants (call patterns) for the same predicate, where some of them may be non-failing (resp. covered) and the other ones can be failing (resp. not covered). For instance, in the case of the program *Mv* in Table 1 (also described in Example 1), which has 4 predicates (`mv`/3, `qsort`/2, `partition`/4 and `append`/3), the analysis infers one variant for `mv`/3, which is non-failing and covered, 2 variants for `qsort`/2 (one of them which is non-failing and covered, and the other one which is failing and not covered), one variant for `partition`/4, which is non-failing and covered, and 3 variants for `append`/3 (2 of them which are non-failing and covered, and the other one which is failing and not covered). For this reason, and in order to make the results comparable, column **AF** shows two figures (both corresponding to the analysis presented in this paper): the number of predicates such that **all of their variants** (call patterns) are detected as non-failing, and (between parenthesis) the number of predicates such that **some of their variants** are detected as non-failing (this second figure is omitted if it is equal to the first one).

Similarly, **ACov** shows two figures (both corresponding to the analysis presented in this paper): the number of predicates detected to cover **all** of their (calling) types (variants), and (between parenthesis), the number of predicates detected to cover **some** of their (calling) types. **Cov** is the number of predicates detected to cover their (calling) types by the analysis in [9].

$\mathbf{T}_{AF}$ and $\mathbf{T}_{F}$ are the total time (in milliseconds) required by the analysis presented in this paper and the analysis in [9] respectively (both of which include the time required to derive the modes and types). The timings were taken on a medium-loaded Pentium IV Xeon 2.0Ghz with two processors, 1Gb of RAM memory, running Red Hat Linux 8.0, and averaging several runs and eliminating the best and worst values. Ciao version 1.9.111 and CiaoPP-1.0 were used.

Analysis time averages (per predicate) are also provided in the last row of the table. From these numbers, it is clear that the new implementation based on the abstract interpretation engine is more efficient than the previous one. It is

Program	**N**	**AF**	**F**	**C**	**ACov**	**Cov**	$\mathbf{T}_{AF}$	$\mathbf{T}_{F}$	$\frac{T_{AF}}{T_F}$
Hanoi	2	2	2	N/A	2	2	33	242	0.14
Fib	1	1	1	N/A	1	1	17	22	0.77
Tak	1	1	1	N/A	1	1	9	11	0.82
Subs	1	1	1	N/A	1	1	5	33	0.15
Reverse	2	2	2	N/A	2	2	17	29	0.59
Mv	4	2 (4)	1	N/A	2 (4)	2	54	102	0.53
Zebra	6	2	1	N/A	5 (6)	4	1008	1100	0.92
Family	3	3	1	N/A	3	2	10	18	0.56
Blocks	7	1 (2)	0	N/A	4 (5)	4	30	59	0.51
Reach	2	2	0	N/A	2	1	19	30	0.63
Bid	20	5 (8)	5	N/A	14 (17)	14	3089	3369	0.92
Occur	4	1 (3)	1	N/A	1 (3)	1	69	78	0.88
Plan	16	5 (8)	3	0	11 (13)	10	2626	4128	0.64
Qsort	3	3	3	0	3	3	29	65	0.45
Qsort2	5	3	3	0	3	3	33	76	0.43
Queens	5	2 (3)	2	0	3 (4)	3	60	74	0.81
Pg	10	2 (3)	2	0	6 (9)	6	412	477	0.86
Mean							38 (/p)	58 (/p)	0.67 (/p)

Table 1. Accuracy and efficiency of the non-failure analysis (times in mS).

also more precise, as shown for example in the benchmarks *Mv*, *Zebra, Family, Blocks, Reach,* and *Plan.*

6 Conclusions

We have described a non-failure analysis based on abstract interpretation, which extends the previous proposal of Debray et al. Our analysis improves in precision, and enjoys a clear theoretical setting, and a simpler implementation. Also, the implementation is more efficient. The abstract domain underlying the analysis can be easily modified to cater for a determinacy analysis. Such an analysis, provided with a depth-k abstraction, would be the abstract interpretation counterpart of determinacy analyses such as that of [6]. We are currently working on the verification of this proposition.

The implemented analysis we have described in this paper is currently integrated in CiaoPP, and is being used for lower-bounds cost analysis, granularity control, and program debugging. Arguably, although our presentation covers strictly constraint logic programming, the technique could be easily applied to functional logic languages with similar results, as is indeed the case in the Ciao system, where the analysis presented works without modification for Ciao's functional subset or for combinations of functions and predicates.

Acknowledgments

This work has been supported in part by the European Union IST program under contract IST-2001-34717 "Amos" and IST-2001-38059 "ASAP," by MCYT projects TIC 2002-0055 "CUBICO" and HI2000-0043 "ADELA," and by the Prince of Asturias Chair in Information Science and Technology at the University of New Mexico. The Ciao system is a collaborative effort of members of several institutions, including UPM, UNM, U.Melbourne, Monash U., U.Arizona, Linköping U., NMSU, K.U.Leuven, Bristol U., Ben-Gurion U., and INRIA. The system documentation and related publications contain more specific credits.

References

1. C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of prolog. In *Proc. International Symposium on Logic Programming,* pages 457–471, Ithaca, NY, November 1994. MIT Press.
2. M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Tech. Rep. CW62, Dept. of C.S., Katholieke Universiteit Leuven, October 1987.
3. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming,* 10:91–124, 1991.
4. F. Bueno, P. López-García, G. Puebla, and M. Hermenegildo. The Ciao Prolog Preprocessor. Technical Report CLIP1/04, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, January 2004.
5. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages,* pages 238–252, 1977.
6. Steven Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Extracting determinacy in logic programs. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming,* pages 424–438, Budapest, Hungary, 1993. The MIT Press.
7. S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming,* 5(3):207–229, September 1988.
8. S.K. Debray and N.-W. Lin. Static Estimation of Query Sizes in Horn Programs. In *Third International Conference on Database Theory,* Lecture Notes in Computer Science 470, pages 515–528, Paris, France, December 1990. Springer-Verlag.
9. S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming,* pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.
10. S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium,* pages 291–305. MIT Press, Cambridge, MA, October 1997.
11. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems,* 23(4):472–602, July 2001.
12. M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming,* pages 52–66, Cambridge, MA, November 1999. MIT Press.

13. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03),* number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.
14. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems,* 22(2): 187–223, March 2000.
15. N. Jones and H. Sondergaard. A semantics-based framework for the abstract interpretation of prolog. In *Abstract Interpretation of Declarative Languages,* chapter 6, pages 124–142. Ellis-Horwood, 1987.
16. H. Mannila and E. Ukkonen. Flow Analysis of Prolog Programs. In *Fourth IEEE Symposium on Logic Programming,* pages 205–214, San Francisco, California, September 1987. IEEE Computer Society.
17. C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming,* number 225 in LNCS, pages 463–475. Springer-Verlag, July 1986.
18. K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.
19. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming,* 13(2/3) :315–347, July 1992.
20. T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science,* 34:227–240, 1984.
21. A. Waern. An Implementation Technique for the Abstract Interpretation of Prolog. In *Fifth International Conference and Symposium on Logic Programming,* pages 700–710, Seattle,Washington, August 1988.
22. R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming,* pages 684–699. MIT Press, August 1988.

Set-Sharing Is Not Always Redundant for Pair-Sharing

Francisco Bueno[1] and Maria Garcia de la Banda[2]

[1] Dept. of Computer Science, UPM, Spain
[2] School of Computer Science and S.E., Monash University, Melbourne

Abstract. Sharing among program variables is vital information when analyzing logic programs. This information is often expressed either as sets or as pairs of program variables that (may) share. That is, either as set-sharing or as pair-sharing. It has been recently argued that (a) set-sharing is interesting not as an observable property in itself, but as an encoding for accurate pair-sharing, and that (b) such an encoding is in fact redundant and can be significantly simplified without loss of pair-sharing accuracy. We show that this is not the case when set-sharing is combined with other kinds of information, such as the popular freeness.

1 Introduction

Program analysis is the process of inferring at compile–time inferring information about run–time properties of programs. In logic programs one of the most studied run-time properties is *sharing* among program variables. Two program variables share in a given run-time store if the terms to which they are bound have at least one run-time variable in common. A set of program variables share if they all have at least one run-time variable in common. The former kind of sharing is called *pair-sharing* while the latter is called *set-sharing*. Any of the two may be target observables of an analysis.

The importance (and hence popularity) of sharing comes from two sources. First, sharing information is in itself vital for several applications such as exploitation of independent AND-parallelism [18,5], occurs check reduction [26,27], and compile-time garbage collection [23]. And second, sharing can be used to accurately keep track of other interesting run-time properties such as *freeness* (a program variable is free in a run-time store if it is either unbound or bound to a run-time variable).

Sharing analysis has therefore raised an enormous amount of interest in our research community, with many different analysis domains being proposed in the literature (see e.g., [27,17,25,3,19]). Two of the best known sharing analysis domains are `ASub` defined by Søndergaard [27] and `Sharing` defined by Jacobs and Langen [17,18]. The main difference between these two domains is the way in which they represent sharing information: while `ASub` keeps track of *pairs* of program variables that possibly share, `Sharing` keeps track of *sets* of program variables that possibly share certain variable occurrences.

© Springer-Verlag Berlin Heidelberg 2004

These differences have subtle consequences. On the one hand, the pair sharing encoding in ASub allows it to keep track of linear program variables (a program variable is *linear* in a run-time store if it is bound to a term which does not have multiple occurrences of the same run-time variable). Linearity information, in turn, allows ASub to improve the accuracy of the abstract sharing operations. On the other hand, the set sharing encoding in Sharing allows it to represent several other kinds of information (such as groundness and sharing dependencies) which also result in more accurate abstract operations. In fact, when combined with linearity, Sharing is strictly more accurate than ASub. In practice, this accuracy improvement has proved to be significant [7].

As a result, Sharing became the standard choice for sharing analysis, usually combined with other kinds of information such as freeness or structural information, even though its complexity can have significant impact on efficiency. However, the benefits of using set sharing for sharing analysis have been recently questioned (see [10,1,2]). As a paradigm of the case, we cite the title of a paper by Bagnara, Hill, and Zaffanella: "Set-Sharing is redundant for Pair-Sharing" [1,2]. In this paper, the authors state the following

> **Assumption:** The goal of sharing analysis for logic programs is to detect which *pairs* of variables are definitely independent (namely they cannot be bound to terms having one or more variables in common).
>
> As far as we know this assumption is true. In the literature we can find no reference to the "independence of a *set* of variables". All the proposed applications of sharing analysis (compile-time optimizations, occur-check reduction and so on) are based on information about the independence of *pairs* of variables.

Based on the above assumption, the authors focus on defining a simpler version of Sharing which is however as precise as far as pair-sharing is concerned. This new simpler domain, referred to in the future as SS^ρ, is obtained by eliminating from Sharing information which is considered "redundant" w.r.t. the pair-sharing property. This elimination allows further simplification of the abstract operations in SS^ρ which can significantly improve its efficiency.

The popularity of the Sharing domain combined with the great accuracy and efficiency results obtained for SS^ρ (and the clarity with which the authors explained the intricacies of the Sharing domain), ensured the paper had a significant impact on the community, with many researchers now accepting that set-sharing is indeed redundant for pair-sharing (see, e.g., [20,8,22,21]).

The aim of this paper is to prove that this is not always the case. In particular, we will show that: (1) There exist applications which use set-sharing analysis (combined with freeness) to infer properties other than sharing between pairs of variables; and (2) When combined with information capable of distinguishing among the different variable occurrences represented by Sharing, this domain can yield results not obtainable with SS^ρ, *including better pair-sharing*. Such a combination is found in at least two common situations: when Sharing is used as a carrier for other analyses (such as freeness), and when the analysis process is improved with extra information (such as in-lined knowledge of the semantics of some predicates, for example builtins). Possible approaches to combine SS^ρ with other kinds of information without losing accuracy are also suggested.

We believe our insights will contribute to the better understanding of an abstract domain which, while being one of the most popular and more intensively studied abstract domains ever defined, remains somewhat misunderstood.

2 Preliminaries

Let us start by introducing our notation as well as the basics of the Sharing domain [17,18]. In doing this we will mainly follow the extremely clear summary presented in [1]. Given a set S, $\wp(S)$ denotes the powerset of S, and $\wp_f(S)$ denotes the set of all the finite subsets of S. $\mathcal{V}$ denotes a denumerable set of variables, $Var \in \wp_f(\mathcal{V})$ denotes a finite set of variables, called the *variables of interest* (e.g., the variables of a program). The set of variables in a syntactic object o is denoted $vars(o)$. $\mathcal{T}_\mathcal{V}$ is the set of first order terms over $\mathcal{V}$. A substitution θ is a mapping $\theta : \mathcal{V} \rightarrow \mathcal{T}_\mathcal{V}$, whose application to variable x is denoted by $x\theta$. Substitutions are denoted by the set of their bindings: $\theta = \{x \mapsto x\theta \mid x\theta \neq x\}$. We define the image of a substitution θ as the set $img(\theta) \stackrel{\text{def}}{=} \bigcup\{vars(x\theta) \mid x \in Var\}$.

The Sharing domain is formally defined as follows. Let $SH \stackrel{\text{def}}{=} \wp(SG)$, where $SG \stackrel{\text{def}}{=} \{S \subseteq Var \mid S \neq \emptyset\}$. Each element $S \in SG$ is called a *sharing set*. We will write sharing sets as strings with the variables that belong to it, e.g., sharing set $\{x, y, z\}$ will be denoted xyz. A sharing set of size 2 is called a *sharing pair*.

The function $occ(\theta, v)$ obtains a sharing set that represents the occurrence of variable v through the variables of interest as per the substitution θ.

$$occ(\theta, v) \stackrel{\text{def}}{=} \{x \in Var \mid v \in vars(x\theta)\}$$

The abstraction of a substitution θ is obtained by computing all relevant sharing sets: $\alpha(\theta) \stackrel{\text{def}}{=} \{occ(\theta, v) \mid v \in img(\theta)\}$.

Abstract element $sh \in SH$ approximates substitution θ iff $\alpha(\theta) \subseteq sh$. Conversely, the concretization of $sh \in SH$ is the set of all substitutions approximated by sh. Projection over a set $V \subseteq Var$ is given by

$$proj(sh, V) \stackrel{\text{def}}{=} \{S \cap V \mid S \in sh[V]\}$$

where, for any syntactic object o and abstraction $sh \in SH$,

$$sh[o] \stackrel{\text{def}}{=} \{S \in sh \mid S \cap vars(o) \neq \emptyset\}.$$

The pairwise (or binary) union of two abstractions is defined as:

$$sh_1 \uplus sh_2 \stackrel{\text{def}}{=} \{S_1 \cup S_2 \mid S_1 \in sh_1, S_2 \in sh_2\}.$$

The closure under (or star) union of an abstract element sh is defined as the least set $sh*$ that satisfies:

$$sh^* = sh \cup \{S_1 \cup S_2 \mid S_1, S_2 \in sh^*\}.$$

Abstract unification for a substitution θ is given by extending to the set of bindings of θ the following abstract unification operation for a binding:

$$amgu(sh, x \mapsto t) = (sh \setminus (sh[x] \cup sh[t])) \cup (sh[x]^* \uplus sh[t]^*).$$

The set-sharing lattice is thus given by the set

$$SS \stackrel{\text{def}}{=} \{(sh, U) | sh \in SH, U \subseteq Var, \forall S \in sh : S \subseteq U\} \cup \{\bot, \top\}$$

which is a complete lattice ordered by $\leq_{SS}$ defined as follows. For elements $\{d, (sh_1, U_1), (sh_2, U_2)\} \subseteq SS$:

$$\begin{array}{l} \bot \leq_{SS} d \\ d \leq_{SS} \top \\ (sh_1, U_1) \leq_{SS} (sh_2, U_2) \text{ iff } U_1 = U_2 \text{ and } sh_1 \subseteq sh_2. \end{array}$$

The lifting of $\cup$, *proj,* and *amgu* defined over *SH* to define the abstract operations $\sqcup$, *Proj,* and *Amgu* over *SS* is straightforward.

Example 1. Let $Var = \{x, y, z\}$ be the set of variables of interest and consider the substitutions $\theta_1 = \{x \mapsto f(u, u, v), y \mapsto g(u, v, w, o), z \mapsto h(u)\}$ and $\theta_2 = \{x \mapsto u, y \mapsto u, z \mapsto 1\}$. Then, $sh_1 = \alpha(\theta_1) = \{xy, xyz, y\}$, where sharing set xyz represents the occurrence of variable u in x, y and z, sharing set xy represents the occurrence of variable v in x and y, and sharing set y represents the occurrence of variables w and o in y. Similarly, we have that $sh_2 = \alpha(\theta_2) = \{xy\}$ where sharing set xy represents the occurrence of variable u in x and y. Let $U = Var$. We then have that $(sh_2, U) \leq_{SS} (sh_1, U)$ and thus $(sh_1, U) \sqcup (sh_2, U) = (sh_1, U)$. Finally, let $V = \{x, y\}$, $Proj((sh_1, U), V) = (\{xy, y\}, V)$. Note that the sharing set xy in the projected abstraction represents not only the occurrence of variable u but also that of v. $\diamond$

3 Eliminating Redundancy from `Sharing`

One of the main insights in [10,1] regarding the `Sharing` domain is the detection of sets which are redundant (and can thus be safely eliminated or not produced) as far as pair-sharing is concerned. Given an element *sh* of *SH,* sharing set $S \in sh$ is *redundant* w.r.t. pair sharing if and only if all its sharing pairs can be extracted from other subsets of *S* which also appear in *sh.* Formally, let $pairs(S) \stackrel{\text{def}}{=} \{xy \mid x, y \in S, x \neq y\}$. Then, *S* is redundant iff

$$pairs(S) = \bigcup \{pairs(T) \mid T \in sh, T \subset S\}$$

Example 2. Consider the abstraction $sh = \{xy, xz, yz, xyz\}$ defined over $Var = \{x, y, z\}$. It is easy to see that set $xyz \in sh$ is redundant w.r.t. pair sharing. $\diamond$

Based on this insight, a closure operator, $\rho : SH \rightarrow SH$, is defined in [1] to add to each $sh \in SH$ the set of elements which are redundant for sh. Formally:

$$\rho(sh) \stackrel{\text{def}}{=} \{S \in SG \mid \forall x \in S : S \in sh[x]^*\}.$$

This function is then used to define a new domain SS^ρ which is the quotient of *SS* w.r.t. the new equivalence relation induced by ρ: elements d_1 and d_2 are equivalent iff $\rho(d_1) = \rho(d_2)$. The authors prove that (a) the addition of redundant elements does not cause any precision loss as far as pair-sharing is concerned, i.e., that SS^ρ is as good as *SS* at representing pair-sharing, and that (b) ρ is a congruence w.r.t. the abstract operations *Amgu,* $\sqcup$ and *Proj.* Thus, they conclude that SS^ρ is as good as *SS* also for propagating pair-sharing through the analysis process.

The above insight is used by [1] to perform two major changes to the `Sharing` domain. Firstly, redundant elements can be eliminated (although experimental results suggest that this is not always advantageous). And secondly, addition of redundant elements can be avoided by replacing the star union with the binary union operation without loss of accuracy. This is a very important change since it can have significant impact on efficiency by simplifying one of the most expensive abstract operations in `Sharing`.

The results obtained in [1] are indeed interesting and can be very useful in some contexts. However, there are situations in which the lack of redundant sets can lead to loss of accuracy w.r.t. pair sharing, and even incorrect results if the full expressive power of `Sharing` is assumed to be still present in SS^ρ.

Example 3. Consider the abstractions $sh_1 = \{x, y, z, xy, xz, yz\}$ and $sh_2 = \{x, y, z, xy, xz, yz, xyz\}$ defined over $Var = \{x, y, z\}$, and note that $\rho(sh_1) = sh_2$, i.e., the sharing set *xyz* is redundant for sh_2.

Consider the Prolog builtin $x == y$ which succeeds if program variables x and y are bound at run-time to identical terms. A sophisticated implementation of the `Sharing` domain (such as that of [4]) could take advantage of this information and eliminate every single sharing set in which the program variables x and y appear but not together (since all variables which occur in x must also occur in y, and vice versa). Thus, correct and precise abstractions of a situation in which the builtin was successfully executed in stores represented by sh_1 and sh_2, will become $sh'_1 = \{z, xy\}$ and $sh'_2 = \{z, xy, xyz\}$, respectively. However, it is easy to see that $pairs(sh'_1) \neq pairs(sh'_2)$, since z is definitely independent of both x and y in sh'_1 while it might still share with them in sh'_2. $\diamond$

The above example shows that `Sharing` can make use of the information provided by other sources in order to improve the pair-sharing accuracy of its elements, while the same action might lead to incorrect results for elements of SS^ρ if redundant sharing sets had actually been eliminated from those elements. As we will see in the following sections, this can happen when using information coming not only from builtins, but also from other domains (such as freeness) which are usually combined with set-sharing. Furthermore, useful information other than sharing can be inferred from combinations of `Sharing` and other sources which are not possible with SS^ρ.

4 When Redundant Sets Are No Longer Redundant

The problem illustrated in the previous example is rooted in the always surprising complexity of the information encoded by elements of *SH.* As indicated by [1,2], elements of *SH* can encode definite groundness (e.g., x is ground), groundness dependencies (e.g., if x becomes ground then y is ground), and sharing dependencies.[3] However, as we will see in this section, these are only by-products of the main property represented by elements of *SH:* the different variable occurrences shared by each set of program variables.

The groundness of variable x, and the sharing independence between variables x and y (i.e., the fact that x and y are known not to share) can be expressed by an element $sh \in SH$ as follows:

$$\begin{aligned} ground(x) &\text{ iff } \forall S \in sh : \; x \notin S \\ indep(x,y) &\text{ iff } \forall S \in sh : xy \not\subseteq S \end{aligned}$$

where $ground(x)$ represents the fact that variable x is ground in all substitutions abstracted by *sh,* and $indep(x,y)$ represents the fact that variables x and y do not share in any substitution abstracted by $sh \in SH$.

Groundness dependencies in $sh \in SH$ can be easily obtained from the above statements in the following way. Let us assume that x is known to be ground. We can then modify *sh* by enforcing $\forall S \in sh : \; x \notin S$ to hold, i.e., by eliminating every $S \in sh$ such that $x \in S$. If we can then prove that the same statement holds for some other variable y, we would then know that the implication $ground(x) \rightarrow ground(y)$ holds for *sh.* This simply illustrates the well known result that `Sharing` subsumes the groundness dependency domain *Def.* The same method can be used for obtaining other dependencies for elements *sh* of *SH.* The following were used in [5] for simplifying parallelization tests:

1. $ground(x_1) \wedge \ldots \wedge ground(x_n) \rightarrow ground(y)$ if
   $\forall S \in sh : \text{ if } y \in S \text{ then } \{x_1, \ldots, x_n\} \cap S \neq \emptyset$
2. $ground(x_1) \wedge \ldots \wedge ground(x_n) \rightarrow indep(y,z)$ if
   $\forall S \in sh : \text{ if } \{y,z\} \subseteq S \text{ then } \{x_1, \ldots, x_n\} \cap S \neq \emptyset$
3. $indep(x_1,y_1) \wedge \ldots \wedge indep(x_n,y_n) \rightarrow ground(z)$ if
   $\forall S \in sh : \text{ if } z \in S \text{ then } \exists j \in [1,n], \; \{x_j, y_j\} \subseteq S$
4. $indep(x_1,y_1) \wedge \ldots \wedge indep(x_n,y_n) \rightarrow indep(w,z)$ if
   $\forall S \in sh : \text{ if } \{w,z\} \subseteq S \text{ then } \exists j \in [1,n], \; \{x_j, y_j\} \subseteq S$

Let us now characterize in a similar way the (non-symmetrical) property $covers(x,y)$ expressed by an element $sh \in SH$ as follows:

$$covers(x,y) \text{ iff } \forall S \in sh : \text{ if } y \in S \text{ then } x \in S$$

where $covers(x,y)$ indicates that variable y shares all its variables with variable x and, therefore, every sharing set in which y appears must also contain x. We can now derive other sharing dependencies for any $sh \in SH$, such as:

5. $covers(x_1,y_1) \wedge \ldots \wedge covers(x_n,y_n) \rightarrow ground(z)$ if
   $\forall S \in sh : \text{ if } z \in S \text{ then } \exists j \in [1,n], \; y_j \in S, \; x_j \notin S$
6. $covers(x_1,y_1) \wedge \ldots \wedge covers(x_n,y_n) \rightarrow indep(w,z)$ if
   $\forall S \in sh : \text{ if } \{w,z\} \subseteq S \text{ then } \exists j \in [1,n], \; y_j \in S, \; x_j \notin S$

[3] The fact that it also encodes independence (e.g., x does not share with y) was probably obviated because this is also encoded by pair-sharing.

7. $covers(x_1, y_1) \wedge \ldots \wedge covers(x_n, y_n) \rightarrow covers(w, z)$ if
$\forall S \in sh$: if $z \in S, w \notin S$ then $\exists j \in [1, n]$, $y_j \in S$, $x_j \notin S$

It is important to note that while the expressions with only $ground(x)$ and $indep(x, y)$ elements can also hold for any element of SS^ρ, this is not true for the expressions with coverage information.

Example 4. Consider again the abstractions introduced by Example 3, $sh_1 = \{x, y, z, xy, xz, yz\}$ and $sh_2 = \{x, y, z, xy, xz, yz, xyz\}$ which are defined over $Var = \{x, y, z\}$. Let us assume that both abstractions belong to `Sharing`. While implication $covers(x, y) \wedge covers(y, x) \rightarrow indep(x, z)$ holds for sh_1, it does not hold for sh_2. If we now consider the SS^ρ domain, both abstractions would be represented by the element sh_1. Therefore, the implication should not hold for sh_1 in SS^ρ. ◇

In order to understand why, consider the differences between the expressions $ground(x)$ iff $\forall S \in sh : x \notin S$, and $indep(x, y)$ iff $\forall S \in sh : xy \not\subseteq S$, and the expression $covers(x, y)$ iff $\forall S \in sh$: if $y \in S$ then $x \in S$. While in the first two the sharing sets which violate the right hand side of the expressions would always include the redundant set (if any), those which violate the last expression would not. Thus, to assume coverage might result in the subset of a redundant set being eliminated without the redundant set itself being eliminated. In this way sharing sets which are considered redundant at some point, might become non redundant once coverage information is added and, therefore, their elimination (or non generation) can lead to incorrect information. For example, consider the substitution *sh* = {*xyz,xy,xz,yz*}. While the problematic sets for $ground(x)$ and $indep(x, y)$ in *sh* are *xyz, xy, xz* and *xyz, xy,* respectively, the only one for $covers(x, y)$ is *yz*. But once *yz* is removed from *sh, xyz* is no longer redundant: it is the only sharing set able (when x covers y) to represent the possible sharing between x and y.

As a result, sharing sets initially redundant for pair-sharing can prove useful whenever combined with other sources of information (coming from builtins, other analysis domains, etc.) capable of distinguishing between the variable occurrences represented by the redundant sharing sets and the variable occurrences represented by their subsets, so that, once the extra information is added, a sharing set previously identified as redundant will no longer be so.

5 Combining `Sharing` with Freeness

In this section we will use the popular combination of `Sharing` with freeness information to illustrate two points. First, that very common sources of information (such as freeness) can distinguish between variable occurrences, an ability which can be exploited in ways that can make a redundant set no longer redundant. Thus, it can be advantageous not to eliminate them. And second, that the goal of sharing analysis for logic programs is not only to detect which pairs of variables are definitely independent, but also to detect (or propagate) many other kinds of information.

In order to illustrate these points we will use the notion of *active* sharing sets [6]. A sharing set $S \in sh$ is said to be *active* for store $c \in \gamma(sh)$ iff $S \in \alpha(c)$. All sharing sets $\{S_1, \cdots, S_n\} \subseteq sh$ are said to be active *at the same time* if there exists a store $c \in \gamma(sh)$ such that $\forall 1 \leq i \leq n, S_i \in \alpha(c)$. If only the information in Sharing is taken into account, then all sharing sets in any $sh \in SH$ can be active at the same time.

Example 5. Consider the set-sharing abstraction $sh = \{x, xy, yz\}$ defined over $Var = \{x, y, z\}$. All sets in *sh* can be active at the same time since there exists a store, say $\theta = \{x = f(u, v), y = f(v, w), z = f(w)\}$, such that $\alpha(\theta) = sh$. In particular, u is the variable represented by sharing set x, v is represented by *xy*, and w is represented by *yz*. ◇

However, this is not always the case when considering information outside the scope of Sharing. In some cases, two or more sharing sets cannot be active at the same time since, thanks to some extra information, we can determine that these sharing sets must represent the same variable(s) occurrence.

Example 6. Consider again the set-sharing abstraction $sh = \{x, xy, yz\}$ defined over $Var = \{x, y, z\}$, and let us now assume y and z are known to be free variables. As pointed out in [6], since each sharing set in an abstraction represents a different occurrence of one or more variables, no two sharing sets containing the same free variable can be active at the same time (the same variable cannot be a different occurrence). In our example, *xy* and *yz* cannot be active at the same time since there is no concrete store with both y and z free, such that both share a variable not shared with anyone else (sharing set *yz*) and y also shares a different variable with x (sharing set *xy*). ◇

Knowing which sharing sets in abstraction *sh* can be active at the same time according to Ω is useful because we can use thois notion to divide *sh* into $\{sh_1, \cdots, sh_n\}$ such that $sh = sh_1 \cup \ldots \cup sh_n$, $\forall i, 1 \leq i \leq n$ all sets in sh_i can be active at the same time, and $\neg \exists j, 1 \leq j \leq n : j \neq i, sh_j \subseteq sh_i$.

Example 7. Consider again the abstraction $sh = \{x, xy, yz\}$ defined over $Var = \{x, y, z\}$. If y and z are known to be free variables, *sh* can be divided into two different sets, $\{x, xy\}$ and $\{x, yz\}$, whose sharing sets can all be active at the same time. The former represents the concrete stores in which x definitely shares a variable with y (which is actually known to be y itself), and x might also have some variable which is not shared with anyone else. The latter represents the stores in which the free variables y and z are aliased and x might have some variables which are not shared with anyone else. ◇

Note that the different sh_i together with Ω describe disjoints sets of concrete stores. Furthermore, even though $(\bigcup_i \gamma(sh_i)) \cap \gamma(\Omega)$ is still equivalent to $\gamma(sh) \cap \gamma(\Omega)$ (which justifies the correctness of dividing *sh* into the different sh_i in the presence of Ω), it is often the case that $\bigcup_i \gamma(sh_i) \subset \gamma(sh)$, as it happens in the above example. As a result, it is generally easier to understand the concretization of *sh* and Ω by means of the concretization of each sh_i and Ω. Let us use this to show how the direct-product domain [11] of Sharing and freeness can be used to improve pair-sharing.

Example 8. Consider the abstraction $sh = \{xy, xz, yz, xyz\}$ defined over program variables x, y and z. If we knew that x, y, and z are free we could divide sh into the sets $sh_1 = \{xy\}, sh_2 = \{xz\}, sh_3 = \{yz\}$ and $sh_4 = \{xyz\}$.Now, sh_1 represents stores in which z is known to be ground, which is not true according to our freeness information. Thus, its sharing sets (xy) can be eliminated from sh. The same reasoning applies to sh_2 and sh_3. Thus, sh can be simplified to $\{xyz\}$ indicating that all variables definitely share (which of course also implies their definite pair-sharing dependencies). Note that if the set xyz did not belong to the abstraction, the concretization of sh in the context of freeness would be empty (indicating a failure in the program). $\diamond$

The above example shows how the direct-product domain of SS^ρ and freeness might be incorrect if the full power of set-sharing is assumed to be still present in SS^ρ. This occurs whenever a redundant set is known to contain a free variable, since it would then appear in an sh_i without one or more of its subsets. Thus, the set would no longer be redundant for sh_i. A simple solution would be to behave as if redundant sets containing free variables were present in the SS^ρ abstractions even if they do not appear explicitly in them. It would be easy to think that such solution does not lose accuracy w.r.t. pair sharing. This is, however, not true.

Example 9. Consider the set-sharing abstraction $sh = \{xy, xz, yz\}$ defined over $Var = \{x, y, z\}$. If we knew that y and z were free, we could divide sh into the sets $sh_1 = \{xy, xz\}$ and $sh_2 = \{yz\}$, respectively representing the concrete stores in which x shares with y and z, which do not share among them, and those in which x does not share with anyone and y shares with z. Note that these two situations are mutually exclusive. This allow us to prove (among others) that:

$$indep(y, z) \text{ iff } \neg indep(x, y) \text{ and } indep(y, z) \text{ iff } \neg indep(x, z).$$

This is crucial pair-sharing information (e.g., for automatic AND-parallelization, as we will see in the next section). If the redundant set xyz could have been eliminated from sh, the above expression might not hold, since the variables might then be aliased to the same free variable, thus capturing also the case in which all of them are definitely dependent of each other. $\diamond$

Let us now show how combining Sharing and freeness information, as done for example in Sharing+Freeness [25], yields interesting kinds of information other than the sharing itself, information which is the goal of such analyses for several applications.

Example 10. Consider again the set-sharing abstraction $sh = \{xy, xz, yz\}$ defined over $Var = \{x, y, z\}$. As mentioned above, if we knew that y and z were free, we could divide sh into the sets $sh_1 = \{xy, xz\}$ and $sh_2 = \{yz\}$. The concrete stores represented by these sets can in fact be described much more accurately than we did in the previous example: While sh_1 represents stores in which x is bound to a term with two (and only two) non-aliased free variables

(y and z), sh_2 represents those stores in which x is ground, and y and z are free aliased variables. As a result, we can be sure *sh* only represents stores in which x is bound to a non-variable term. $\diamond$

Definite information about non-variable bindings is used, for example, to determine whether dynamic scheduled goals waiting for a program variable to become non-variable can be woken up, as performed by [15]. However, such information cannot be obtained if redundant sets containing free variables are eliminated.

Example 11. Consider the set-sharing abstractions $sh = \{xy, xz, yz\}$ above and $sh' = sh \cup \{xyz\}$ where y and z are known to be free, we could divide sh' into the sets $sh_1 = \{xy, xz\}$ and $sh_2 = \{yz\}$ and $sh_3 = \{xyz\}$. The first two are as above, while the third represents stores in which all x, y and z share the same variables (with x possibly being a free variable). Thus, sh' does not only represent stores in which x is bound to a non-variable term. $\diamond$

Definite knowledge about non-variable bindings is not the only kind of useful information that can be inferred from combining `Sharing` and freeness. The combination can also be used to detect new bindings added by some body literal.

Example 12. Consider again the set-sharing abstraction $sh = \{xy, xz, yz\}$ where y and z are known to be free. Let us assume that *sh* is the abstract call for body literal $p(x, y, z)$ (i.e., the abstraction at the program point right before executing the literal) and that $sh' = \{xy, xz, yz, xyz\}$ is the abstract answer for $p(x, y, z)$ (i.e., the abstraction at the program point right after executing the literal) with y and z still known to be free. The addition of sharing set *xyz* means that a new binding aliasing y and z might have been introduced by $p(x, y, z)$. However, if the abstract answer is found to be identical to the call *sh,* we can be sure that none of the three program variables has been further instantiated (since they are still known to be free) nor any new aliasing introduced among them. $\diamond$

The above kind of information is used, for example, for detecting non-strict independence [6] as we will see in the next section. As shown in the above example, this information cannot be inferred if redundant sets might have been eliminated (or not produced).

6 When Independence among Sets Is Relevant

This section uses the well-known application of automatic parallelization within the independent AND-parallelism model [9] to illustrate how some applications (a) require independence among sets (as opposed to pairs) of variables, and (b) can benefit from combining `Sharing` with freeness information in ways which would not be possible with SS^ρ. The relevance of this application comes from the fact that it is not only one of the best known applications of sharing information, but also the one for which the `Sharing` domain was developed.

In the independent AND-parallelism model goals g_1 and g_2 in the sequence g_1, g_2 can be run in parallel in constraint store c if g_2 is independent of g_1 for store c. In this context, independence refers to the conditions that the run-time behavior of these goals must satisfy in order to guarantee the correctness and efficiency of their parallelization w.r.t. their sequential execution. This can be expressed as follows: goal g_2 is independent of goal g_1 for store c iff the execution of g_2 in c has the same number of computation steps, cost, and answers as that of g_2 in any store c' obtained from executing g_1 in c.

Note that the general independence condition introduced above is thus neither symmetric nor established between pairs of variables, as assumed by [1,2]. However, this general notion of independence is indeed rarely used. Instead, sufficient (and thus simpler) conditions are generally used to ensure independence. These conditions can be divided into two main groups: a priori and a posteriori. A priori conditions can always be checked prior to the execution of the goals involved, while a posteriori conditions can be based on the actual behaviour of the goals to be run in parallel.

A priori conditions are more popular even though they can be less accurate. The reasons are twofold. First, they can only be based on the characteristics of the store c and the variables belonging to the goals to be run in parallel. Thus, they are relatively simple. And second, they can be used as run-time tests without actually running the goals themselves. This is useful whenever the conditions cannot be proved correct at compile-time. Note that a priori conditions must be symmetric: goals g_1 and g_2 are independent for c iff g_1 is independent of g_2 for c *and* g_2 is independent of g_1 for c.

The most general a priori condition, called *projection independence,* was defined in [14] as follows: goals g_1 and g_2 are independent for c if for any variable $x \in vars(g_1) \cap vars(g_2)$, x is uniquely defined by c (i.e., ground), and the constraint obtained by conjoining the projection of c over $vars(g_1)$ and the projection of c over $vars(g_2)$ entails (i.e., logically implies) the constraint obtained by projecting c over $vars(g_1) \cup vars(g_2)$.

Example 13. Consider the literals $p(x), q(y), r(z)$ and constraint $c \equiv \{x = y + z\}$. The projection of c over the sets of variables containing either one or two variables from $\{x, y, z\}$ is the empty constraint *true.* Thus, we can ensure that every pair of literals, say $p(x)$ and $q(y)$, can run in parallel. However, no literal can run in parallel with the goal formed by the conjunction of the other two literals, e.g., $p(x)$ cannot run in parallel with goal $q(y), r(z)$, since the projection of c over $\{x, y, z\}$ is c itself, which is indeed not entailed by *true.* $\diamond$

Therefore, as mentioned in both [24] and [13], in general projection independence does indeed rely on the independence of a pair of *sets* of variables. However, for the Herbrand case projection independence is equivalent to the better known a priori condition called *strict independence,* which was introduced in [9,12] and formally defined and proved correct in [16]. It states that goals g_1 and g_2 are strictly independent for substitution θ iff $vars(g_1)$ do not share with $vars(g_2)$ for θ, i.e., iff $vars(g_1\theta) \cap vars(g_2\theta) = \emptyset$. It is easy to prove that this is equivalent to requiring that for every pair of variables $xy, x \in vars(g_1), y \in vars(g_2)$, x and y do not share.

Therefore, only for a priori conditions and the Herbrand domain, is parallelization based on the independence of pairs of variables. And even in this case, the `Sharing` domain is more powerful than SS^ρ when combined with other kinds of information.

Example 14. Consider again the abstractions $sh = \{xy, xz, yz, xyz\}$ and $sh' = \{xy, xz, yz\}$ defined over $Var = \{x, y, z\}$. Example 9 illustrated how the formula

$$indep(y, z) \text{ iff } \neg indep(x, y) \text{ and } indep(y, z) \text{ iff } \neg indep(x, z)$$

hold for sh' but not for sh when y and z are known to be free.

Consider the automatic parallelization of sequential goal `p(y),q(z),r(x)` for the usual case of the a priori condition strict independence and the Herbrand domain. In the absence of any information regarding the state of the store occurring right before the sequential goal is executed, the compiler could rewrite the sequential goal into the following parallel goal (leftmost column):

```
( indep(y,z) ->              | ( indep(y,z) ->       | ( indep(y,z) ->
  ( indep(x,y) ->            |                       |
    ( indep(x,z) ->          |                       |
      p(y)&q(z)&r(x)         |                       |
    ; p(y)&(q(z),r(x))       |                       |
    )                        |                       |
  ; (p(y)&q(z)),r(x)         |   (p(y)&q(z)),r(x)    |   (p(y)&q(z)),r(x)
  )                          |                       |
; indep(x,z) ->              | ;                     | ; indep(x,z) ->
  p(y),(q(z)&r(x))           |   p(y),(q(z)&r(x))    |   p(y),(q(z)&r(x))
; p(y),q(z),r(x)             |                       | ; p(y),q(z),r(x)
)                            | )                     | )
```

where the operator & represents parallel execution of two goals, and the run-time test `indep(x,y)` succeeds if the two variables do not share at run-time. The middle and right columns represent the simplifications that can be performed to the parallel goal in the context of sh' and sh, respectively. This is because while test `indep(x, y)` is known to fail if `indep(y, z)` succeeds for both sh and sh', test `indep(x, z)` is known to succeed if `indep(y, z)` fails for sh' but not for sh. Thus, `indep(x, z)` still needs to be tested at run-time with the resulting loss of efficiency. ◇

The assumption is also incorrect when considering a posteriori conditions, even those associated to the Herbrand domain. In particular, strict independence has been generalised to several different [16] a posteriori notions of *non-strict independence.* These notions allow goals that share variables to run in parallel as long as the bindings established for those shared variables satisfy certain conditions. For example, one of the simpler notions only allows g_1 to instantiate a shared variable and does not allow any aliasing (of different shared variables) to be created during the execution of g_1 that might affect goals to the right. Thus, for this notion, the conditions are established between the *bindings* introduced by the two goals over their respective set of variables, and cannot be expressed using only sharing between pairs of variables.

There has been at least one attempt [6] at inferring non-strict independence at compile-time using the abstract domain Sharing+Freeness. The inference is based on two conditions. The first ensures that (C1) no shared variables are further instantiated by g_1. This is done by requiring that (a) all shared variables share through variables known to be free in the abstract call of g_1 (all sharing sets in the abstract call containing shared variables also contain a free variable), and (b) all these variables must remain free in the abstract answer of g_1 (all such sharing sets still contain a free variable after the analysis of g_1). This first condition can be detected in the SS^ρ domain since the existence of a free variable in every sharing pair ensures the existence of a free variable in the "redundant" sharing set. Thus, the absence of such sharing set is not a problem.

This is not however the case for the second condition, which ensures that no aliasing is introduced among shared variables by requiring C1 and, additionally, that (C2) there is no introduction in the abstract answer of any sharing set resulting from the union of several sets such that none contain the same free variable, and at least two contain variables belonging to both goals.

Example 15. Consider again the set-sharing abstraction $sh = \{xy, xz, yz\}$ where y and z are known to be free. Let us assume that sh is the abstract call for body $p(x,y,z), q(x,y,z)$ and that $sh' = \{xy, xz, yz, xyz\}$ is the abstract answer for $p(x,y,z)$ with y and z still known to be free. All sharing sets in sh containing variables from both literals contain a free variable which remains free in sh'. Thus, C1 is satisfied. However, there exists a set xyz in sh' which can be obtained by unioning at least two sets xy and xz in sh which contain variables from both literals and have no variable in common known to be free in sh. The appearance of such a set represents the possible aliasing of y and z by $p(x,y,z)$. This appearance violates C2 and thus the goals cannot run in parallel. Note that if the abstract answer was found to be identical to sh (i.e., if the redundant set xyz was absent), we would have been able to ensure that none of the three program variables had been further instantiated nor any new aliasing introduced among them. Therefore, we could have ensured that g_2 is independent of g_1 for the stores represented by sh and the associated freeness information, thus allowing their parallel execution. $\diamond$

The above example illustrates the fact that an equivalent inference cannot be performed in the SS^ρ domain augmented with freeness *unless care is taken when considering redundant sharing sets which include program variables known to be free.* This is because the inference strongly depends on distinguishing between the different bindings introduced during execution of the goals to be run in parallel, and as a result, on distinguishing between the different shared variables represented by the abstractions in the domain. Thus, elimination of redundant sets can render the method incorrect. One possible solution is to always assume that redundant sets containing free variables are present when combining SS^ρ with freeness information. However, as shown in Example 9, this might be imprecise. Another, more accurate solution, is to only eliminate redundant sets which do not contain variables known to be free.

7 Conclusion

We have shown that the power of set-sharing does not come from representing sets of variables that share, but from representing different variable occurrences. As a result, eliminating from `Sharing` information which is considered "redundant" w.r.t. the pair-sharing property as performed in SS^ρ can have unexpected consequences. In particular, when `Sharing` is combined with some other kinds of information capable of distinguishing among variable occurrences in a way that can make a redundant set no longer redundant, it can yield results not obtainable with SS^ρ, *including better pair-sharing.* Furthermore, there exist applications which use `Sharing` analysis (combined with freeness) to infer properties other than sharing between pairs of variables and which cannot be inferred if SS^ρ is used instead. We have proposed some possible solutions to this problem.

Acknowledgments

This work was funded in part by projects ASAP (EU IST FET Programme Project Number IST-2001-38059) and CUBICO (MCYT TIC 2002-0055). It was also supported by ARC IREX grant X00106666.

References

1. R. Bagnara, P. M. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. In *Static Analysis Symposium,* pages 53–67. Springer-Verlag, 1997.
2. R. Bagnara, P. M. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. *Theoretical Computer Science,* 277(1-2):3–46, 2002.
3. M. Bruynooghe and M. Codish. Freeness, sharing, linearity and correctness — all at once. In *International Workshop on Static Analysis,* 1993.
4. F. Bueno, M. Garcia de la Banda, and M. Hermenegildo. The PLAI Abstract Interpretation System. Technical Report CLIP2/94.0, Computer Science Dept., Technical U. of Madrid (UPM), February 1994.
5. F. Bueno, M. Garcia de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems,* 21(2):189–238, 1999.
6. D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *1994 International Static Analysis Symposium,* number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994. Springer-Verlag.
7. M. Codish, A. Mulkers, M. Bruynooghe, M. Garcia de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems,* 17(1):28–44, January 1995.
8. Michael Codish, Harald Søndergaard, and Peter J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems,* 21(5):948–976, 1999.
9. J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

10. A. Cortesi, G. Filé, and W. Winsborough. The quotient of an abstract interpretation for comparing static analyses. In *GULP-PRODE'94 Joint Conference on Declarative Programming,* pages 372–397, 1994.
11. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Sixth ACM Symposium on Principles of Programming Languages,* pages 269–282, San Antonio, Texas, 1979.
12. D. DeGroot. A Technique for Compiling Execution Graph Expressions for Restricted AND-parallelism in Logic Programs. In *Int'l Supercomputing Conference,* pages 80–89. Springer Verlag, 1987.
13. M. Garcia de la Banda, F. Bueno, and M. Hermenegildo. Towards Independent And-Parallelism in CLP. In *Programming Languages: Implementation, Logics, and Programs,* number 1140 in LNCS, pages 77–91. Springer-Verlag, September 1996.
14. M. Garcia de la Banda, M. Hermenegildo, and K. Marriott. Independence in CLP Languages. *ACM Transactions on Programming Languages and Systems,* 22(2):296–339, 2000.
15. M. Garcia de la Banda, K. Marriott, and P. Stuckey. Efficient Analysis of Constraint Logic Programs with Dynamic Scheduling. In *1995 International Logic Programming Symposium.* MIT Press, December 1995.
16. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming,* 22(1):1–45, 1995.
17. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.
18. D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming,* 13(2 and 3):291–314, July 1992.
19. A. King and P. Soper. Depth-k Sharing and Freeness. In *International Conference on Logic Programming.* MIT Press, June 1995.
20. Andy King, Jan-Georg Smaus, and Patricia M. Hill. Quotienting share for dependency analysis. In *European Symposium on Programming,* pages 59–73, 1999.
21. V. Lagoon and P. J. Stuckey. Precise pair-sharing analysis of logic programs. In *ACM SIGPLAN international conference on Principles and practice of declarative programming,* pages 99–108, 2002.
22. Giorgio Levi and Fausto Spoto. Non pair-sharing and freeness analysis through linear refinement. In *Partial Evaluation and Semantic-Based Program Manipulation,* pages 52–61, 2000.
23. A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs. In *Seventh International Conference on Logic Programming,* pages 747–762. MIT Press, June 1990.
24. K. Muthukumar, F. Bueno, M. Garcia de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming,* 38(2):165–218, February 1999.
25. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming,* pages 49–63. MIT Press, June 1991.
26. D. A. Plaisted. The occur-check problem in prolog. In *International Symposium on Logic Programming,* pages 272–281. IEEE, 1984.
27. H. Søndergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 213,* pages 327–338. Springer-Verlag, 1986.

Backward Pair Sharing Analysis

Lunjin Lu[1] and Andy King[2]

[1] Oakland University, MI 48309, USA.
[2] University of Kent, CT2 7NF, UK.

Abstract. This paper presents a backward sharing analysis for logic programs. The analysis computes pre-conditions for a query that guarantee a given post-condition is satisfied after the query is successfully executed. The analysis uses a pair sharing domain and is capable of inferring pre-conditions that ensure the absence of sharing. This, in turn, has many applications in logic programming. The work is unique in that it demonstrates that backward analysis is applicable even for properties that are not closed under instantiation.
Keywords: Abstract interpretation; Backward analysis; Pair-Sharing

1 Introduction

Sharing analysis is useful in specialising, optimising, compiling and parallelising logic programs and thus sharing analysis is an important topic of both abstract interpretation and logic programming. Sharing domains track possible sharing between program variables since optimisations and transformations can typically only be applied in the absence of sharing. Conventionally, sharing is traced in the direction of the control-flow in a query-directed fashion from an initial state. This paper considers the dual problem: the problem of inferring a set of initial states for which an optimisation or transformation is applicable. Specifically, the paper presents a novel backward sharing analysis that propagates information against the control-flow to infer pre-conditions on the variable sharing of a query. The pre-conditions are inferred from a given post-condition which encodes the sharing requirement. The analysis guarantees that if the inferred pre-condition holds for a query, then any successful computation will satisfy the post-condition, thereby ensuring that the optimisation or transformation is applicable.

This paper presents a novel, backward sharing analysis that is realised with abstract interpretation [2]. It is constructed as a suite of abstract operations on the classic pair-sharing domain [1,14,24] which captures information about linearity and variable independence. These operations instantiate a backward analysis framework which, in turn, takes care of the algorithmic concerns associated with a fixpoint calculation. This paper focuses on the two key abstract operations: the backward abstract unification and the backward abstract composition operations. The backward abstract unification operation computes a pre-condition for a given equation and its post-condition. The backward abstract composition operation calculates a pre-condition for a call from its post-condition and a description of its answer substitutions. The other abstract operations are

© Springer-Verlag Berlin Heidelberg 2004

much simpler and are more or less straightforward to construct. These operations are omitted from the paper for brevity.

The remainder of the paper is organised as follows. Section 2 introduces basic concepts used throughout the paper. Section 3 contains a brief description of the abstract interpretation framework within which the backward sharing analysis sits. Sections 4-6 introduce the abstract domain and the abstract operations. Section 7 reviews related work and section 8 concludes. Proofs are omitted due to space limitation.

2 Preliminaries

This section recalls some basic concepts in logic programming and abstract interpretation. The reader is referred to [17] and [2] for more detailed exposition.

Let Σ be a set of function symbols, $\mathcal{V}$ a denumerable set of variables. We assume that Σ contains at least one function symbol of arity 0. *Term* denotes the set of terms that can be constructed from Σ and $\mathcal{V}$.

An equation is a formula of the form $t_1 = t_2$ with $t_1, t_2 \in \mathit{Term}$. The set of all equations is denoted as *Eqn* whereas the set of all substitutions is denoted *Sub*. Let $dom(\theta)$ be the domain of a substitution θ, and $\mathbf{V}(o)$ the set of variables in the syntactic object o. Let $\mathit{Sub}_{fail} = \mathit{Sub} \cup \{fail\}$. Given $e \in \mathit{Eqn}$, $mgu : \mathit{Eqn} \mapsto \mathit{Sub}_{fail}$ returns either a most general unifier for e if e is unifiable or *fail* otherwise. For brevity, let $mgu(t_1, t_2) = mgu(t_1 = t_2)$. The function composition operation $\circ$ is defined as $f \circ g = \lambda x.f(g(x))$. Denote the size of a term t by $|t|$ and the number of elements in a set S by $|S|$.

Let $\langle C, \sqsubseteq^C, \sqcup^C, \sqcap^C, \top^C, \bot^C \rangle$ be a complete lattice and $S \subseteq C$. S is a Moore family iff $\top^C \in S$ and $s_1 \sqcap^C s_2 \in S$ for any $s_1, s_2 \in S$. Let $\langle D, \sqsubseteq^D \rangle$ be a poset. A function $\gamma : D \mapsto C$ is a concretization function iff γ is a monotone and $\gamma(D)$ is a Moore family. A concretization function from D to C induces a Galois connection between D and C [2]. The induced adjoint, called an abstraction function, is $\alpha(c) = \sqcap^D \{d \in D \mid c \sqsubseteq^C \gamma(d)\}$.

3 Framework for Backward Analysis

The backward sharing analysis is based on a novel abstract semantics [18]. The abstract semantics is sketched below so that the paper is self-contained. It is a (lower) approximation to a collecting semantics that maps a call $p(\boldsymbol{x})$ and a set Θ of substitutions into a set Ξ of substitutions such that, for any $\xi \in \Xi$, if δ is a computed answer for $\xi(p(\boldsymbol{x}))$ then $\delta \circ \xi \in \Theta$, i.e., $\{\Xi\}p(\boldsymbol{x})\{\Theta\}$ is a valid partial correctness formula. Note that $\Xi = \emptyset$ is a valid solution. In a more precise solution, Ξ contains more substitutions without compromising correctness. The collecting semantics is defined on the concrete domain $\langle \wp(\mathit{Sub}), \subseteq \rangle$. It is defined in terms of a suite of concrete operations. The two most important operators are $uf^{-1} : \mathit{Eqn} \times \wp(\mathit{Sub}) \mapsto \wp(\mathit{Sub})$ and $\circ^{-1} : \wp(\mathit{Sub}) \times \wp(\mathit{Sub}) \mapsto \wp(\mathit{Sub})$ defined

$$uf^{-1}(e, \Theta) = \{\xi \in \mathit{Sub} \mid mgu(\xi(e)) \circ \xi \in \Theta\}$$
$$\Psi \circ^{-1} \Theta = \{\omega \in \mathit{Sub} \mid \forall \psi \in \Psi.(\psi \circ \omega \in \Theta)\}$$

The concrete operation uf^{-1} reverses unification. For a given equation e and a set Θ of success substitutions, it returns the set of those initial substitutions ξ such that the unification of e under ξ results in a success substitution in Θ. The concrete operation $\circ^{-1}$ reverses the composition of one substitution with another. Given a set Θ of success substitutions and a set Ψ of computed answer substitutions, it calculates the set of those initial substitutions ω such that the composition of any $\psi \in \Psi$ with ω obtains a success substitution in Θ.

The abstract semantics is parameterised by an abstract domain $\langle Z, \sqsubseteq \rangle$ but actually operates on the disjunctive completion of $\langle Z, \sqsubseteq \rangle$. Let $S \subseteq Z$ and define $\downarrow(S) = \{z_1 \in Z \mid \exists z_2 \in S. z_1 \sqsubseteq z_2\}$. The set of order-ideals of Z, denoted $\wp^{\downarrow}(Z)$, is defined by $\wp^{\downarrow}(Z) = \{S \subseteq Z \mid S = \downarrow(S)\}$. Note that each order-ideal can be represented by the collection of its maximal elements. This representation of an order-ideal will be used in the sequel. The abstract semantics operates over $\wp^{\downarrow}(Z)$ to express pre-conditions which are disjunctive [18]. The semantics essentially computes a denotation for each call which maps a single post-condition (in Z) to a disjunction of pre-conditions (in $\wp^{\downarrow}(Z)$). The abstract semantics is defined in terms of a suite of abstract operations - one for each concrete operation. Implementing these operations instantiates the abstract semantics to obtain a backward analysis. The two abstract operations that mimic uf^{-1} and $\circ^{-1}$ are $\overline{uf}^{-1} : Eqn \times Z \mapsto \wp^{\downarrow}(Z)$ and $\overline{\circ}^{-1} : Z \times Z \mapsto \wp^{\downarrow}(Z)$. The backward abstract unification operation $\overline{uf}^{-1}$ computes a pre-condition for a given equation and its post-condition. The backward abstract composition operation $\overline{\circ}^{-1}$ calculates a pre-condition for an atom from its post-condition and a description of its answer substitutions. These abstract operations are obtained by inverting the corresponding abstract operations from a forward sharing analysis. Let $\gamma : Z \mapsto \wp(Sub)$ be a concretization function. Define $\gamma^{\cup}(Y) = \bigcup_{y \in Y} \gamma(y)$. These abstract operations are required to satisfy their local safety requirements.

(a) $\gamma^{\cup}(\overline{uf}^{-1}(e, z)) \subseteq uf^{-1}(e, \gamma(z))$ for any $e \in Eqn$ and any $z \in Z$, and
(b) $\gamma^{\cup}(z \overline{\circ}^{-1} z') \subseteq \gamma(z) \circ^{-1} \gamma(z')$ for any $z, z' \in Z$.

These requirements state that each abstract operation faithfully lower approximates its corresponding concrete operation.

The following three sections present the backward sharing analysis. The sharing domain captures information about linearity and dependencies between variables of interest. The abstract operations are obtained by inverting abstract operations from a forward sharing analysis.

4 Abstract Domain

A term t is linear iff it does not contain multiple occurrences of any variable. Let the predicate $linear(t)$ hold iff t is linear. Two terms s and t share a variable iff $\mathbf{V}(s) \cap \mathbf{V}(t) \neq \emptyset$. Two variables x and y share under a substitution θ if $\theta(x)$ and $\theta(y)$ share. The possible sharing and possible non-linearity of variables under a substitution θ are represented as a symmetric relation $\pi \subseteq VI \times VI$ [24] where VI is the set of variables in the program. Let PS be the set of symmetric relations over *VI*. The abstract domain for sharing and linearity, dubbed pair sharing, is $\langle PS, \subseteq, \emptyset, VI^2, \cap, \cup\rangle$ which is a complete lattice. A Galois connection between $\langle PS, \subseteq\rangle$ and $\langle \wp(Sub), \subseteq\rangle$ is obtained as follows [1].

$$\alpha : \wp(Sub) \mapsto PS$$
$$\gamma : PS \mapsto \wp(Sub)$$
$$\alpha(\Theta) = \bigcup_{\theta\in\Theta} \left\{ \langle x,y\rangle \in VI^2 \;\middle|\; \begin{array}{c} (x \neq y \wedge \mathbf{V}(\theta(x)) \cap \mathbf{V}(\theta(y)) \neq \emptyset) \\ \vee \\ (x = y \wedge \neg linear(\theta(x))) \end{array} \right\}$$
$$\gamma(\pi) = \bigcup\{\Theta \subseteq Sub \mid \alpha(\Theta) \subseteq \pi\}$$

We will write $(u \leftrightarrow v) \in \pi$ to stand for $\{\langle u,v\rangle, \langle v,u\rangle\} \subseteq \pi$. Thus, the set $\{x_1 \leftrightarrow y_1, \cdots, x_n \leftrightarrow y_n\}$ abbreviates $\cup_{i=1}^{n}\{\langle x_i,y_i\rangle, \langle y_i,x_i\rangle\}$. If $(u \leftrightarrow v) \in \pi$ then $(u \leftrightarrow v)$ is called a link in π. We will also use $u \overset{\pi}{\leftrightarrow} v$ to abbreviate $(u \leftrightarrow v) \in \pi$ and $u \overset{\pi}{\Leftrightarrow} v$ to indicate $(u = v \vee u \overset{\pi}{\leftrightarrow} v)$. Define $X \otimes Y = X \times Y \cup Y \times X$ where $X \times Y$ is the Cartesian product of X and Y. $X \otimes Y$ is used to generate a link between each variable of X and each variable of Y. For instance $\{x\} \otimes \{y,z\} = \{x \leftrightarrow y, x \leftrightarrow z\}$. Define $\pi_x = \{y \mid (x \leftrightarrow y) \in \pi\}$. The set π_x includes all the variables that share with x in π. Note that $x \in \pi_x$ if $\langle x,x\rangle \in \pi$. As a further example, $\pi_x = \{y,z\}$ and $\pi_y = \{x,y\}$ where $\pi = \{x \leftrightarrow y, x \leftrightarrow z, y \leftrightarrow y\}$. Define $\phi_X = \bigcup_{x\in X}\phi_x$ where $X \subseteq VI$.

The next stage in the design of the backward sharing analysis is to construct the abstract operations and argue their correctness.

5 Abstract Operation $\overline{uf}^{-1}$

The backward abstract unification operation $\overline{uf}^{-1}$ computes a pre-condition for a given equation and a given post-condition. It is constructed by inverting a forward abstract unification operation given below. The following predicate $\chi : Term \times PS \mapsto \{true, false\}$ will be used in the definition of the forward abstract unification operation: $\chi(t,\pi) = \neg linear(t) \vee ((\mathbf{V}(t))^2 \cap \pi \neq \emptyset)$. The predicate $\chi(t,\pi)$ holds if $\theta(t)$ is non-linear for some $\theta \in \gamma(\pi)$ [1]. We abbreviate $\chi(t,\pi) = true$ as $\chi(t,\pi)$. The forward abstract unification operation is derived from an operation given in [14].

$$\overline{uf}(s = t, \pi) =$$
$$\begin{cases} \pi \setminus (\mathbf{V}(s) \otimes VI) & \text{if } \mathbf{V}(t) = \emptyset \\ \pi \setminus (\mathbf{V}(t) \otimes VI) & \text{if } \mathbf{V}(s) = \emptyset \\ \pi \cup link(s,t,\pi) \cup (\chi(t,\pi) \rhd link(s,s,\pi)) \cup (\chi(s,\pi) \rhd link(t,t,\pi)) & \text{otherwise} \end{cases}$$

where $link(s,t,\pi) = \{u \leftrightarrow v \mid x \in \mathbf{V}(s) \wedge x \overset{\pi}{\Leftrightarrow} u \wedge v \overset{\pi}{\Leftrightarrow} y \wedge y \in \mathbf{V}(t)\}$ and $\rhd$ is defined $B \rhd \pi = (\text{if } B \text{ then } \pi \text{ else } \emptyset)$. The forward abstract unification operation safely upper approximates the forward concrete unification operation *uf* [14] where

$$uf(e,\Theta) = \{mgu(\theta(e)) \circ \theta \mid \theta \in \Theta\}$$

The following lemma justifies the construction of the backward abstract unification operation by inverting the forward abstract unification operation.

Lemma 1. *If* $\overline{uf}(s=t,\pi) \subseteq \psi$ *then* $\gamma(\pi) \subseteq uf^{-1}(s=t,\gamma(\psi))$. ∎

According to lemma 1, a pre-condition for an equation and a post-condition can be obtained as follows. The forward abstract unification operator is run on the equation $s=t$ and each $\pi \in PS$. The pre-condition contains those π such that $\overline{uf}(s=t,\pi) \subseteq \psi$ which are also maximal. Therefore, the following is a correct specification for the backward abstract unification operation.

$$\overline{uf}^{-1}(s=t,\psi) = \{\pi \mid \overline{uf}(s=t,\pi) \subseteq \psi \land \forall \pi' \in PS.(\overline{uf}(s=t,\pi') \subseteq \psi \Rightarrow \pi \not\subset \pi')\}$$

Computing $\overline{uf}^{-1}(s=t,\psi)$ via membership checking is however not feasible. Suppose that VI contains n variables. The abstract domain then has $2^{\frac{n(n+1)}{2}}$ elements. Running $\overline{uf}$ on all these elements is practically impossible even for a relatively small n, say 7. The remainder of this subsection gives a polynomial method for computing $\overline{uf}^{-1}(s=t,\psi)$ starting with simple cases. Without loss of generality, we assume that s and t unify for otherwise, $\{VI^2\}$ is a valid pre-condition.

Case $\mathbf{V}(t)=\emptyset$. The effect of the forward abstract unification of $s=t$ in a pair sharing π is to remove those links that are incident to variables in $\mathbf{V}(s)$. Let ψ be the result of this pruning process – the post-condition. Then the unique maximal pre-condition is given by $\psi \cup (\mathbf{V}(s) \otimes VI)$. So, $\overline{uf}^{-1}(s=t,\psi) = \{\psi \cup (\mathbf{V}(s) \otimes VI)\}$.

Example 1. Let $\psi = \{w \leftrightarrow x, x \leftrightarrow x\}$ and $VI = \{w,x,y,z\}$. Then $\overline{uf}^{-1}(f(x,y)=f(a,b),\psi) = \{\{w \leftrightarrow x, w \leftrightarrow y, x \leftrightarrow x, x \leftrightarrow y, x \leftrightarrow z, y \leftrightarrow y, y \leftrightarrow z\}\}$. ∎

Case $\mathbf{V}(s)=\emptyset$. By symmetry to the above, $\overline{uf}^{-1}(s=t,\psi) = \{\psi \cup (\mathbf{V}(t) \otimes VI)\}$. When $\mathbf{V}(t)=\emptyset$ and $\mathbf{V}(s)=\emptyset$, both cases apply and $\overline{uf}^{-1}(s=t,\psi)=\{\psi\}$.

Case $\mathbf{V}(s) \neq \emptyset \land \mathbf{V}(t) \neq \emptyset$. By the definition of $\overline{uf}$, we have $\pi \subseteq \overline{uf}(s=t,\pi)$ for any π. Thus if $\overline{uf}(s=t,\pi) \subseteq \psi$ then $\pi \subseteq \psi$, hence π can be obtained by removing a symmetric subset τ of ψ. The problem is how to find τ. The forward abstract unification operation $\overline{uf}$ produces a link $u \leftrightarrow v$ from $s=t$ and a subset of the links in ψ. Such a subset of links justifies the presence of $u \leftrightarrow v$ in $\overline{uf}(s=t,\psi)$; and is henceforth called a support set for $u \leftrightarrow v$.

Example 2. Let $\psi = \{w \leftrightarrow x, x \leftrightarrow y, x \leftrightarrow z, y \leftrightarrow z\}$. We have $\chi(x,\psi) = \mathit{false}$ and $\chi(f(y,z),\psi) = \mathit{true}$. So, $\overline{uf}(x=f(y,z),\psi) = \psi \cup \mathit{link}(x,f(y,z),\psi) \cup (\mathit{true} \rhd \mathit{link}(x,x,\psi)) \cup (\mathit{false} \rhd \mathit{link}(f(y,z),f(y,z),\psi)) = \psi \cup \mathit{link}(x,f(y,z),\psi) \cup \mathit{link}(x,x,\psi)$. That $(w \leftrightarrow y) \in \mathit{link}(x,f(y,z),\psi)$ has two justifications: one is that $(w \leftrightarrow x) \in \psi$, $x \in \mathbf{V}(s)$ and $y \in \mathbf{V}(f(y,z))$; the other is that $(w \leftrightarrow x) \in \psi$, $x \in \mathbf{V}(x)$, $z \in \mathbf{V}(f(y,z))$ and $(y \leftrightarrow z) \in \psi$. The link $(w \leftrightarrow y)$ occurs in $\mathit{link}(x,x,\psi)$ because $(w \leftrightarrow x) \in \psi$, $x \in \mathbf{V}(x)$, $x \in \mathbf{V}(x)$ and $(x \leftrightarrow y) \in \psi$. Thus, there are three support sets for $w \leftrightarrow y$: $S_1 = \{w \leftrightarrow x\}$ and $S_2 = \{w \leftrightarrow x, y \leftrightarrow z\}$ and $S_3 = \{w \leftrightarrow x, x \leftrightarrow y\}$. ∎

In order to ensure that forward abstract unification cannot produce a link $u \leftrightarrow v$ that is not in ψ, all the support sets for $u \leftrightarrow v$ must be destroyed. A support set is destroyed if just one of its links is removed. Therefore, to prevent $u \leftrightarrow v$ from being produced, it is necessary to remove a set of links that contains one link from each of its support sets. Such a set is called a frontier for $u \leftrightarrow v$.

Definition 1. *Let $h : PS \mapsto PS$ be monotonic and $\psi \in PS$ and $\tau \subseteq \psi$. If $(u \leftrightarrow v) \in h(\psi)$ and $(u \leftrightarrow v) \notin h(\psi \setminus \tau)$, we say that (removal of) τ (from ψ) excludes $u \leftrightarrow v$ from $h(\psi)$ and that τ is a frontier for $u \leftrightarrow v$ in $h(\psi)$. Let $\phi \in PS$. We say that τ is a frontier for ϕ in $h(\psi)$ if, for each link $u \leftrightarrow v \in \phi$, τ is a frontier for $u \leftrightarrow v$ in $h(\psi)$.* ∎

The particular notions of frontier and exclusion that are required to define backward abstract unification are obtained by putting $h = \lambda\phi.\overline{uf}(s = t, \phi)$. Another instance of these concepts appears in section 6.

Example 3. There are four frontiers for the $w \leftrightarrow y$ link of example 2. They are $F_1 = \{w \leftrightarrow x\}$, $F_2 = \{w \leftrightarrow x, x \leftrightarrow y\}$, $F_3 = \{w \leftrightarrow x, y \leftrightarrow z\}$ and $F_4 = \{w \leftrightarrow x, x \leftrightarrow y, y \leftrightarrow z\}$. Removing any F_i for $1 \leq i \leq 4$ from ψ will prevent $w \leftrightarrow y$ from being produced. ∎

The above example demonstrates that one frontier for a link may be contained in another. Removing one frontier from ψ results in a pair sharing that is a superset of that obtained by removing another frontier from ψ. Since the precondition that is the object of the computation contains maximal pair sharings, only minimal frontiers for the link should be removed. The following example shows that a link may have more than one minimal frontiers.

Example 4. Let $\psi = \{w \leftrightarrow x, y \leftrightarrow z\}$. Then $(w \leftrightarrow z) \in \overline{uf}(x = g(y), \psi)$. The link has one support set: $\{w \leftrightarrow x, y \leftrightarrow z\}$. Two minimal frontiers for $w \leftrightarrow z$ are $\{w \leftrightarrow x\}$ and $\{y \leftrightarrow z\}$ which are incomparable. ∎

Some links have no frontiers at all. For example, let $\psi = \emptyset$. Then $\overline{uf}(x, y, \psi) = \{x \leftrightarrow y\}$. This indicates that the post-condition ψ is unsatisfiable.

Definition 2. *Let $h : PS \mapsto PS$ be monotonic, $\psi \in PS$ and $\Pi \subseteq PS$. Π is a complete set of frontiers for a link $u \leftrightarrow v$ (a set ϕ of links respectively) in $h(\psi)$ if*

(i) every $\pi \in \Pi$ is a frontier for $u \leftrightarrow v$ (ϕ respectively) in $h(\psi)$; and
(ii) every minimal frontier for $u \leftrightarrow v$ (ϕ respectively) in $h(\psi)$ is in Π. ∎

Observe that a complete set of frontiers may contain a non-minimal frontier.

5.1 Minimal Frontier Function

By the definition of $\overline{uf}$, a link $(u \leftrightarrow v) \notin \psi$ occurs in $\overline{uf}(s = t, \psi)$ iff it occurs in $link(s, t, \pi)$, or $\chi(t, \pi) \rhd link(s, s, \pi)$ or $\chi(s, \pi) \rhd link(t, t, \pi)$. Thus, it is excluded from $\overline{uf}(s = t, \psi)$ iff it is excluded from $link(s,t,\pi)$, and $\chi(t,\pi) \rhd link(s,s,\pi)$ and $\chi(s,\pi) \rhd link(t,t,\pi)$.

We first consider how to exclude a link $u \leftrightarrow v$ from $link(s,t,\psi)$. We rewrite the definition of $link(s,t,\psi)$ into $link(s,t,\psi) = \bigcup_{i=1}^{4} \sigma_i(s,t,\psi)$ where

$$\begin{aligned}
\sigma_1(s,t,\psi) &= \{u \leftrightarrow v \mid u \in \mathbf{V}(s) \wedge v \in \mathbf{V}(t)\} \\
\sigma_2(s,t,\psi) &= \{u \leftrightarrow v \mid u \in \mathbf{V}(s) \wedge v \notin \mathbf{V}(t) \wedge (\psi_v \cap \mathbf{V}(t)) \neq \emptyset\} \\
\sigma_3(s,t,\psi) &= \{u \leftrightarrow v \mid u \notin \mathbf{V}(s) \wedge v \in \mathbf{V}(t) \wedge (\psi_u \cap \mathbf{V}(s)) \neq \emptyset\} \\
\sigma_4(s,t,\psi) &= \{u \leftrightarrow v \mid u \notin \mathbf{V}(s) \wedge v \notin \mathbf{V}(t) \wedge (\psi_u \cap \mathbf{V}(s)) \neq \emptyset \wedge (\psi_v \cap \mathbf{V}(t)) \neq \emptyset\}
\end{aligned}$$

Observe that $(u \leftrightarrow v) \in link(s,t,\psi)$ iff $(u \leftrightarrow v) \in \sigma_i(s,t,\psi)$ for some $1 \leq i \leq 4$. Note that $(u \leftrightarrow v) \in \sigma_i(s,t,\psi)$ implies $(u \leftrightarrow v) \notin \sigma_j(s,t,\psi)$ for $j \neq i$. The following computes the set of minimal frontiers for $u \leftrightarrow v$ in $link(s,t,\psi)$.

$$mf_{link}(u,v,s,t,\psi) = \begin{cases} \emptyset & \text{if } u \in \mathbf{V}(s) \wedge v \in \mathbf{V}(t) \\ \{(\{v\} \otimes \mathbf{V}(t)) \cap \psi\} & \text{if } u \in \mathbf{V}(s) \wedge v \notin \mathbf{V}(t) \\ \{(\{u\} \otimes \mathbf{V}(s)) \cap \psi\} & \text{if } u \notin \mathbf{V}(s) \wedge v \in \mathbf{V}(t) \\ \{(\{u\} \otimes \mathbf{V}(s)) \cap \psi, (\{v\} \otimes \mathbf{V}(t)) \cap \psi\} & \text{if } u \notin \mathbf{V}(s) \wedge v \notin \mathbf{V}(t) \end{cases}$$

Each element in $mf_{link}(u,v,s,t,\psi)$ excludes $u \leftrightarrow v$ from $link(s,t,\psi)$. The empty set in the first branch indicates that the presence of $u \leftrightarrow v$ in $\sigma_1(s,t,\psi)$ is independent of ψ and hence cannot be excluded. The second contains one frontier that consists of links between v and variables in $\mathbf{V}(t)$. The third is dual to the second. The fourth returns a set of two minimal frontiers. One consists of links between v and variables in $\mathbf{V}(t)$; and the other consists of links between u and variables in $\mathbf{V}(s)$.

Lemma 2. *$mf_{link}(u,v,s,t,\psi)$ is a complete set of frontiers for $u \leftrightarrow v$ in $link(s,t,\psi)$.* ∎

We now consider how to exclude $u \leftrightarrow v$ from $(\chi(t,\psi) \rhd link(s,s,\psi))$. By the definition of $\rhd$, $\chi(t,\psi) \rhd link(s,s,\psi) =$ (if $\chi(t,\psi)$ then $link(s,s,\psi)$ else $\emptyset$). Thus, we can either make the condition $\chi(t,\psi)$ false or exclude $u \leftrightarrow v$ from $link(s,s,\psi)$. The latter can be accomplished by removing from ψ any element in $mf_{link}(u,v,s,s,\psi)$. Note that $\chi(t,\psi) = \neg linear(t) \vee ((\mathbf{V}(t))^2 \cap \psi \neq \emptyset)$. If $\neg linear(t)$ holds then $\chi(t,\psi)$ cannot be falsified by removing any part of ψ. In this case, we can exclude $u \leftrightarrow v$ from $\chi(t,\psi) \rhd link(s,s,\psi)$ only by removing an element in $mf_{link}(u,v,s,s,\psi)$. Otherwise, $linear(t)$ holds. We can alternatively choose to falsify $((\mathbf{V}(t))^2 \cap \psi \neq \emptyset)$. This can be done by removing all the links in $(\mathbf{V}(t))^2 \cap \psi$. Thus, each element in $(linear(t) \wedge ((\mathbf{V}(t))^2 \cap \psi \neq \emptyset) \rhd \{(\mathbf{V}(t))^2 \cap \psi\}) \cup mf_{link}(u,v,s,s,\psi)$ excludes $u \leftrightarrow v$ from $\chi(t,\psi) \rhd link(s,s,\psi)$. Excluding $u \leftrightarrow v$ from $(\chi(s,\psi) \rhd link(t,t,\psi))$ is symmetric.

Lemma 3. *$(linear(t) \wedge ((\mathbf{V}(t))^2 \cap \psi \neq \emptyset) \rhd \{(\mathbf{V}(t))^2 \cap \psi\}) \cup mf_{link}(u,v,s,s,\psi)$ is a complete set of frontiers for $u \leftrightarrow v$ in $\chi(t,\psi) \rhd link(s,s,\psi)$.* ∎

In order to exclude a link $u \leftrightarrow v$ from $h_1(\psi) \cup h_2(\psi)$, it is necessary to exclude $u \leftrightarrow v$ from both $h_1(\psi)$ and $h_2(\psi)$. This can be accomplished by removing from ψ a frontier for $u \leftrightarrow v$ in $h_1(\psi)$ and a frontier for $u \leftrightarrow v$ in $h_2(\psi)$. The union of a frontier for $u \leftrightarrow v$ in $h_1(\psi)$ and a frontier for $u \leftrightarrow v$ in $h_2(\psi)$ is a frontier for $u \leftrightarrow v$ in $h_1(\psi) \cup h_2(\psi)$. To this end, define $\Psi \uplus \Phi = min(\{\psi \cup \phi \mid \psi \in \Psi \wedge \phi \in \Phi\})$ where $min(\Pi)$ returns the set of the elements in Π that are minimal with respect to $\subseteq$. The operation $\uplus$ is commutative and associative.

Lemma 4. *Let $\psi \in PS$, $h_1, h_2 : PS \mapsto PS$ monotonic functions, and Φ_i a complete set of frontiers for a link $u \leftrightarrow v$ in $h_i(\psi)$ for $1 \leq i \leq 2$. Then $\Phi_1 \uplus \Phi_2$ is a complete set of frontiers for $u \leftrightarrow v$ in $h_1(\psi) \cup h_2(\psi)$.* ∎

The function $mf(u, v, s, t, \psi)$ below returns the set of all minimal frontiers for $u \leftrightarrow v$ in $\overline{uf}(s = t, \psi)$.

$$
\begin{aligned}
&mf(u, v, s, t, \psi) = \\
&\quad mf_{link}(u, v, s, t, \psi) \\
&\quad \uplus ((linear(t) \wedge ((\mathbf{V}(t))^2 \cap \psi \neq \emptyset) \rhd \{(\mathbf{V}(t))^2 \cap \psi\}) \cup mf_{link}(u, v, s, s, \psi)) \\
&\quad \uplus ((linear(s) \wedge ((\mathbf{V}(s))^2 \cap \psi \neq \emptyset) \rhd \{(\mathbf{V}(t))^2 \cap \psi\}) \cup mf_{link}(u, v, t, t, \psi))
\end{aligned}
$$

Note that non-minimal frontiers for a link are removed by the *min* operation employed in the $\uplus$ operation and that minimal frontiers for a link are computed without computing support sets for the link.

Lemma 5. *$mf(u, v, s, t, \psi)$ is a complete set of frontiers for $u \leftrightarrow v$ in $\overline{uf}(s = t, \psi)$.* ∎

Example 5. Let $\psi = \{w \leftrightarrow x, y \leftrightarrow z\}$. Then $(w \leftrightarrow z) \in \overline{uf}(x = g(y), \psi)$. The set $mf(w, z, x, g(y), \psi)$ of minimal frontiers for $w \leftrightarrow z$ in $\overline{uf}(x = g(y), \psi)$ is computed as follows. We calculate $linear(g(y)) = true$ and $\mathbf{V}(g(y))^2 \cap \psi = \{y \leftrightarrow y\} \cap \psi = \emptyset$. Thus, $(linear(g(y)) \wedge (\mathbf{V}(g(y))^2 \cap \psi \neq \emptyset)) = false$ and hence $(linear(g(y)) \wedge (\mathbf{V}(g(y))^2 \cap \psi \neq \emptyset)) \rhd \{\mathbf{V}(g(y))^2 \cap \psi\} = \emptyset$. We can also obtain $(linear(x) \wedge (\mathbf{V}(x)^2 \cap \psi \neq \emptyset)) \rhd \{\mathbf{V}(x)^2 \cap \psi\} = \emptyset$. Thus, $mf(w, z, x, g(y), \psi) = mf_{link}(w, z, x, g(y), \psi) \uplus (\emptyset \cup mf_{link}(w, z, x, x, \psi)) \uplus (\emptyset \cup mf_{link}(w, z, g(y), g(y), \psi))$. We first calculate $mf_{link}(w, z, x, g(y), \psi) = \{(\{w\} \otimes \{x\}) \cap \psi, (\{z\} \otimes \{y\}) \cap \psi\} = \{\{w \leftrightarrow x\}, \{y \leftrightarrow z\}\}$ since $w \notin \{x\}$ and $z \notin \{y\}$. We can also obtain $mf_{link}(w, z, x, x, \psi) = \{\{w \leftrightarrow x\}\}$ and $mf_{link}(w, z, g(y), g(y), \psi) = \{\{y \leftrightarrow z\}\}$. So, $mf(w, z, x, g(y), \psi) = \{\{w \leftrightarrow x\}, \{y \leftrightarrow z\}\} \uplus \{\{w \leftrightarrow x\}\} \uplus \{\{y \leftrightarrow z\}\} = \{\{w \leftrightarrow x\}, \{y \leftrightarrow z\}\}$. ∎

Suppose that two links $u \leftrightarrow v$ and $u' \leftrightarrow v'$ need be excluded from $\overline{uf}(s = t, \psi)$. Removing from ψ a frontier for one link will exclude the link. Both links will be excluded if the union of a frontier for $u \leftrightarrow v$ and a frontier for $u' \leftrightarrow v'$ is removed from ψ. A frontier for a set of n links is the union of n frontiers - one for each link.

Lemma 6. *Let $\psi \in PS$, $h : PS \mapsto PS$ a monotonic function, $\phi_i \in PS$ and $\Pi_i \subseteq PS$ for $1 \leq i \leq 2$. If Π_i is a complete set of frontiers for ϕ_i in $h(\psi)$ then $\Pi_1 \uplus \Pi_2$ is a complete set of frontiers for $\phi_1 \cup \phi_2$ in $h(\psi)$.* ∎

The following lemma provides a constructive method for computing $\overline{uf}^{-1}(s = t, \psi)$ for the case $\mathbf{V}(s) \neq \emptyset \wedge \mathbf{V}(t) \neq \emptyset$. The forward abstract unification operation $\overline{uf}$ is first employed to compute $\psi' = \overline{uf}(s = t, \psi)$. The set of minimal frontiers for $\psi' \setminus \psi$ is then computed. It is $\uplus_{(u \leftrightarrow v) \in (\psi' \setminus \psi)} mf(u, v, s, t, \psi)$. Each pair sharing in $\overline{uf}^{-1}(s = t, \psi)$ is obtained by removing one of these minimal frontiers from ψ.

Lemma 7. $\overline{uf}^{-1}(s = t, \psi) = \{\psi \setminus \tau \mid \tau \in \uplus_{(u \leftrightarrow v) \in (\psi' \setminus \psi)} mf(u, v, s, t, \psi)\}$ *where* $\psi' = \overline{uf}(s = t, \psi)$. ∎

Lemmas 1 and 7 imply the correctness of $\overline{uf}^{-1}$. We now show that $\overline{uf}^{-1}(s = t, \psi)$ is polynomial in $|s| + |t| + |VI|$. Operations $\cup$, $\cap$ and $\otimes$ are polynomial in $|VI|$. Let Π be a set of minimal frontiers and $\pi_1, \pi_2 \in \Pi$ such that $\pi_2 \neq \pi_1$. Then π_1 contains at least one link that does not belong to π_2 Thus, $|\Pi| \leq |VI|^2$ because $|\pi_1| \leq |VI|^2$. So, $\uplus$ is polynomial in $|VI|$. The forward abstract unification $\psi' = \overline{uf}(s = t, \psi)$ is polynomial in $|s| + |t| + |VI|$ [1]. All links $u \leftrightarrow v$ in $\psi' \setminus \psi$ invoke $mf(u, v, s, t, \psi)$ with the same s, t and ψ. Thus, $\mathbf{V}(s)$, $\mathbf{V}(t)$, $linear(s)$ and $linear(t)$ can be computed with their results being memoised for use in computing $mf(u, v, s, t, \psi)$ for different links $u \leftrightarrow v$. This takes $O(|s| + |t|)$ time. Using the memoised results, $mf_{link}(u, v, s, t, \psi)$ is polynomial in $|VI|$; so is $mf(u, v, s, t, \psi)$. The computation $\uplus_{(u \leftrightarrow v) \in (\psi' \setminus \psi)} mf(u, v, s, t, \psi)$ is polynomial in $|VI|$ since it invokes mf and $\uplus$ for $|\psi' \setminus \psi| \leq |VI|^2$ times and both mf and $\uplus$ are polynomial in $|VI|$. So, $\overline{uf}^{-1}(s = t, \psi)$ is polynomial in $|s| + |t| + |VI|$.

6 Abstract Operation $\bar{\circ}^{-1}$

The operator $\bar{\circ} : PS \times PS \mapsto PS$ was originally proposed in [1] for composing an abstract initial substitution for an atom with an abstract answer substitution (for the atom and the abstract initial substitution) to obtain an abstract success substitution. It will inverted to obtain $\bar{\circ}^{-1}$ and it is defined

$$\pi \bar{\circ} \phi = \{\langle u, v \rangle \mid u \overset{\phi}{\leftrightarrow} v \vee \exists x, y.(u \overset{\phi}{\Leftrightarrow} x \wedge x \overset{\pi}{\leftrightarrow} y \wedge y \overset{\phi}{\Leftrightarrow} v)\}$$

Note that $\bar{\circ}$ is not commutative. The following result is Lemma 4.4 in [1].

Proposition 1 (Codish, Dams and Yardeni). *Let $\sigma \in \gamma(\pi)$ and $\theta \in \gamma(\phi)$. Then $\sigma \circ \theta \in \gamma(\pi \bar{\circ} \phi)$.*

The following lemma justifies the construction tactic of inverting $\bar{\circ}$ to obtain $\bar{\circ}^{-1} : PS \times PS \mapsto PS$.

Lemma 8. *If $\pi \bar{\circ} \phi \subseteq \psi$ then $\gamma(\phi) \subseteq (\gamma(\pi) \circ^{-1} \gamma(\psi))$.* ∎

The above lemma implies that the following is a correct specification for $\bar{\circ}^{-1}$.

$$\pi\bar{\circ}^{-1}\psi = \{\phi \mid (\pi\bar{\circ}\phi \subseteq \psi) \wedge \forall\phi' \in PS.((\pi\bar{\circ}\phi' \subseteq \psi) \Rightarrow (\phi' \not\supset \phi))$$

Again, we need to find a practical method for computing $\pi\bar{\circ}^{-1}\psi$. For any $\pi, \phi \in PS$, we have $\phi \subseteq \pi\bar{\circ}\phi$. Thus, if $\pi\bar{\circ}\phi \subseteq \psi$ then $\phi \subseteq \psi$. A pair sharing in $\pi\bar{\circ}^{-1}\psi$ can be obtained by removing a set τ of links from ψ. The problem is thus again equivalent to finding τ. The notion of a support set and that of a frontier carry over with $\lambda\phi.(\pi\bar{\circ}\phi)$ taking the place of $\lambda\phi.\overline{uf}(s = t, \phi)$.

Example 6. Let $\psi = \{w \leftrightarrow x, x \leftrightarrow y, y \leftrightarrow z\}$ and $\pi = \{x \leftrightarrow y\}$. Then $\pi\bar{\circ}\psi = \pi \cup \psi \cup \{w \leftrightarrow y, w \leftrightarrow z, x \leftrightarrow x, x \leftrightarrow z, y \leftrightarrow y\}$. There is one support set for $w \leftrightarrow z$: $\{w \leftrightarrow x, y \leftrightarrow z\}$. Two minimal frontiers are obtained from the support set: $\{w \leftrightarrow x\}$ and $\{y \leftrightarrow z\}$. Removing either of them excludes $w \leftrightarrow z$ from $\pi\bar{\circ}\psi$. ∎

By expanding the definition of $\bar{\circ}$, we have $\pi\bar{\circ}\psi = \pi \cup \psi \cup (\psi \bowtie \pi) \cup (\pi \bowtie \psi) \cup (\psi \bowtie \pi \bowtie \psi)$ where $\pi_1 \bowtie \pi_2 = \{u \leftrightarrow v \mid \exists w.(u \overset{\pi_1}{\leftrightarrow} w \wedge w \overset{\pi_2}{\leftrightarrow} v)\}$ is associative. The following function computes the set of minimal frontiers for $u \leftrightarrow v$ in $\pi\bar{\circ}\psi$.

$$\begin{aligned} mftc(u, v, \pi, \psi) = {} & \{(\{u\} \otimes \pi_v) \cap \psi\} \uplus \{(\pi_u \otimes \{v\}) \cap \psi\} \\ & \uplus \{(\{u\} \otimes \pi_{\psi_v}) \cap \psi, (\{v\} \otimes \pi_{\psi_u}) \cap \psi\} \end{aligned}$$

Some explanation is in order. Assume that $(u \leftrightarrow v) \notin \psi$ and $(u \leftrightarrow v) \notin \pi$. Consider how to exclude $u \leftrightarrow v$ from $\pi\bar{\circ}\psi$. The link $u \leftrightarrow v$ belongs to $\psi \bowtie \pi$ if u is linked via ψ to any variable that is linked to v via π. Thus, it is necessary to remove all links in $(\{u\} \otimes \pi_v) \cap \psi$. Excluding $u \leftrightarrow v$ from $\pi \bowtie \psi$ is symmetric. Observe that π_{ψ_v} consists of those variables that are linked to v via a link in π followed by a link in ψ. In order to exclude $u \leftrightarrow v$ from $\psi \bowtie \pi \bowtie \psi$, we either remove the set of all links from u to variables in π_{ψ_v} or remove the set of all links from v to variables in π_{ψ_u}. The former is $(\{u\} \otimes \pi_{\psi_v}) \cap \psi$ and the latter is $(\{v\} \otimes \pi_{\psi_u}) \cap \psi$. Finally, the link $u \leftrightarrow v$ is excluded from $\pi\bar{\circ}\psi$ if it is excluded from $\psi \bowtie \pi$, $\pi \bowtie \psi$ and $\psi \bowtie \pi \bowtie \psi$.

Lemma 9. *For any $\pi \in PS$, $mftc(u, v, \pi, \psi)$ is a complete set of frontiers for $u \leftrightarrow v$ in $(\psi \bowtie \pi) \cup (\pi \bowtie \psi) \cup (\psi \bowtie \pi \bowtie \psi)$.* ∎

The following lemma provides a polynomial method for computing $\pi\bar{\circ}^{-1}\psi$. Together with lemma 8, it ensures the correctness of the abstract operation $\bar{\circ}^{-1}$. If $\pi \not\subseteq \psi$ then $\pi\bar{\circ}\phi \not\subseteq \psi$ for any $\phi \in PS$. In this case, the post-condition ψ is unsatisfiable and hence the pre-condition is the empty set of pair sharings. Otherwise, the pre-condition consists of those pair sharings that are obtained by removing minimal frontiers for $(\pi\bar{\circ}\psi) \setminus \psi$.

Lemma 10. *For any $\pi, \psi \in PS$,*

$$\pi\bar{\circ}^{-1}\psi = \begin{cases} \emptyset & \textit{if } \pi \not\subseteq \psi \\ \{\psi \setminus \tau \mid \tau \in \uplus_{(u \leftrightarrow v) \in (\pi\bar{\circ}\psi)\setminus\psi} mftc(u, v, \pi, \psi)\} & \textit{otherwise} \end{cases}$$ ∎

Example 7. Continuing with example 6, $(\pi \bar{\circ} \psi) \setminus \psi = \{w \leftrightarrow y, w \leftrightarrow z, x \leftrightarrow x, x \leftrightarrow z, y \leftrightarrow y\}$. We have $\pi_w = \pi_z = \emptyset$, $\pi_{\psi_z} = \pi_y = \{x\}$ and $\pi_{\psi_w} = \pi_x = \{y\}$. Thus, $\mathit{mftc}(w, z, \pi, \psi) = \{\emptyset\} \uplus \{\emptyset\} \uplus \{(\{w\} \otimes \pi_{\psi_z}) \cap \psi, (\{z\} \otimes \pi_{\psi_w}) \cap \psi\} = \{\{w \leftrightarrow x\}, \{y \leftrightarrow z\}\}$. Omitting details, we obtain other sets of minimal frontiers: $\mathit{mftc}(w, y, \pi, \psi) = \{\{w \leftrightarrow x\}\}$, $\mathit{mftc}(x, z, \pi, \psi) = \{\{y \leftrightarrow z\}\}$ and $\mathit{mftc}(x, x, \pi, \psi) = \mathit{mftc}(y, y, \pi, \psi) = \{\{x \leftrightarrow y\}\}$. The set of minimal frontiers for $(\pi \bar{\circ} \psi) \setminus \psi$ is $\uplus_{(u \leftrightarrow v) \in (\pi \bar{\circ} \psi) \setminus \psi} \mathit{mftc}(u, v, \pi, \psi) = \mathit{mftc}(w, y, \pi, \psi) \uplus \mathit{mftc}(w, z, \pi, \psi) \uplus \mathit{mftc}(x, x, \pi, \psi) \uplus \mathit{mftc}(x, z, \pi, \psi) \uplus \mathit{mftc}(y, y, \pi, \psi) = \{\{w \leftrightarrow x, x \leftrightarrow y, y \leftrightarrow z\}\} = \{\psi\}$. Thus, $\pi \bar{\circ}^{-1} \psi = \{\emptyset\}$. ∎

We now turn to the time complexity of $\pi \bar{\circ}^{-1} \psi$. Observe that $\pi \bar{\circ} \psi$ is polynomial in $|\mathit{VI}|$ since $\bowtie$ and $\cup$ are polynomial in $|\mathit{VI}|$. Since $\uplus$, $\otimes$ and $\cap$ are polynomial in $|\mathit{VI}|$, $\mathit{mftc}(u, v, \pi, \psi)$ is polynomial in $|\mathit{VI}|$. Thus, $\uplus_{(u \leftrightarrow v) \in (\pi \bar{\circ} \psi) \setminus \psi} \mathit{mftc}(u, v, \pi, \psi)$ is polynomial in $|\mathit{VI}|$; so is $\pi \bar{\circ}^{-1} \psi$. Both $\overline{\mathit{uf}}^{-1}(s = t, \psi)$ and $\pi \bar{\circ}^{-1} \psi$ are polynomial; in contrast the widely-used set-sharing analysis has a forward abstract unification operator that is exponential [13].

7 Related Work

Though backward analysis has been a subject of intense research in functional programming [25,11,5], backward analysis has until very recently [9,15,19,22] been rarely applied in logic programming. One notable exception is the demand analysis of [4]. This analysis infers the degree of instantiation necessary to allow the guards of a concurrent constraint program to reduce: it is local analysis that does not consider the possible suspension of body calls. The information it infers is useful in detecting (uni-modal) predicates which can be implemented with relatively straightforward suspension machinery. A more elaborate backward analysis for concurrent constraint programs is [6]. This demand analysis infers how much input is necessary for a procedure to generate a certain amount of output. This information is useful for adding synchronisation constraints to a procedure to delay execution and thereby increase grain size, and yet not introduce deadlock.

Mazur, Janssens and Bruynooghe [21] present a kind of *ad hoc* backward analysis to derive reuse conditions from a goal-independent reuse analysis. The analysis propagates reuse information from a point where a structure is decomposed in a clause to the point where the clause is invoked in its parent clause. This is similar in spirit to how demand is passed from a callee to a caller in the backward analysis described in this paper. However, the reuse analysis does not propagate information right-to-left across a clause, resulting in a less precise analysis. The above backward analyses are not specialisations of any framework for logic program analysis. In [22], a backward analysis is proposed to infer specialisation conditions that decide whether a call to a predicate from a particular call site should invoke a specialised version of the predicate or an unoptimised version. Specifically, if these conditions are satisfied by an (abstract) call in a goal-dependent analysis then the call will possibly lead to valuable optimisations, and therefore it should not be merged with calls that lead to a lower level of optimisation. The specialisation conditions produced by the backward analysis are not sufficient conditions and need to be checked by an ensuing forward analysis. In contrast, the pre-conditions obtained by the backward sharing analysis are guaranteed to be sufficient and thus need not be checked by a forward analysis.

In [15], the authors of the current paper present an abstract semantics for backward analysis of logic programs and specialise it to infer safe modes for queries which ensure that the groundness assertions are not violated using the groundness domain *Pos* [20]. The backward groundness analysis is also used in termination inference by [9]. A backward analysis using the abstract semantics is performed by first computing an upper approximation to the success set of the program and then a lower approximation to the set of programs states (substitutions) that will not violate any assertion. The key operation that propagates information backwards the control-flow is the (intuitionistic) logical implication operation. Thus, the abstract domain of the analysis is required to condense. This, however, is a strong requirement for any domain. The abstract domain for the backward sharing analysis does not condense. The approach advocated in this paper is to found backward analysis on a novel abstract semantics for backward analysis which relaxes this requirement. An analysis using the new abstract semantics is a greatest fixpoint computation whilst an analysis with the abstract semantics in [15] additionally computes a least fixpoint computation.

The authors of this paper have also shown how backward analysis can be used to perform type inference [19]. Given type signatures for a collection of selected predicates such as builtin or library predicates, the analysis of [19] infers type signatures for other predicates such that the execution of any query satisfying the inferred type signatures will not violate the given type signatures. Thus, the backward type analysis generalises type checking in which the programmer manually specifies type signatures for all predicates that are checked for consistency by a type checker. The work of [19] is distinct from that reported in this paper. The property considered in [19] is closed under instantiation whilst that in this paper is not.

Very recently, Gallagher [8] has proposed a program transformation as a tactic for realising backward analysis in terms of forward analysis. Assertions are realised with a meta-predicate d(G,P). The meta-predicate d(G,P) expresses the relationship between an initial goal G and a property P to be checked at some program point. The meta-predicate d(G,P) holds if there is a derivation starting from G leading to the program point with which P is associated. The transformed program defining the predicate ***d*** can be seen as a realisation of the resultants semantics [7]. Backward analysis is performed by examining the meaning of ***d***, which can be approximated using a standard forward analysis, to deduce goals G that imply that the property P holds. This work is both promising and intriguing because it finesses the requirement of calculating a greatest fixpoint. One interesting line of enquiry would be to compare the expressive power of transformation – the pre-conditions its infers – against those deduced via a be-spoke backward analysis framework [15,19].

Giacobazzi [10] proposes a method for an abductive analysis of modular logic programs. From a specification of the success patterns of a predicate defined in one module which calls open predicates defined in other modules, the method derives a specification for the success patterns of the open predicates. In contrast, our method derives a specification for the call patterns of some (unspecified) predicates from a specification of the call patterns of other (specified) predicates.

Pedreschi and Ruggieri [23] develop a calculus of weakest pre-conditions and weakest liberal pre-conditions, the latter of which is essentially a reformulation of Hoare's logic. Weakest liberal pre-conditions are characterised as the greatest fixpoint of a co-continuous function on the space of interpretations. Our work takes these ideas forward to show how abstract interpretation can infer weakest liberal pre-conditions.

Cousot and Cousot [3] explain how a backward collecting semantics can be used to precisely characterise states that arise in finite SLD-derivations. They present both a forward collecting semantics that records the descendant states that arise from a set of initial states and a backward collecting semantics that records those states which occur as ascendant states of the final states. By combining both semantics, they characterise the set of descendant states of the initial states which are also ascendant states of the final states of the transition system. This use of backward analysis is primarily as a device to improve the precision of a goal-dependent analysis. Our work is more radical in the sense that it shows how a bottom-up analysis performed in a backward fashion, can be used to characterise initial queries. Moreover it is used for lower approximation rather than upper approximation.

Hughes and Launchbury [12] shows how and when to reverse an analysis based on abstract interpretation. Their work is concerned with analyses of functional programs. In fact, Hughes and Launchbury argue that ideally the direction of an analysis should be reversed without reference to the concrete semantics. Our work demonstrates that this can be accomplished in the analysis of logic programs.

A systematic comparison of the relative precision of forward and backward abstract interpretation of logic programs is given in [16].

8 Conclusions

A backward sharing analysis for logic programs has been presented. From a given post-condition for an atom, it derives a pre-condition. Any successful execution of the atom in any state satisfying the pre-condition ends in a state satisfying the post-condition. Abstract operations for the backward sharing analysis are constructed by inverting those for a forward sharing analysis. The work demonstrates that backward analysis is applicable for properties that are not closed under instantiation.

Acknowledgements

This work was supported by the National Science Foundation under grants CCR-0131862 and INT-0327760.

References

1. M. Codish, D. Dams, and E. Yardeni. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming,* pages 79–93. The MIT Press, 1991.
2. P. Cousot and R. Cousot. Abstract interpretation: a unified framework for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages,* pages 238–252. The ACM Press, 1977.
3. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming,* 13(1, 2, 3 and 4):103–179, 1992.
4. S.K. Debray. QD-Janus: A sequential implementation of Janus in Prolog. *Software Practice and Experience,* 23(12):1337–1360, 1993.
5. P. Dyber. Inverse image analysis generalises strictness analysis. *Information and Computation,* 90(2):194–216, 1991.
6. M. Falaschi, P. Hicks, and W. Winsborough. Demand Transformation Analysis for Concurrent Constraint Programs. *The Journal of Logic Programming,* 41(3):185–215, 2000.
7. M. Gabbrielli, G. Levi, and M. C. Meo. Resultants Semantics for Prolog. *Journal of Logic and Computation,* 6(4):491–521, 1996.
8. J.P. Gallagher. A Program Transformation for Backwards Analysis of Logic Programs. In *Pre-Proceedings of LOPSTR 2003 - International Symposium on Logic-based Program Synthesis and Transformation,* 2003.
9. S. Genaim and M. Codish. Inferring termination conditions for logic programs using backwards analysis. In R. Nieuwenhuis and A. Voronkov, editors, *Proceedings of the Eighth International Conference on Logic for Programming, Artificial Intelligence and Reasoning,* volume 2250 of *Lecture Notes in Artificial Intelligence,* pages 681–690. Springer, 2001.
10. R. Giacobazzi. Abductive analysis of modular logic programs. *Journal of Logic and Computation,* 8(4):457–484, 1998.
11. C. Hall and D. Wise. Generating function versions with rational strictness patterns. *Science of Computer Programming,* 12(1):39–74, 1989.
12. J. Hughes and J. Launchbury. Reversing abstract interpretations. *Science of Computer Programming.,* 22:307–326, 1994.
13. D. Jacobs and A. Langen. Static analysis of logic programs for independent and parallelism. *The Journal of Logic Programming,* 13(1–4):291–314, 1992.
14. A. King. Pair-sharing over rational trees. *The Journal of Logic Programming,* 46(1–2):139–155, 2000.
15. A. King and L. Lu. A backward analysis for constraint logic programs. *Theory and Practice of Logic Programming,* 2(4&5):517–547, 2002.
16. A. King and L. Lu. Forward versus Backward Verification of Logic Programs. In C. Palamidessi, editor, *Proceedings of Nineteenth International Conference on Logic Programming,* volume 2916 of *Lecture Notes in Computer Science,* pages 315–330, 2003.

17. J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.
18. L. Lu and A. King. A Framework for Backward Analysis of Logic Programs. In Preparation.
19. L. Lu and A. King. Type inference generalises type checking. In M. Hermenegildo and G. Puebla, editors, *Proceedings of Ninth International Static Analysis Symposium,* volume 2477 of *Lecture Notes in Computer Science,* pages 85–101, 2002.
20. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems,* 2(1–4):181–196, 1993.
21. N. Mazur, G. Janssens, and M. Bruynooghe. A module based analysis for memory reuse in Mercury. In J.W. Lloyd et. al, editor, *Proceedings of the First International Conference on Computational Logic,* volume 1861 of *Lecture Notes in Computer Science,* pages 1255–1269. Springer, 2000.
22. N. Mazur, G. Janssens, and W. Vanhoof. Collecting potential optimizations. In M. Leuschel and F. Bueno, editors, *LOPSTR 2002, Logic-based Program Synthesis and Transformation, Revised Selected Papers,* volume 2664 of *Lecture Notes in Computer Science,* pages 109–110. Springer, 2002.
23. D. Pedreschi and S. Ruggieri. Weakest preconditions for pure Prolog programs. *Information Processing Letters,* 67(3):145–150, 1998.
24. H. Søndergaard. An application of abstract interpretation of logic programs: occur check problem. In B. Robinet and R. Wilhelm, editors, *ESOP 86, European Symposium on Programming,* volume 213 of *Lecture Notes in Computer Science,* pages 324–338. Springer, 1986.
25. P. Wadler and R.J.M. Hughes. Projections for strictness analysis. In *Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture,* volume 274 of *Lecture Notes in Computer Science,* pages 385–407, 1987.

Implementing Natural Rewriting and Narrowing Efficiently*

Santiago Escobar

DSIC, Universidad Politécnica de Valencia
Camino de Vera, s/n, E-46022 Valencia, Spain
sescobar@dsic.upv.es

Abstract. *Outermost-needed rewriting/narrowing* is a sound and complete optimal demand-driven strategy for the class of inductively sequential constructor systems. Its parallel extension, known as *weakly,* deals with non-inductively sequential constructor systems. Recently, refinements of (weakly) outermost-needed rewriting and narrowing have been obtained. These new strategies are called *natural rewriting* and *natural narrowing,* respectively, and incorporate a better treatment of demandedness. In this paper, we address the problem of how to implement natural rewriting and narrowing efficiently by using a refinement of the notion of definitional tree, which we call matching definitional tree. We also show how to compile natural rewriting and narrowing to Prolog and provide some promising experimental results.

1 Introduction

A challenging problem in modern programming languages is the discovery of sound and complete evaluation strategies which are 'optimal' w.r.t. some efficiency criterion (typically the number of evaluation steps and the avoidance of infinite, failing or redundant derivations) and which are easily implementable.

A sound and complete rewrite strategy for the class of inductively sequential constructor systems (CSs) is *outermost-needed rewriting* [3]. The extension to narrowing is called *outermost-needed narrowing* (or *needed narrowing*) [5]. Intuitively, a left-linear CS is inductively sequential if there exists some branching selection structure that is inherent to the rules. The optimality properties associated to inductively sequential CSs explain why outermost-needed narrowing has become useful in functional logic programming as the functional logic counterpart of Huet and Lévy's strongly needed reduction [13]. *Weakly outermost-needed rewriting* [4] is defined for non-inductively sequential CSs. Its extension to narrowing is called *weakly outermost-needed narrowing* [4] and is considered as the functional logic counterpart of Sekar and Ramakrishnan's parallel needed reduction [15].

Whereas outermost-needed rewriting and narrowing are optimal w.r.t. to inductively sequential CSs, weakly outermost-needed rewriting and narrowing

* Work partially supported by MCyT under grants TIC2001-2705-C03-01, HA2001-0059 and HU2001-0019.

© Springer-Verlag Berlin Heidelberg 2004

do not work appropriately on non-inductively sequential CSs, as shown by the following example.

Example 1. Consider Berry's program [15] where T and F are constructor symbols and X is a variable:

$$\texttt{B(T,F,X) = T} \qquad \texttt{B(F,X,T) = T} \qquad \texttt{B(X,T,F) = T}$$

This CS is not inductively sequential since there is no branching selection structure in the rules. However, although the CS is not inductively sequential, some terms can still be reduced sequentially, where 'to be reduced sequentially' is understood as the property of reducing only positions that are unavoidable (or "needed") when attempting to obtain a normal form (see [13]). For instance, the term B(B(T,F,T),B(F,T,T),F) has a unique 'optimal' rewrite sequence which achieves its associated normal form T

$$\texttt{B(B(T,F,T),}\underline{\texttt{B(F,T,T)}}\texttt{,F)} \to \underline{\texttt{B(B(T,F,T),T,F)}} \to \texttt{T}$$

However, weakly outermost-needed rewriting is not optimal since, besides the previous optimal sequence, the sequence

$$\texttt{B(}\underline{\texttt{B(T,F,T)}}\texttt{,B(F,T,T),F)} \to \texttt{B(T,}\underline{\texttt{B(F,T,T)}}\texttt{,F)} \to \underline{\texttt{B(T,T,F)}} \to \texttt{T}$$

is also obtained. The reason is that weakly outermost-needed rewriting partitions the CS into the inductively sequential subsets $\mathcal{R}_1 = \{\texttt{B(X,T,F) = T}\}$ and $\mathcal{R}_2 = \{\texttt{B(T,F,X) = T,B(F,X,T) = T}\}$ in such a way that the first step of the former (optimal) rewriting sequence is obtained w.r.t. subset $\mathcal{R}_1$ whereas the first (useless) step of the latter rewriting sequence is obtained w.r.t. subset $\mathcal{R}_2$. Note that the problem also occurs in weakly outermost-needed narrowing. For instance, term B(X, B(F, T, T), F) has the optimal narrowing sequence

$$\texttt{B(X,}\underline{\texttt{B(F,T,T)}}\texttt{,F)} \leadsto_{id} \underline{\texttt{B(X,T,F)}} \leadsto_{id} \texttt{T}$$

whereas weakly outermost-needed narrowing also produces the following non-optimal (due to the unnecessary substitution $\{\texttt{X} \mapsto \texttt{T}\}$) narrowing sequence

$$\texttt{B(X,}\underline{\texttt{B(F,T,T)}}\texttt{,F)} \leadsto_{\{\texttt{X}\mapsto\texttt{T}\}} \underline{\texttt{B(T,T,F)}} \leadsto_{id} \texttt{T}$$

On the other hand, outermost-needed rewriting and narrowing are optimal for inductively sequential CSs only when non-failing input terms are considered, i.e. terms which can be reduced or narrowed to a constructor head-normal form. Hence, some refinement is still possible for failing input terms.

Modern (multiparadigm) programming languages apply computational strategies which are based on some notion of demandness of a position in a term by a rule (see [7]). Programs in these languages are commonly modeled by left-linear CSs, and computational strategies take advantage of this constructor condition (see [2,14]).

Example 2. Consider the following TRS borrowed from [8] defining the symbol $\div$, which encodes the division function between natural numbers.

$$\begin{array}{ll}
\texttt{0} \div \texttt{s(N) = 0} & \texttt{M} - \texttt{0 = M} \\
\texttt{s(M)} \div \texttt{s(N) = s((M}-\texttt{N)}\div\texttt{s(N))} & \texttt{s(M)} - \texttt{s(N) = M}-\texttt{N}
\end{array}$$

Consider the term $t = \mathtt{10!}\div\mathtt{0}$, which is a (non-constructor) head-normal form. Outermost-needed rewriting forces[1] the reduction of the first argument and evaluates `10!`, which is useless. The reason is that outermost-needed rewriting uses a data structure called *definitional tree* which encodes the branching selection structure existing in the rules without testing whether the rules associated to each branch could ever be matched to the term or not. A similar problem occurs when narrowing the term $\mathtt{X} \div \mathtt{0}$, since variable $\mathtt{X}$ is instantiated to `0` or `s`. However, neither instantiation is really necessary.

In [9], we proposed a solution to these two problems, namely the non-optimal evaluation for non-inductively sequential programs and the unnecessary evaluation in failing terms, which is based on a suitable extension of the demandedness notion associated to weakly outermost-needed rewriting and narrowing. The new strategies are called *natural rewriting* and *natural narrowing,* respectively. Our strategies incorporate a better treatment of demandedness and enjoy good computational properties; in particular, we show how to use them for computing (head-)normal forms and we prove they are conservative w.r.t. (weakly) outermost-needed rewriting and (weakly) outermost-needed narrowing. Moreover, we defined a new class of CSs called *inductively sequential preserving* where natural rewriting and narrowing preserve optimality for sequential parts of the program. This new class of CSs is based on the extension of the notion of inductive sequentiality from defined function symbols to terms and is larger than the class of inductively sequential CSs.

In this paper, we address how to implement natural rewriting and natural narrowing efficiently. After some preliminaries in Section 2, in Section 3, we provide a generalization of the notion of definitional tree, which we call *matching definitional tree.* In Section 4, we present how to reproduce natural rewriting and natural narrowing by traversing matching definitional trees. In Section 5, we show that it is possible to implement natural rewriting and narrowing efficiently by compiling to Prolog. Finally, Section 6 presents our conclusions.

2 Preliminaries

We assume some familiarity with term rewriting (see [16] for missing definitions) and narrowing (see [10] for missing definitions). Let $R \subseteq A \times A$ be a binary relation on a set A. We denote the reflexive closure of R by $R^=$, its transitive closure by R^+, and its reflexive and transitive closure by R^*. An element $a \in A$ is an R-normal form, if there exists no b such that $a\ R\ b$. We say that b is an R-normal form of a (written $a\ R^!\ b$), if b is an R-normal form and $a\ R^* b$.

Throughout the paper, $\mathcal{X}$ denotes a countable set of variables $\{\mathsf{x}, \mathsf{y}, \ldots\}$ and $\mathcal{F}$ denotes a *many-sorted* signature, i.e. a set of function symbols $\{\mathsf{f}, \mathsf{g}, \ldots\}$ grouped into sorts. We denote the set of terms built from $\mathcal{F}$ and $\mathcal{X}$ by $\mathcal{T}(\mathcal{F},\mathcal{X})$. A k-tuple $t_1,\ldots,t_k$ of terms is written $\bar{t}$. A term is said to be linear if it has

[1] Note that this behavior is independent of the fact that two possible definitional trees exist for symbol $\div$ (see [9, Example 21]).

no multiple occurrences of a single variable. Let $\mathit{Subst}(\mathcal{T}(\mathcal{F},\mathcal{X}))$ denote the set of substitutions. We denote by *id* the "identity" substitution: $id(x) = x$ for all $x \in \mathcal{X}$. Terms are ordered by the preorder $\leq$ of "relative generality", i.e. $s \leq t$ if there exists σ s.t. $\sigma(s) = t$. Term t is a variant of s if $t \leq s$ and $s \leq t$. A *most general unifier* (*mgu*) of t, s is a unifier σ such that for each unifier σ' of t, s there exists θ such that $\sigma' = \theta \circ \sigma$.

By $\mathcal{P}os(t)$ we denote the set of positions of a term t. Given a set $S \subseteq \mathcal{F} \cup \mathcal{X}$, $\mathcal{P}os_S(t)$ denotes positions in t where symbols in S occur. We denote the root position by Λ. Given positions p, q, we denote its concatenation as $p.q$. Positions are ordered by the standard prefix ordering $\leq$. The subterm at position p of t is denoted as $t|_p$, and $t[s]_p$ is the term t with the subterm at position p replaced by s. The symbol labeling the root of t is denoted as $root(t)$.

A rewrite rule is an ordered pair (l, r), written $l \to r$, with $l, r \in \mathcal{T}(\mathcal{F},\mathcal{X})$ and $l \notin \mathcal{X}$. The left-hand side *(lhs)* of the rule is l and the right-hand side *(rhs)* of the rule is r. A TRS is a pair $\mathcal{R} = (\mathcal{F}, R)$ where R is a set of rewrite rules. $L(\mathcal{R})$ denotes the set of *lhs*'s of $\mathcal{R}$. A TRS $\mathcal{R}$ is left-linear if for all $l \in L(\mathcal{R})$, l is a linear term. Given $\mathcal{R} = (\mathcal{F}, R)$, we take $\mathcal{F}$ as the disjoint union $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ of symbols $c \in \mathcal{C}$, called *constructors* and symbols $f \in \mathcal{D}$, called *defined functions,* where $\mathcal{D} = \{root(l) \mid l \to r \in R\}$ and $\mathcal{C} = \mathcal{F} - \mathcal{D}$. A pattern is a term $f(l_1, \ldots, l_k)$ where $f \in \mathcal{D}$ and $l_i \in \mathcal{T}(\mathcal{C},\mathcal{X})$, for $1 \leq i \leq k$. A TRS $\mathcal{R} = (\mathcal{C} \uplus \mathcal{D}, R)$ is a constructor system (CS) if all lhs's are patterns. A term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ rewrites to s (at position p), written $t \xrightarrow{p}_{l\to r} s$ (or just $t \to s$), if $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$, for some rule $l \to r \in R$, $p \in \mathcal{P}os(t)$ and substitution σ. The subterm $\sigma(l)$ in t is called a redex. On the other hand, a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ narrows to s (at position p with substitution σ), written $t \leadsto_{\{p,\sigma,l\to r\}} s$ (or just $t \leadsto_\sigma s$) if p is a non-variable position in t and $\sigma(t) \xrightarrow{p}_{l\to r} s$. A term t is a head-normal form (or root-stable) if it cannot be reduced to a redex.

3 Matching Definitional Trees

In order to efficiently implement natural rewriting and narrowing, we must integrate the demandedness notion of natural rewriting and narrowing [9] into a statically built structure, as happens in weakly outermost-needed rewriting and narrowing which use definitional trees [3]. Thus, we define *matching definitional trees.* First, we recall the definition of a (generalized) definitional tree.

Definition 1. *[4] $\mathcal{T}$ is a generalized definitional tree, or gdt, with pattern π iff one of the following cases holds:*

$\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$ *where π is a pattern, o is the occurrence (called* inductive*) of a variable of π, the sort of $\pi|_o$ has different constructors $c_1, \ldots, c_k$ for $k > 0$, and for all i in $\{1, \ldots, k\}$, $\mathcal{T}_i$ is a gdt with pattern $\pi[c_i(\overline{x})]_o$, where $\overline{x}$ are new distinct variables.*

$\mathcal{T} = leaf(\pi, l \to r)$ *where π is a pattern and $l \to r$ is a rule such that π and l are variants.*

$\mathcal{T} = or(\mathcal{T}_1, \ldots, \mathcal{T}_k)$ *where $k > 1$ and each $\mathcal{T}_i$ is a gdt with pattern π.*

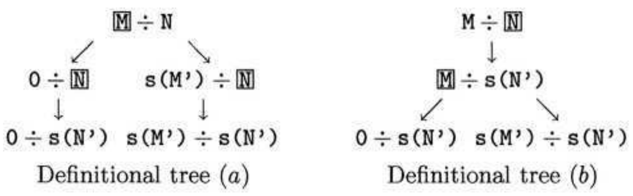


Fig. 1. The two possible definitional trees for the symbol ÷

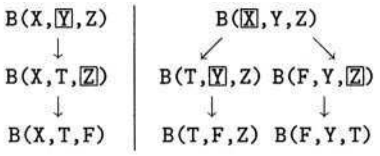


Fig. 2. A partition of definitional trees for the symbol B

A *definitional tree* is a generalized definitional tree without *or*-nodes. A *parallel definitional tree* is a generalized definitional tree where only one *or*-node is allowed at the top of the tree. A defined symbol f is called *inductively sequential* if there exists a definitional tree $\mathcal{T}$ with pattern $f(x_1,\ldots,x_k)$ (where $x_1,\ldots,x_k$ are different variables) whose leaves contain all and only the rules defining f. In this case, we say that $\mathcal{T}$ is a definitional tree for f, denoted as $\mathcal{T}_f$. A left-linear CS $\mathcal{R}$ is *inductively sequential* if all its defined function symbols are inductively sequential. It is often convenient and simplifies understanding to provide a graphical representation of definitional trees as a tree of patterns where the inductive position in *branch*-nodes is surrounded by a box [3].

Example 3. The symbol ÷ in Example 2 is inductively sequential since there exists a definitional tree for pattern M ÷ N. Figure 1 shows the two possible definitional trees.

When a function symbol is not inductively sequential, a parallel definitional tree, instead of a definitional tree, is obtained. The graphical representation of a parallel definitional tree corresponds to a set of definitional trees.

Example 4. The symbol B in Example 1 is non-inductively sequential, since a rule partition that splits its rules into subsets that are inductively sequential is necessary. Figure 2 shows a parallel definitional tree for symbol B.

Definitional trees are used by (weakly) outermost-needed rewriting and narrowing as a finite state automaton to compute the demanded positions to be reduced (in line with [13,15]). However, definitional trees merge two different processes: the pattern matching process and the evaluation process through demanded positions. These two processes coincide in the case of inductively sequential and non-failing terms because the evaluation sequence corresponds to the pattern matching sequence, but do not coincide for non-inductively sequential

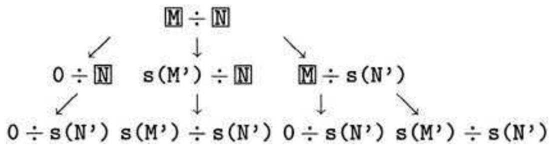


Fig. 3. A matching definitional tree for symbol $\div$

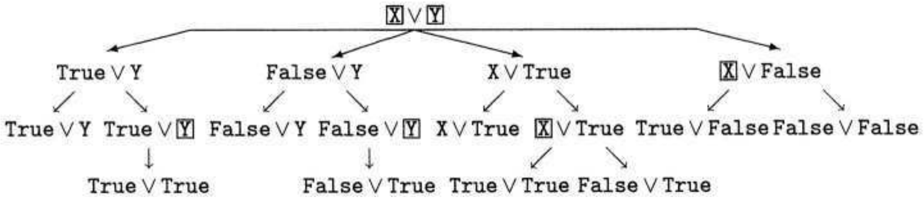


Fig. 4. A matching definitional tree for symbol $\vee$

or failing terms where the pattern matching process may determine a (possibly better) evaluation sequence (as shown in Examples 1 and 2). We define a *matching definitional tree* as a definitional tree where more than one inductive position is allowed for *branch*-nodes.

Definition 2. *$\mathcal{T}$ is a matching definitional tree, or mdt, with pattern π iff one of the following cases holds:*

$\mathcal{T} = branch(\pi, (o_1, \mathcal{T}_1^1, \ldots, \mathcal{T}_{k_1}^1), \ldots, (o_n, \mathcal{T}_1^n, \ldots, \mathcal{T}_{k_n}^n))$ *where π is a pattern, $o_1, \ldots, o_n$ with $n > 0$ are occurrences of variables of π, the sort of $\pi|_{o_i}$ has different constructors $c_1^i, \ldots, c_{k_i}^i$ for i in $\{1, \ldots, n\}$ and $k_i > 0$, and for all j in $\{1, \ldots, k_i\}$, $\mathcal{T}_j^i$ is a mdt with pattern $\pi[c_j^i(\overline{x})]_{o_i}$, where $\overline{x}$ are new distinct variables.*

$\mathcal{T} = leaf(\pi, l \to r)$ *where π is a pattern and $l \to r$ is a rule such that $l \leq \pi$.*

$\mathcal{T} = or(\mathcal{T}_1, \ldots, \mathcal{T}_k)$ *where $k > 1$ and each $\mathcal{T}_i$ is a mdt with pattern π.*

For each defined symbol f in a left-linear CS, we can build a matching definitional tree $\mathcal{T}$ with pattern $f(x_1, \ldots, x_k)$ (where $x_1, \ldots, x_k$ are different variables) whose leaves contain all and only the rules defining f. The graphical representation is similar to that of a definitional tree with the difference that it is possible to have more than one inductive position in *branch*-nodes and *or*-nodes are identified as nodes which do not have any boxed position. We denote *branch*-nodes that have only one position o as simply $branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$ (as happens in gdt's).

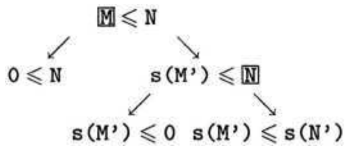


Fig. 5. The definitional tree and matching definitional tree for symbol $\leqslant$

Example 5. Figure 3 represents the following[2] matching definitional tree which could be associated to the symbol $\div$ in Example 2:

$$\begin{array}{l} \mathit{branch}(\mathtt{M} \div \mathtt{N}, (1, \mathit{branch}(\mathtt{0} \div \mathtt{N}, 2, \mathit{leaf}(\mathtt{0} \div \mathtt{s(N')})), \\ \qquad \mathit{branch}(\mathtt{s(M')} \div \mathtt{N}, 2, \mathit{leaf}(\mathtt{s(M')} \div \mathtt{s(N')})), \\ \quad (2, \mathit{branch}(\mathtt{M} \div \mathtt{s(N')}, 1, \mathit{leaf}(\mathtt{0} \div \mathtt{s(N')}), \\ \qquad\qquad \mathit{leaf}(\mathtt{s(M')} \div \mathtt{s(N')})))) \end{array}$$

Example 6. Consider the following parallel-or TRS from [9]:

```
True ∨ X = True      X ∨ True = True      False ∨ X = X
```

Figure 4 denotes the following *mdt* associated to the symbol $\vee$:

$$\begin{array}{l} \mathit{branch}(\mathtt{X} \vee \mathtt{Y}, (1, \mathit{or}(\mathit{leaf}(\mathtt{True} \vee \mathtt{Y}), \\ \qquad \mathit{branch}(\mathtt{True} \vee \mathtt{Y}, 2, \mathit{leaf}(\mathtt{True} \vee \mathtt{True}))), \\ \quad \mathit{or}(\mathit{leaf}(\mathtt{False} \vee \mathtt{Y}), \\ \qquad \mathit{branch}(\mathtt{False} \vee \mathtt{Y}, 2, \mathit{leaf}(\mathtt{False} \vee \mathtt{True})))), \\ \quad (2, \mathit{or}(\mathit{leaf}(\mathtt{X} \vee \mathtt{True}), \\ \qquad \mathit{branch}(\mathtt{X} \vee \mathtt{True}, 1, \mathit{leaf}(\mathtt{True} \vee \mathtt{True}), \\ \qquad\qquad \mathit{leaf}(\mathtt{False} \vee \mathtt{True}))), \\ \quad \mathit{branch}(\mathtt{X} \vee \mathtt{False}, 1, \mathit{leaf}(\mathtt{True} \vee \mathtt{False}), \\ \qquad\qquad \mathit{leaf}(\mathtt{False} \vee \mathtt{False})))), \end{array}$$

Note that every definitional tree is also a matching definitional tree.

Example 7. Consider the following defined symbol $\leqslant$, which is an example commonly used to refer to definitional trees [5]:

```
0 ⩽ N = True        s(M) ⩽ 0 = False        s(M) ⩽ s(N) = M ⩽ N
```

Since this symbol has only one possible definitional tree (instead of symbol $\div$ of Example 2 which has two), the associated definitional tree of Figure 5 corresponds to its matching definitional tree:

$$\begin{array}{l} \mathit{branch}(\mathtt{M} \leqslant \mathtt{N}, 1, \mathit{leaf}(\mathtt{0} \leqslant \mathtt{N}), \\ \qquad \mathit{branch}(\mathtt{s(M')} \leqslant \mathtt{N}, 2, \mathit{leaf}(\mathtt{s(M')} \leqslant \mathtt{0}), \mathit{leaf}(\mathtt{s(M')} \leqslant \mathtt{s(N')}))) \end{array}$$

[2] In this paper, we often omit the rule in a leaf node for simplicity.

4 Evaluation through Matching Definitional Trees

In inductively sequential left-linear CSs, the order of evaluation is determined by a mapping φ which implements the strategy by traversing definitional trees as a finite state automaton, in order to compute the demanded positions to be reduced [3].

In order to implement natural rewriting efficiently, we define a mapping $\mathtt{mt}$ which traverses a matching definitional tree as a pattern matching finite state automaton and returns the set of demanded positions associated to a concrete *branch*-node if all its inductive positions are rooted by non-constructor symbols.

Definition 3. *The function* $\mathtt{mt}$ *takes two arguments: an operation-rooted term, t, and a mdt, $\mathcal{T}$, such that $pattern(\mathcal{T})$ and t unify. The function* $\mathtt{mt}$ *yields a set of tuples of the form (p, R), where p is a position of t, and R is a rule $l \to r$ of $\mathcal{R}$. The function* $\mathtt{mt}$ *is defined as follows:*

$$\mathtt{mt}(t,\mathcal{T}) \ni \begin{cases} (\Lambda, R) & \textit{if } \mathcal{T} = leaf(\pi, R); \\ (p,R) & \textit{if } \mathcal{T} = or(\mathcal{T}_1,\ldots,\mathcal{T}_k) \textit{ and } (p,R) \in \mathtt{mt}(t,\mathcal{T}_i) \textit{ for some } i; \\ (p,R) & \textit{if } \mathcal{T} = branch(\pi,(o_1,\mathcal{T}_1^1,\ldots,\mathcal{T}_{k_1}^1),\ldots,(o_n,\mathcal{T}_1^n,\ldots,\mathcal{T}_{k_n}^n)), \\ & \quad root(t|_{o_i}) \in \mathcal{C} \textit{ for some } 1 \le i \le n,\ pattern(\mathcal{T}_j^i) \le t \\ & \quad \textit{for some } 1 \le j \le k_i, \textit{ and } (p,R) \in \mathtt{mt}(t,\mathcal{T}_j^i); \\ (o_i.p,R) & \textit{if } \mathcal{T} = branch(\pi,(o_1,\mathcal{T}_1^1,\ldots,\mathcal{T}_{k_1}^1),\ldots,(o_n,\mathcal{T}_1^n,\ldots,\mathcal{T}_{k_n}^n)), \\ & \quad root(t|_{o_i}) \in \mathcal{D} \textit{ for all } 1 \le i \le n, \\ & \quad \textit{and } (p,R) \in \mathtt{mt}(t|_{o_i},\mathcal{T}_{root(t|_{o_i})}) \textit{ for some } 1 \le i \le n. \end{cases}$$

Example 8. Consider the TRS and term $t = \mathtt{10!} \div \mathtt{0}$ in Example 2 together with the matching definitional tree $\mathcal{T}_\div$ in Example 5 (and Figure 3). We have $\mathtt{mt}(t,\mathcal{T}_\div) = \varnothing$ since the two inductive positions of the topmost *branch*-node are not operation-rooted and no subtree can be selected: the two first subtrees because position 1 is operation-rooted and the third one because it needs an $\mathtt{s}$ constructor symbol at position 2, instead of $\mathtt{0}$.

Outermost-needed narrowing is determined by a mapping λ which implements the strategy by traversing definitional trees [5]. Here, we define a mapping $\mathtt{mnt}$ which extends $\mathtt{mt}$ to narrowing as λ extends φ.

Definition 4. *The function* $\mathtt{mnt}$ *takes two arguments: an operation-rooted term, t, and a mdt, $\mathcal{T}$, such that $pattern(\mathcal{T})$ and t unify. The function* $\mathtt{mnt}$ *yields a set of triples of the form (p, R, σ), where p is a position of t, R is a rule $l \to r$ of $\mathcal{R}$, and σ is a unifier of $pattern(\mathcal{T})$ and t. The function* $\mathtt{mnt}$ *is defined as follows:*

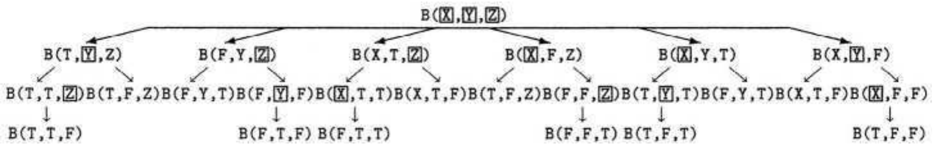

B(X,Y,Z)
B(T,Y,Z) B(F,Y,Z) B(X,T,Z) B(X,F,Z) B(X,Y,T) B(X,Y,F)
B(T,T,Z) B(T,F,Z) B(F,Y,T) B(F,Y,F) B(X,T,T) B(X,T,F) B(T,F,Z) B(F,F,Z) B(T,Y,T) B(F,Y,T) B(X,T,F) B(X,F,F)
B(T,T,F) B(F,T,F) B(F,T,T) B(F,F,T) B(T,F,T) B(T,F,F)

Fig. 6. A matching definitional tree for symbol B

$$
\mathtt{mnt}(t,\mathcal{T}) \ni \begin{cases}
(\Lambda, R, \sigma) & \text{if } \mathcal{T} = \mathit{leaf}(\pi, R) \text{ and } \sigma = \mathit{mgu}(\pi, t);\\
(p, R, \sigma) & \text{if } \mathcal{T} = \mathit{or}(\mathcal{T}_1, \ldots, \mathcal{T}_k) \text{ and } (p,R,\sigma) \in \mathtt{mnt}(t, \mathcal{T}_i) \text{ for some } i;\\
(p, R, \sigma) & \text{if } \mathcal{T} = \mathit{branch}(\pi, (o_1, \mathcal{T}_1^1, \ldots, \mathcal{T}_{k_1}^1), \ldots, (o_n, \mathcal{T}_1^n, \ldots, \mathcal{T}_{k_n}^n)),\\
& \quad \mathit{root}(t|_{o_i}) \in \mathcal{C} \text{ for some } 1 \leq i \leq n,\ \mathit{pattern}(\mathcal{T}_j^i) \leq t\\
& \quad \text{for some } 1 \leq j \leq k_i, \text{ and } (p, R, \sigma) \in \mathtt{mnt}(t, \mathcal{T}_j^i);\\
(o_i.p, R, \sigma) & \text{if } \mathcal{T} = \mathit{branch}(\pi, (o_1, \mathcal{T}_1^1, \ldots, \mathcal{T}_{k_1}^1), \ldots, (o_n, \mathcal{T}_1^n, \ldots, \mathcal{T}_{k_n}^n)),\\
& \quad \mathit{root}(t|_{o_i}) \notin \mathcal{C} \text{ for all } 1 \leq i \leq n,\ \mathit{root}(t|_{o_i}) \in \mathcal{D} \text{ for}\\
& \quad \text{some } 1 \leq i \leq n, \text{ and } (p, R, \sigma) \in \mathtt{mnt}(t|_{o_i}, \mathcal{T}_{\mathit{root}(t|_{o_i})});\\
(p, R, \theta \circ \sigma) & \text{if } \mathcal{T} = \mathit{branch}(\pi, (o_1, \mathcal{T}_1^1, \ldots, \mathcal{T}_{k_1}^1), \ldots, (o_n, \mathcal{T}_1^n, \ldots, \mathcal{T}_{k_n}^n)),\\
& \quad \mathit{root}(t|_{o_i}) \notin \mathcal{C} \text{ for all } 1 \leq i \leq n,\ t|_{o_i} \in \mathcal{X} \text{ for some}\\
& \quad 1 \leq i \leq n,\ \theta = \mathit{mgu}(t, \mathit{pattern}(\mathcal{T}_j^i)) \text{ for some}\\
& \quad 1 \leq j \leq k_i, \text{ and } (p, R, \sigma) \in \mathtt{mnt}(\theta(t), \mathcal{T}_j^i).
\end{cases}
$$

Example 9. Consider the TRS and term $t = \mathtt{B(X,B(F,T,T),F)}$ in Example 1. The *mdt* $\mathcal{T}_\mathtt{B}$ is depicted in Figure 6. Then, although the rule B(F,X,T)=T appears four times in $\mathcal{T}_\mathtt{B}$, we have $\mathtt{mnt}(t, \mathcal{T}_\mathtt{B}) = \{(2, \mathtt{B(F,X,T)=T}, \mathit{id})\}$ since only the rightmost branch associated to inductive position 3 and constructor symbol F at the third argument is selected.

It is worth noting that Definitions 3 and 4 boil down to outermost-needed rewriting [3] and outermost-needed narrowing [5], respectively, when definitional trees are considered (i.e. matching definitional trees where each *branch*-node has only one inductive position and no *or*-node is included).

In the following, we prove that natural rewriting and natural narrowing can be appropriately reproduced by using Definitions 3 and 4 and matching definitional trees.

4.1 Natural Rewriting

In [9], we provided a suitable refinement of the demandedness notion associated to *outermost-needed rewriting* which uses some basic definitions from the on-demand evaluation presented in [1], dealing with syntactic strategy annotations.

Definition 5. [1] *Given* $t, l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, *we let* $\mathcal{P}os_{\neq}(t, l) = \mathit{minimal}_{\leq}(\{p \in \mathcal{P}os(t) \cap \mathcal{P}os_{\mathcal{F}}(l) \mid \mathit{root}(l|_p) \neq \mathit{root}(t|_p)\})$.

Definition 6. [1] *We define the set of* demanded *positions of term* $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ *w.r.t.* $l \in \mathcal{T}(\mathcal{F},\mathcal{X})$ *(a lhs of a rule defining* $root(t)$*), i.e. the set of (positions of) maximal disagreeing subterms as:*

$$DP_l(t) = \begin{cases} \mathcal{P}os_{\neq}(t,l) & \text{if } \mathcal{P}os_{\neq}(t,l) \cap \mathcal{P}os_{\mathcal{C}}(t) = \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

Example 10. Consider again Example 1 and terms $t = \texttt{B(B(T,F,T),B(F,T,T),F)}$ and $t' = \texttt{B(X,B(F,T,T),F)}$. We have (identically for t'): $DP_{\texttt{B(T,F,X)}}(t) = \{1,2\}$, $DP_{\texttt{B(F,X,T)}}(t) = \varnothing$, and $DP_{\texttt{B(X,T,F)}}(t) = \{2\}$.

Note that the restriction of disagreeing positions to positions with non-constructor symbols (*defined* symbols in the case of rewriting) disables the evaluation of subterms which could never help to produce a redex (see [1,2,14]).

In [7], Antoy and Lucas showed that modern functional (logic) languages include evaluation strategies that consider elaborated definitions of 'demandedness' which are, in fact, related. They introduced the informal idea that some rewriting strategies select 'popular' demanded positions over the available set. In [9], we formalized a notion of popularity of demanded positions which makes the difference by counting the number of times a position is demanded by some rule, and then selects the most (frequently) demanded positions covering all eventually applicable rules.

Definition 7. [9] *We define the multiset of demanded positions of a term* $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ *w.r.t. TRS* $\mathcal{R}$ *as* $DP_{\mathcal{R}}(t) = \oplus\{DP_l(t) \mid l \to r \in \mathcal{R} \wedge root(t) = root(l)\}$ *where* $M_1 \oplus M_2$ *is the union of multisets* M_1 *and* M_2.

Example 11. Continuing Example 10, we have $DP_{\mathcal{R}}(t) = DP_{\mathcal{R}}(t') = \{1,2,2\}$.

Definition 8 (Demanded positions). [9] *We define the set of demanded positions of a term* $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ *w.r.t. TRS* $\mathcal{R}$ *as*

$$DP^{\mathrm{m}}_{\mathcal{R}}(t) = \{p \in DP_{\mathcal{R}}(t) \mid \exists l \in L(\mathcal{R}).p \in DP_l(t) \text{ and } \forall q \in DP_{\mathcal{R}}(t) : p <_{DP_{\mathcal{R}}(t)} q \Rightarrow l|_q \in \mathcal{X}\}$$

where $x <_M y$ *denotes that the number of occurrences of* x *in the multiset* M *is less than the number of occurrences of* y.

Example 12. Continuing Example 11, we have $DP^{\mathrm{m}}_{\mathcal{R}}(t) = DP^{\mathrm{m}}_{\mathcal{R}}(t') = \{2\}$ since even though position 1 is demanded, the redex at position 2 is the most frequently demanded.

Example 13. Consider the TRS $\mathcal{R}$ of Example 6 and the term $t = \texttt{(True} \vee \texttt{False)} \vee \texttt{(True} \vee \texttt{False)}$. We have $DP^{\mathrm{m}}_{\mathcal{R}}(t) = \{1,2\}$ since position 1 alone does not cover the rule $\texttt{X} \vee \texttt{True = True}$.

The following definition establishes the strategy used in natural rewriting for selecting demanded positions.

Definition 9 (Natural rewriting). [9] *Given a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ and a TRS $\mathcal{R}$, $\mathfrak{m}(t)$ is defined as the smallest set satisfying*

$$\mathfrak{m}(t) \ni \begin{cases} (\Lambda, l \to r) & \text{if } l \to r \in \mathcal{R} \text{ and } l \leq t \\ (p.q, l \to r) & \text{if } p \in DP^{\mathfrak{m}}_{\mathcal{R}}(t) \text{ and } (q, l \to r) \in \mathfrak{m}(t|_p) \end{cases}$$

We consider $\mathfrak{m}$ simply as a function returning positions if the rules selected for reduction are not relevant or they are clear from the context. We say term t reduces by *natural rewriting* to term s, denoted by $t \overset{\mathfrak{m}}{\to}_{\{p,l\to r\}} s$ (or simply $t \overset{\mathfrak{m}}{\to} s$) if $(p, l \to r) \in \mathfrak{m}(t)$.

Example 14. Continuing Example 12, we have $\mathfrak{m}(t) = \{2\}$ and the only possible natural rewriting step is: $\texttt{B(B(T,F,T),}\underline{\texttt{B(F,T,T)}}\texttt{,F)} \overset{\mathfrak{m}}{\to} \underline{\texttt{B(B(T,F,T),T,F)}}$

Example 15. Continuing Example 13, it is clear that $\mathfrak{m}(t) = \{1,2\}$ and, thus, there exist two possible natural rewriting steps:

$$\underline{\texttt{(True} \vee \texttt{False)}} \vee \texttt{(True} \vee \texttt{False)} \overset{\mathfrak{m}}{\to} \texttt{True} \vee \texttt{(True} \vee \texttt{False)}$$
$$\texttt{(True} \vee \texttt{False)} \vee \underline{\texttt{(True} \vee \texttt{False)}} \overset{\mathfrak{m}}{\to} \texttt{(True} \vee \texttt{False)} \vee \texttt{True}$$

Now, we prove that natural rewriting is appropriately reproduced by using matching definitional trees and Definition 3. First, we give a class of matching definitional trees, called *safe matching definitional trees,* where the inductive positions of a *branch*-node correspond to the demanded positions obtained by the operator $DP^{\mathfrak{m}}_{\mathcal{R}}$. In the following, we denote the set of rules appearing at leaves of a tree as $leaves(\mathcal{T})$. Similarly, given a TRS $\mathcal{R}$, we denote the set of rules which are instantiations of a pattern π as $rules_{\mathcal{R}}(\pi) = \{l \to r \in \mathcal{R} \mid \pi \leq l\}$. A *mdt* $\mathcal{T}$ is called *total* if it contains all and only the rules eventually applicable to the pattern of $\mathcal{T}$ (i.e. $rules_{\mathcal{R}}(pattern(\mathcal{T})) = leaves(\mathcal{T})$).

Definition 10. *Let $\mathcal{R} = (\mathcal{F}, R)$ be a left-linear CS and $\mathcal{T}$ be a matching definitional tree for symbol $f \in \mathcal{D}$ with pattern π. We say $\mathcal{T}$ is a* safe *matching definitional tree for symbol f iff one of the following cases holds:*

$\mathcal{T} = branch(\pi, (o_1, \mathcal{T}^1_1, \ldots, \mathcal{T}^1_{k_1}), \ldots, (o_n, \mathcal{T}^n_1, \ldots, \mathcal{T}^n_{k_n}))$, $DP^{\mathfrak{m}}_{\mathcal{R}}(\pi) = \{o_1, \ldots, o_n\}$, and $\mathcal{T}^1_1, \ldots, \mathcal{T}^1_{k_1}, \ldots, \mathcal{T}^n_1, \ldots, \mathcal{T}^n_{k_n}$ are total, safe mdt's.
$\mathcal{T} = leaf(\pi, l \to r)$ and $l \to r \in \mathcal{R}$.
$\mathcal{T} = or(\mathcal{T}_1, \ldots, \mathcal{T}_k)$, $\mathcal{T}_1, \ldots, \mathcal{T}_k$ are safe mdt's, $\mathcal{T}_1, \ldots, \mathcal{T}_{k-1}$ are leaves with pattern π, and each rule $l \to r \in leaves(\mathcal{T})$ is unique among $\mathcal{T}_1, \ldots, \mathcal{T}_k$.

Note that the matching definitional trees of Figures 3, 4, 5 and 6 are safe *mdt*'s whereas the two definitional trees of Figure 1 are not. Indeed, it is worth noting that the construction of safe *mdt*'s is easily achievable by using the function $DP^{\mathfrak{m}}_{\mathcal{R}}$ for *branch*-nodes and creating *or*-nodes when a pattern is a redex and also has demanded positions. Furthermore, there is only one safe *mdt* per function symbol, due to the use of $DP^{\mathfrak{m}}_{\mathcal{R}}$. Now, we prove that both definitions of natural rewriting coincide for safe *mdt*'s.

Theorem 1. *Let $\mathcal{R} = (\mathcal{F}, R)$ be a left-linear CS, $f \in \mathcal{D}$, $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ s.t. $root(t) = f$, and $\mathcal{T}_f$ be a safe mdt for f. Then, $\mathfrak{mt}(t, \mathcal{T}_f) = \mathfrak{m}(t)$.*

4.2 Natural Narrowing

The following definition establishes the strategy used in natural narrowing for allowing narrowing steps.

Definition 11 (Natural narrowing). [9] *Given a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ and a TRS $\mathcal{R}$, $\mathfrak{mn}(t)$ is defined*[3] *as the smallest set satisfying*

$$\mathfrak{mn}(t) \ni \begin{cases} (\Lambda, id, l{\to}r) \;\; \textit{if } l \to r \in \mathcal{R} \textit{ and } l \le t \\ (p.q, \theta, l{\to}r) \;\; \textit{if } p \in DP^{\mathfrak{m}}_{\mathcal{R}} \cap \mathcal{P}os_{\mathcal{D}}(t) \textit{and } (q, \theta, l{\to}r) \in \mathfrak{mn}(t|_p) \\ (p, \theta\circ\sigma, l{\to}r) \;\; \textit{if } p \in DP^{\mathfrak{m}}_{\mathcal{R}}.t|_p{=}x \in \mathcal{X}, c \textit{ is in the sort of } x, \sigma(x){=}c(\overline{w}), \\ \qquad\qquad \overline{w} \textit{ are fresh variables, and } (p, \theta, l{\to}r) \in \mathfrak{mn}(\sigma(t)) \end{cases}$$

We say term t narrows by *natural narrowing* to term s at position p using substitution σ, denoted by $t \overset{\mathfrak{m}}{\rightsquigarrow}_{\{p,l\to r,\sigma\}} s$ (or simply $t \overset{\mathfrak{m}}{\rightsquigarrow}_{\sigma} s$), if $(p, \sigma, l \to r) \in \mathfrak{mn}(t)$.

Example 16. Continuing Example 12, we have $\mathfrak{mn}(t') = \{2\}$ and, thus, the natural narrowing sequence $\texttt{B(X,}\underline{\texttt{B(F,T,T)}}\texttt{,F)} \overset{\mathfrak{m}}{\rightsquigarrow}_{id} \underline{\texttt{B(X,T,F)}} \overset{\mathfrak{m}}{\rightsquigarrow}_{id} \texttt{T}$

Similarly to the rewriting case, we prove that both definitions of natural narrowing coincide for safe *mdt*'s.

Theorem 2. *Let $\mathcal{R} = (\mathcal{F}, R)$ be a left-linear CS, $f \in \mathcal{D}$, $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ s.t. $root(t) = f$, and $\mathcal{T}_f$ be a safe mdt for f. Then, $\mathfrak{mnt}(t, \mathcal{T}_f) = \mathfrak{mn}(t)$.*

5 Implementation

In this section, we show that natural rewriting and natural narrowing can be implemented efficiently by reusing modern state-of-the-art compilation techniques developed for (weakly) outermost-needed rewriting and narrowing. Curry [12] is a multiparadigm programming language based on weakly outermost-needed narrowing which combines purely functional programming, purely logic programming, and concurrent (logic) programming in a seamless way. The Curry2Prolog compiler [6] included into the Curry development system PAKCS [11] is one of the fastest Curry implementations available nowadays. The following example helps to understand how the compilation of Curry programs to Prolog [6] works.

[3] This definition is a simplification of that of [9] and differs only in the selection of demanded positions rooted by defined symbols. Note that both definitions satisfy the same properties described in [9].

Example 17. Consider the TRS of Example 2. Basically, the technique in [6] translates a function symbol and its associated definitional tree to `case` expressions and, then, compiles these `case` expressions to Prolog. For instance, Curry2Prolog uses the definitional tree (a) of Figure 1 and translates it to the following (auxiliary) case expressions, where the constructor symbol `0` is represented by the constructor symbol `z`:

```
M ÷ N = case M of z     -> (case N of s(N') -> z)
                  s(M') -> (case N of s(N') -> s(M'-N' ÷ s(N')))
```

Its translation to Prolog according to [6] is as follows:

```
:- block ÷(?,?,?,-,?).
÷(M,N,Result,Ein,Eout):-hnf(M,HM,Ein,E1),÷_1(HM,N,Result,E1,EOut).

:- block ÷_1(?,?,?,-,?).
÷_1(z,N,Result,Ein,Eout):-hnf(N,HN,Ein,E1),÷_1_z_2(HN,Result,E1,Eout).
÷_1(s(M),N,Res,Ein,Eout):-hnf(N,HN,Ein,E1),÷_1_s_2(HN,M,Res,E1,Eout).

:- block ÷_1_z_2(?,?,-,?).
÷_1_z_2(s(N),z,E,E).

:- block ÷_1_s_2(?,?,?,-,?).
÷_1_s_2(s(N),M,s(÷(-(M,share(N,EN,RN)),s(share(N,EN,RN)))),E,E).
```

As is usual in the transformation of functions to predicates, the result of the function call is included as an extra argument called `Result`. Since the computational model of Curry allows concurrent executions of calls, the Curry2Prolog compiler introduces `block` Prolog declarations and arguments `Ein` and `Eout` for each function symbol in order to control whether a function call is suspended or not (see [6]). Moreover, a call `hnf(T, HT, Ein, Eout)` is responsible for obtaining the head normal form `HT` of a term `T`. Finally, Curry2Prolog implements the sharing of variables using an extra symbol `share`. A term `share(T,ET,RT)` contains the shared term `T`, `ET` (which indicates whether `T` has already been evaluated), and `RT` (which is the result of `T`).

In this section, we argue that it is possible to use the technique in [6] to translate matching definitional trees and natural rewriting and narrowing strategies to Prolog. The main difference between a *gdt* and a *mdt* is that the latter has more than one inductive position. Hence, we can use the previous technique for translating each inductive position and its set of subtrees to Prolog and include some extra rules when more than one inductive position exist. These extra rules check sequentially whether each inductive position is constructor-rooted or not and only start an evaluation if none of the inductive positions are constructor-rooted (according to Definition 4). These rules use a predicate `checkC` that succeeds when its argument is rooted by a constructor symbol. We show how the translation works in the following example.

Example 18. Consider the TRS in Example 2 and the safe *mdt* in Example 5 (and Figure 3). We can apply the technique in Example 17 to each part of the root *branch*-node driven by an inductive position, and encode the pattern matching process into an appropriate set of rules (using predicate `hnf`). Then we add the

Table 1. Runtimes (in ms.) of different calls within Prolog compiled programs

Benchmark	Goal	PAKCS	PAKCS with Natural Narrowing
$\div$	10! $\div$ 10	26632	27345
	$\perp \div$ 0	∞	0
$\leqslant$	10! $\leqslant$ 10!	31360	31212

necessary rules (using predicate checkC) to provide the behavior associated to the natural narrowing strategy. Note that no extra rules (using checkC) are necessary when there exists only one inductive position in a *branch*-node. The compilation to Prolog yields:

```
checkC(A):-var(A),!,fail.
checkC(z).
checkC(s(_)).

:- block ÷(?,?,?,-,?).
÷(M,N,Result,Ein,Eout):-checkC(M),!,÷_1(M,N,Result,Ein,Eout).
÷(M,N,Result,Ein,Eout):-checkC(N),!,÷_2(N,M,Result,Ein,Eout).
÷(M,N,Result,Ein,Eout):-hnf(M,HM,Ein,E1),÷_1(HM,N,Result,E1,Eout).

:- block ÷_1(?,?,?,-,?).
÷_1(z,N,Result,Ein,Eout):-hnf(N,HN,Ein,E1),÷_1_z_2(HN,Result,E1,Eout).
÷_1(s(M),N,Res,Ein,Eout):-hnf(N,HN,Ein,E1),÷_1_s_2(HN,M,Res,E1,Eout).

:- block ÷_1_z_2(?,?,-,?).
÷_1_z_2(s(N),z,E,E).

:- block ÷_1_s_2(?,?,-,?).
÷_1_s_2(s(N),M,s(÷(-(M,share(N,EN,RN)),s(share(N,EN,RN)))),E,E).

:- block ÷_2(?,?,?,-,?).
÷_2(s(N),M,Res,Ein,Eout):-hnf(M,HM,Ein,E1),÷_2_s_1(HM,N,Res,E1,Eout).

:- block ÷_2_s_1(?,?,-,?).
÷_2_s_1(z,N,z,E,E).
÷_2_s_1(s(M),N,s(÷(-(M,share(N,EN,RN)),s(share(N,EN,RN)))),E,E).
```

Here, the predicate checkC is used to perform the constructor-rooted test on all the inductive positions. Note that we do not allow backtracking (using the Prolog cut !) in the selection of an inductive position and a subtree. Note also that a simple optimization is directly performed in the previous example: "when two inductive positions are demanded by the same set of rules, it is sufficient to evaluate only one of them." Thus, only position 1 is evaluated when both 1 and 2 are demanded (see the third clause of predicate $\div$).

Finally, Table 1 shows that the compiled Prolog code of the natural narrowing strategy using matching definitional trees does not introduce any appreciable overhead while providing better computational properties. It shows the average in milliseconds of 10 executions measured on a Pentium III machine running Red Hat 7.2. The Prolog code for symbol $\div$ in the third column corresponds to the Prolog code in Example 17, whereas the Prolog code for the fourth column corresponds to the Prolog code in Example 18. However, the Prolog code for symbol $\leqslant$ is the same for the third and fourth columns since its *mdt* is translated to the same Prolog code as the code produced by [6]. Thus, the execution times for symbol $\leqslant$ are similar. Note that the non-terminating symbol $\perp$ is defined by the rule $\perp = \perp$, and the mark ∞ represents an infinite evaluation sequence.

In future work, we plan to perform more exhaustive experiments.

6 Conclusions

We have provided an extension of the notion of definitional tree, called *matching definitional tree,* and a reformulation of natural rewriting and narrowing using these matching definitional trees. We have proved that both reformulations are correct w.r.t. to their original definitions. Moreover, we have shown that it is possible to implement natural rewriting and narrowing efficiently by compiling to Prolog. Hence, it seems possible to include the natural rewriting and narrowing strategies into current existing implementations without great effort. This could encourage programmers to write non-inductively sequential programs, whose sequential parts could still be executed in an optimal way.

Acknowledgements. I am grateful to María Alpuente, Salvador Lucas, and the anonymous referees for their helpful remarks.

References

1. M. Alpuente, S. Escobar, B. Gramlich, and S. Lucas. Improving on-demand strategy annotations. In M. Baaz and A. Voronkov, editors, *Proc. of the 9th Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'02),* Springer LNCS 2514, pages 1–18, 2002.
2. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of lazy functional logic programs. In *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97, ACM Sigplan Notices,* 32(12): 151–162, ACM Press, New York, 1997.
3. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conf. on Algebraic and Logic Programming ALP'92,* Springer LNCS 632, pages 143–157, 1992.
4. S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. of the Fourteenth Int'l Conf. on Logic Programming (ICLP'97),* pages 138–152. MIT Press, 1997.
5. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Journal of the ACM,* 47(4):776–822, 2000.
6. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into prolog. In *Proc. of the 3rd Int'l Workshop on Frontiers of Combining Systems (FroCoS 2000),* Springer LNCS 1794, pages 171–185, 2000.
7. S. Antoy and S. Lucas. Demandness in rewriting and narrowing. In M. Comini and M. Falaschi, editors, *Proc. of the 11th Int'l Workshop on Functional and (Constraint) Logic Programming WFLP'02,* Elsevier ENTCS 76, 2002.

8. T. Arts and J. Giesl. A collection of examples for termination of term rewriting using dependency pairs. Technical report, AIB-2001-09, RWTH Aachen, Germany, 2001.
9. S. Escobar. Refining weakly outermost-needed rewriting and narrowing. In D. Miller, editor, *Proc. of the 5th Int'l ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, PPDP'03,* pages 113–123. ACM Press, New York, 2003.
10. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming,* 19&20:583–628, 1994.
11. M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS 1.5.0: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany, 2003.
12. M. Hanus, S. Antoy, H. Kuchen, F. López-Fraguas, W. Lux, J. Moreno Navarro, and F. Steiner. Curry: An Integrated Functional Logic Language (version 0.8). Available at : `http://www.informatik.uni-kiel.de/~curry`, 2003.
13. G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems, Part I + II. In *Computational logic: Essays in honour of J. Alan Robinson,* pages 395–414 and 415–443. The MIT Press, Cambridge, MA, 1992.
14. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language Babel. *Journal of Logic Programming,* 12 (3): 191–224, 1992.
15. R. Sekar and I. Ramakrishnan. Programming in equational logic: Beyond strong sequentially. *Information and Computation,* 104(1):78–109, 1993.
16. TeReSe, editor. *Term Rewriting Systems.* Cambridge University Press, Cambridge, 2003.

Complete Axiomatization of an Algebraic Construction of Graphs

Mizuhito Ogawa

Japan Advanced Institute of Science and Technology
1-1 Asahidai Tatsunokuchi Nomi Ishikawa, 923-1292 Japan
mizuhito@jaist.ac.jp

Abstract. This paper presents a complete (infinite) axiomatization for an algebraic construction of graphs, in which a finite fragment denotes the class of graphs with bounded tree width.

1 Introduction

A graph is a flexible relational structure for describing problems. However, solving graph problems can be difficult, partially because graphs lack an obvious recursive construction.

The algebraic construction of graphs opens the possibility for graph algorithms that could be applied:

– efficient programming methodologies, such as depth-first search, divide-and conquer, and dynamic programming, which would enable us to design a new graph algorithm, and
– program transformation techniques, which are well-developed in the functional programming community [FS96, Erw97, SHTO00].

This is especially true for graphs with bounded tree width [RS86]. The class of graphs with bounded tree width is limited, but still contains interesting application areas; for instance, the control flow graphs of GOTO-free C programs have tree widths of at most 6 [Tho98], and those of practical Java programs mainly have at most 3 [GMT02].

A notable feature is that many NP-hard graph problems for general graphs are reduced to linear-time for graphs with bounded tree width [Cou90, BPT92]. This corresponds to the fact that algebraic constructions become finitely generated for a class of graphs with bounded tree width [BC87, ACPS93, OHS03], though they are infinitely generated for general graphs.

However, the algebraic structures referred above are not *initial,* i.e., the same graph could have several different expressions. Clarifying such equivalence could lead

– a debugging opportunity of programs, i.e., programs must have no conflicts with axioms, and
– efficient algorithm design for graph properties, such as graph isomorphism.

Our ultimate aim is to give a complete (finite) axiomatization for graphs with bounded tree width. This is half done; this paper presents the complete (infinite) axiomatization for an algebraic construction of general graphs, in which a finite fragment denotes graphs with bounded tree width. The idea of the proof for ground cases comes from [BC87]; our work further extends the completeness result to non-ground cases.

This paper is organized as follows. Section 2 prepares basic notations. Section 3 presents an algebraic construction of graphs with infinite signatures, which is a variation of those in [ACPS93]. Section 4 gives the complete (infinite) axioms for ground terms, and Section 5 extends them to non-ground terms. Section 6 is a brief overview of related work, and Section 7 discusses future work.

2 Preliminaries

Let F be a set of *function symbols* and X a countably infinite set of *variables*. Each function symbol f is supposed to have its arity $ar(f)$. A function symbol c such that $ar(c) = 0$ is called a *constant symbol*. The set of all *terms*, denoted by *T(F, X)*, built from F and X is defined as follows:

1. Constant symbols in F and variables in X are terms.
2. If $t_1, \ldots, t_n$ are terms, and f is a function symbol in F such that $ar(f) = n$, then $f(t_1, \ldots, t_n)$ is a term.

$\mathcal{V}(t)$ denotes the set of variables occurring in a term t. A term without variables is called a *ground* term, and a term in which each variable occurs at most once is called a *linear* term. The set of ground terms is denoted by *T(F)* for the set F of underlying function symbols.

Let $\Box$ be a fresh special constant symbol. A *context* $C[\ ]$ is a term built from $F \cup \Box$ and X. When $C[\ ]$ is a context with n $\Box$'s and $t_1, \cdots, t_n$ are terms, $C[t_1, \cdots, t_n]$ denotes the term obtained by replacing the i-th $\Box$ from the left in $C[\ ]$ with t_i for each $i = 1, \ldots, n$.

Definition 1. *A* term rewriting system *(TRS) is a set R of* rewrite rules. *A rewrite rule is a pair of terms denoted by $l \to r$ satisfying two conditions: (1) l is not a variable and (2) $\mathcal{V}(l) \supseteq \mathcal{V}(r)$.*

If $t = C[l\theta]$ and $s = C[r\theta]$ for $l \to r \in R$ and a substitution θ, $t \to_R s$ is a (one-step) reduction and $l\theta$ is called a redex.

A TRS R is terminating (or, strongly normalizing, SN *for short) if there are no infinite rewrite sequences $t_1 \to_R \cdots \to_R t_n \to_R \cdots$.*

Throughout the paper, we will use G, G' for (k-terminal) graphs, S for a set, X for a set of variables, s, t for terms, h, i, j, k, l for indices, and x, y for variables, s, t for terms, α, β for maps, θ for a substitution, and σ, τ for permutations, k is also often used for the number of terminals. l (resp. r) is sometimes used for the left-hand (resp. right-hand) side of a rewriting rule in a TRS.

3 Algebraic Construction of Graphs

In this paper we consider graphs with undirected edges, with at most one edge between any two vertices, and with no edge between a vertex and itself. (Extensions to multiple edges between vertices and to loops connecting a vertex to itself are easy, and sketched in Remark 2 of Section 4.) A k-terminal graph G is a graph with k distinguished vertices, called *terminals,* numbered 1 through k. The set of vertices of G is denoted $V(G)$, the set of edges of G is denoted by $E(G)$, and we write $G[i]$ for the i'th terminal of G, where $1 \leq i \leq k$. Ordinary graphs are obtained as 0-terminal graphs.

A k-terminal graph G is a pair of a graph and a tuple of its k distinct vertices, called *terminals.* The i-th terminal in a k-terminal graph G with $1 \leq i \leq k$ is denoted by $G[i]$ (like an array-like notation). Ordinary graphs are obtained as 0-terminal graphs after removal of terminals. For simplicity, we consider simple graphs (i.e., undirected and without multiple edged) without loops; but, the extensions to directed graphs, graphs with multiple edges, and/or graphs with loops are straightforward. The set of vertices of G is denoted by $V(G)$ and the set of edges of G is denoted by $E(G)$. The number of edges from a vertex v is denoted by $\#e(v)$.

Definition 2. *Let B_k be sorts for $k \geq 0$. Let $l_k^i, \oplus_k, r_k, \sigma_k^i, e^2, \mathbf{0}$ be function symbols with sorts below*

$$\begin{cases} e^2 \colon B_2, & l_k^i : B_{k-1} \to B_k, \quad \oplus_k \colon B_k \times B_k \to B_k, \\ \mathbf{0} : B_0, & r_k \colon B_k \to B_{k-1}, \quad \sigma_k^j : B_k \to B_k. \end{cases}$$

where $i \leq k$, $j < k$, and $k \geq 0$ (For readability, $\oplus_k$ is an infix operation and the rest are prefix). Let $\mathcal{B}_n$ be the set of well-sorted ground terms in

$$T(\{\mathbf{0}, e^2, l_k^i, r_k, \oplus_k, \sigma_k^j \mid 1 \leq i \leq k \leq n,\ 1 \leq j < k\})$$

and $\mathcal{B}_\infty = \cup_{n=0}^{\infty} \mathcal{B}_n$.

A term $t \in \mathcal{B}_k$ is interpreted as a k-terminal graph (defined below) by interpreting function symbols $l_k^i, \oplus_k, r_k, \sigma_k^i, e^2, \mathbf{0}$ as following operations. This interpretation is denoted by $\psi(t)$.

Definition 3. *Let $\psi(e^2)$ be the edge with two terminals and $\psi(\mathbf{0})$ be the empty graph. We define operations among k-terminal graphs as*

- *$\psi(l_k^i(t))$ is a lifting for $1 \leq i \leq k$, i.e., insert a new isolated terminal (as a new vertex) to $\psi(t)$ at the i-th position in $k-1$ terminals,*
- *$\psi(r_k(t))$ removes the last terminal from $\psi(t)$.*
- *$\psi(s \oplus_k t)$ is a parallel composition for $k \geq 0$, i.e., fuse each i-th terminal in $\psi(s)$ and $\psi(t)$ for $1 \leq i \leq k$.*
- *$\psi(\sigma_k^i(t))$ is a permutation, i.e., permute the i-th terminal and the $i+1$-th terminal in $\psi(t)$ for $1 \leq i < k$.*

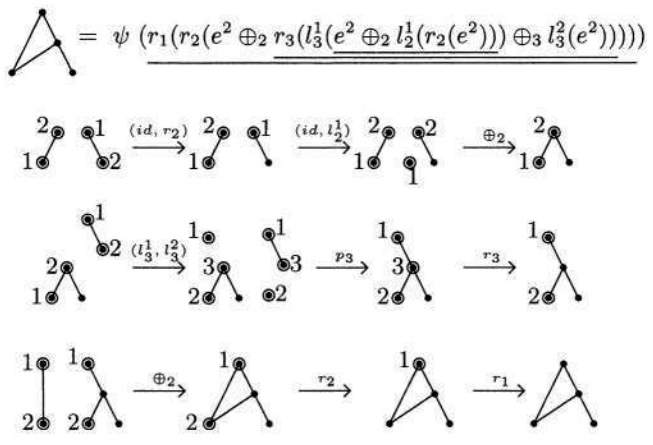


Fig. 1. An example of the algebraic construction

Example 1. Fig. 1 shows that the algebraic construction of a (0-terminal) graph. Each operation, underlined in $r_1(r_2(e^2 \oplus_2 r_3(l_3^1(e^2 \oplus_2 l_2^1(r_2(e^2))) \oplus_3 l_3^2(e^2))))$, is figured in lower columns.

Remark 1. Each permutation σ on $\{1, \cdots, k\}$ is generated from σ_k^i's. For instance, a circular permutation is generated as

$$\sigma_k^{j-1} \cdots \sigma_k^i = \begin{pmatrix} i & i+1 & \cdots & j \\ j & i & \cdots & j-1 \end{pmatrix}$$

for $1 \leq i < j \leq k$.

Although we do not show the definition of graphs with bounded tree width, the characterization of graphs with tree width at most k is given by the following theorem. This theorem is obtained similar to that in [ACPS93].

Theorem 1. *For $k \geq 0$, $\psi(\mathcal{B}_{k+1})$ is the set of graphs with tree width at most k (by neglecting terminals).*

4 Complete Axiomatization of Graphs: Ground Cases

A k-terminal graph could be denoted by different algebraic expressions; for instance, see Example 2.

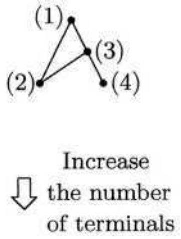
Example 2. Two terms below are equivalent and both denote the (0-terminal) graph in Fig. 1.

$$r_1(r_2(e^2 \oplus_2 r_3(l_3^1(e^2 \oplus_2 l_2^1(r_2(e^2))) \oplus_3 l_3^2(e^2)))))$$
$$r_1(r_2((e^2 \oplus_2 l_2^1(r_2(e^2))) \oplus_2 r_3(l_3^1(e^2) \oplus_3 l_3^2(e^2))))$$

In this section, we show that the (infinite) set of axioms $\mathcal{E}_\infty$ (in Fig. 3) is sound and complete for ground terms (Theorem 2 and 3). The key of the proof is the existence of a *canonical form* that denotes a graph in which all vertices are terminals (see Example 3). Then, canonical forms denoting an isomorphic graph are converted each other by the associativity and commutativity rules of the parallel composition $\oplus_k$'s (AC1 and AC2 in Fig. 3) and suitable permutations σ_k^i's among terminals.

Example 3. Fig. 3 shows a transformation to obtain a canonical form of the expression in Example 1, where R_1 will be defined in Definition 6. The underlined parts correspond to the rewrite steps. (The infix operation $\oplus_4$ has the commutative associative axioms, and we omit parenthesis in the last line for readability.)

$$
\begin{aligned}
& r_1(r_2(e^2 \oplus_2 r_3(l_3^1(e^2 \oplus_2 \underline{l_2^1(r_2}(e^2))) \oplus_3 l_3^2(e^2)))) \\
\to_{R_1}\ & r_1(r_2(e^2 \oplus_2 r_3(l_3^1(\underline{e^2 \oplus_2 r_3(l_3^1(e^2))}), \oplus_3 l_3^2(e^2)))) \\
\to_{R_1}\ & r_1(r_2(e^2 \oplus_2 r_3(\underline{l_3^1(r_3}(l_3^3(e^2) \oplus_3 l_3^1(e^2))) \oplus_2 l_3^2(e^2)))) \\
\to_{R_1}\ & r_1(r_2(e^2 \oplus_2 r_3(r_4(\underline{l_4^1(l_3^3(e^2) \oplus_3 l_3^1(e^2))}) \oplus_3 l_3^2(e^2)))) \\
\to_{R_1}\ & r_1(r_2(e^2 \oplus_2 r_3(\underline{r_4(l_4^1(l_3^3(e^2)) \oplus_4 l_4^1(l_3^1(e^2))) \oplus_3 l_3^2(e^2)})))) \\
\to_{R_1}\ & r_1(r_2(\underline{e^2 \oplus_2 r_3(r_4(l_4^1(l_3^3(e^2)) \oplus_4 l_4^1(l_3^1(e^2)) \oplus_4 l_4^4(l_3^2(e^2)))}))) \\
\to_{R_1}^{+}\ & \underbrace{r_1(r_2(r_3(r_4(}_{R[\ ]}\underbrace{l_4^4(l_3^3}_{L_1[\ ]}\ \underbrace{(e^2)}\,)\underbrace{\oplus_4}\underbrace{l_4^4(l_3^1}_{L_2[\ ]}\ \underbrace{(e^2)}\,)\underbrace{\oplus_4}_{P[\ ]}\underbrace{l_4^2(l_3^1}_{L_3[\ ]}\ \underbrace{(e^2)}\,)\underbrace{\oplus_4}\underbrace{l_4^4(l_3^2}_{L_4[\ ]}\ \underbrace{(e^2)})))))
\end{aligned}
$$



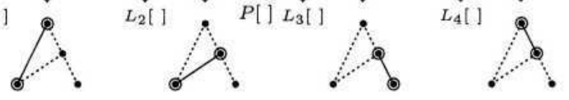


Fig. 2. Example of transformation to a canonical form (ground case)

Definition 4. *k-terminal graphs G_1, G_2 are isomorphic if there exists a one-to-one onto map $\alpha : V(G_1) \to V(G_2)$ such that*

- *For $v \in V(G_1)$, if v is the i-th terminal of G_1 with $1 \le i \le k$, then $\alpha(v)$ is the i-th terminal of G_2, and vice versa.*
- *For $v, v' \in V(G_1)$, if (v, v') is an edge of G_1, then $(\alpha(v), \alpha(v'))$ is an edge of G_2, and vice versa.*

Definition 5. *Two terms s, t of sort B_k are* equivalent *if the k-terminal graphs $\psi(s), \psi(t)$ are isomorphic.*

$\mathcal{E}_k$ in Fig. 3 is the set of axioms indexed by k. Let $\mathcal{E}_\infty = \cup_{k=1}^{\infty} \mathcal{E}_k$ and $\mathcal{E}_{\le n} = \cup_{k=1}^{n} \mathcal{E}_k$. By regarding each equation (axiom) as a left-to-right rewrite

$$
\begin{array}{rcllr}
t_1 \oplus_k t_2 & = & t_2 \oplus_k t_1 & (\mathit{Commut.}) & \text{(AC1)} \\
(t_1 \oplus_k t_2) \oplus_k t_3 & = & t_1 \oplus_k (t_2 \oplus_k t_3) & (\mathit{Assoc.}) & \text{(AC2)} \\
l_k^j(l_{k-1}^i(t)) & = & l_k^i(l_{k-1}^{j-1}(t)) & 1 \le i < j \le k & (l\text{-Com}) \\
l_k^i(t_1 \oplus_{k-1} t_2) & = & l_k^i(t_1) \oplus_k l_k^i(t_2) & 1 \le i \le k & (l\text{-Dist}) \\
l_{k-1}^i(r_{k-1}(t)) & = & r_k(l_k^i(t)) & 1 \le i < k & \text{(E1)} \\
t_1 \oplus_{k-1} r_k(t_2) & = & r_k(l_k^k(t_1) \oplus_k t_2) & & \text{(E2)} \\
t \oplus_k l_k^k(\cdots l_1^1(\mathbf{0}))) & = & t & & \text{(E3)} \\
e^2 \oplus_2 e^2 & = & e^2 & & \text{(E4)} \\
\sigma_k^j(l_k^i(t)) & = & l_k^i(\sigma_{k-1}^{j-1}(t)) & 1 \le i < j < k & (\sigma1\text{-a}) \\
\sigma_k^i(l_k^i(t)) & = & l_k^{i+1}(t) & 1 \le i < k & (\sigma1\text{-b}) \\
\sigma_k^i(l_k^{i+1}(t)) & = & l_k^i(t) & 1 \le i < k & (\sigma1\text{-c}) \\
\sigma_k^j(l_k^i(t)) & = & l_k^i(\sigma_{k-1}^j(t)) & 1 < j+1 < i \le k & (\sigma1\text{-d}) \\
\sigma_2^1(e^2) & = & e^2 & & (\sigma2) \\
\sigma_k^i(t_1 \oplus_k t_2) & = & \sigma_k^i(t_1) \oplus_k \sigma_k^i(t_2) & 1 \le i < k & (\sigma3) \\
\sigma_{k-1}^i(r_k(t)) & = & r_k(\sigma_k^i(t)) & 1 \le i < k-1 & (\sigma4) \\
r_{k-1}(r_k(\sigma_k^{k-1}(t))) & = & r_{k-1}(r_k(t)) & & (\sigma5)
\end{array}
$$

Fig. 3. Axioms $\mathcal{E}_k$ of the algebraic construction of graphs

rule), its reflexive symmetric transitive closure (i.e., the finite application of axioms in $\mathcal{E}_\infty$) is denoted by $=_{\mathcal{E}_\infty}$.

It is easy to see that each axiom in $\mathcal{E}_\infty$ is sound.

Theorem 2. (Soundness for ground terms) *Let s, t be ground terms in $\mathcal{B}_\infty$. Then, s and t are equivalent if $s =_{\mathcal{E}_\infty} t$.*

Theorem 3. (Completeness for ground terms) *Let s, t be ground terms in $\mathcal{B}_\infty$. Then, $s =_{\mathcal{E}_\infty} t$ if s and t are equivalent.*

Definition 6. *For axioms in $\mathcal{E}_\infty$, let TRSs R_1 and R_2 be defined as*

$$\begin{cases} R_1 = \{(E1), (E2), (E2)', (l\text{-}Dist), (\sigma3), (\sigma4)\}, \\ R_2 = \{(\sigma1), (\sigma2), \end{cases}$$

where $(E2)'$ is $r_k(t_1) \oplus_{k-1} t_2 \to r_k(t_1 \oplus_k l_k^k(t_2))$ for each k.

Lemma 1. *R_1 and R_2 are terminating.*

Proof. Let $\delta(t,f)$ be the number of occurrences of a function symbol f in a term t, and let $\Delta(t,g,f)$ be the sum of all $\delta(s,f)$ where s is a subterm of t such that $root(s)=g$. We define the weight $\omega(t)$ of a term t by

$$\omega(t)=(\omega_{\oplus,r}(t),\ \omega_{l,r}(t)+\omega_{l,\oplus}(t)+\omega_{\sigma,r}(t)+\omega_{\sigma,\oplus}(t))$$

where

$$\begin{aligned}\omega_{\oplus,r}(t)&=\Sigma_{j,k}\Delta(t,\oplus_k,r_j),\\ \omega_{l,r}(t)&=\Sigma_{i,j,i',j'}\Delta(t,l^i_j,r_{j'}),\\ \omega_{l,\oplus}(t)&=\Sigma_{i,j,k}\Delta(t,l^i_j,\oplus_k),\\ \omega_{\sigma,r}(t)&=\Sigma_{i,j,i',j'}\Delta(t,\sigma^i_j,r_{j'}),\\ \omega_{\sigma,\oplus}(t)&=\Sigma_{i,j,k}\Delta(t,\sigma^i_j,\oplus_k),\end{aligned}$$

and define the lexicographic order on the weight. Then, for each reduction of R_1 the weight $\omega(t)$ decreases, and R_1 is **SN.** Similarly, each reduction of R_2 decreases the weight $\omega_{\sigma,l}(t)=\Sigma_{i,j,i',j'}\Delta(t,\sigma^i_j,l^{i'}_{j'})$, and R_2 is **SN.** ∎

Definition 7. *Let $t\in\mathcal{B}_\infty$ be a ground term of sort B_k, $n=|V(\psi(t))|$, and $m=|E(\psi(t))|$. t is a canonical form if either*

$$t=r_{k+1}(\cdots r_n(l^n_n(\cdots l^1_1(\mathbf{0})))),$$

or there exist

- $R_{n,k}[\ ]=r_{k+1}(\cdots r_n[\ ])$ *with* $0\le k<n$,
- $P_n[\ ,\cdots,\ ]$ *consists of* $\oplus_n$*'s,*
- $L_i[\ ]$ *has the form* $l_n^{u_{i,n-2}}(\cdots l_3^{u_{i,1}}[\ ])$ *with* $u_{i,n-2}>\cdots>u_{i,1}$ *for* $1\le i\le m$,

such that $t=R_{n,k}[P_n[L_1[e^2],\cdots,L_m[e^2]]]$.

Lemma 2. *For any term s, there exists a canonical form $t\in\mathcal{B}_n$ such that $s=_{\mathcal{E}_{\le n}}t$ where $n=|V(\psi(t))|$.*

Proof. We first show that there exists t' in the form $t'=R_{n,k}[P'[L'_1[c_1],\cdots,L'_l[c_l]]]$ with $s=_{\mathcal{E}_{\le n}}t'$ where

- $R_{n,k}[\ ]=r_{k+1}\cdots r_n[\ ]$,
- $P'[\ ]$ consists of $\oplus_j$'s, and
- $L'_1[\ ],\cdots,L'_l[\ ]$ consist of l^i_j's and $\sigma^{i'}_{k'}$'s.
- c_i is either e^2 or $\mathbf{0}$,

From Lemma 1, s has an R_1-**normal** form t' of the form $R_{n,k}[P'[L'_1[c_1],\cdots,L'_l[c_l]]]$. Since all vertices in e^2 are terminals and l^i_j,σ^i_j preserves a set of terminals, all vertices of each $L'_i[e^2]$ are terminals, r_i and $\oplus_j$ do not change the number of vertices, thus each $\oplus_j$ in $P'[\ ]$ satisfies $j=n=|V(\psi(t))|$. Further, from Lemma 1 each $L'_i[c_i]$ has an R_2-**normal** form, i.e., a σ^j_k-**free** term.

If $|E(\psi(s))|=0$, this means $\psi(s)$ consists of isolated vertices and all c_i's are $\mathbf{0}$. Thus, $L'_i[\ ]=l^k_k(\cdots(l^1_1[\ ]))$ by (*l-Com*) and s is reduced to a canonical form $R_{n,k}[L_1[\mathbf{0}]]$ by (AC1), (AC2), and (E3).

If $|E(\psi(s))| > 0$, we can sort each $L'_i[\,]$ by (l-Com). Since there exists $c_i = e^2$, we can erase **0's** by (AC1), (AC2), and (E3). Thus we assume $c_i = e^2$ for each i. If $L'_i[c_i]$ and $L'_j[c_j]$ are equal, we can eliminate redundant $L'_i[c_i]$'s by (AC1), (AC2), and (E4). Since each $L'_i[c_i]$ corresponds to an edge in $\psi(s)$ (i.e., the number of $L'_i[c_i]$'s is the number of edges in $\psi(s)$), we obtain a canonical form $t = R_{n,k}[P_n[L_1[e^2],\cdots,L_m[e^2]]]$ by (l-Com) (from-right-to-left direction). ■

Definition 8. *Let* $e(n,i,j) = l_n^n \cdots l_{j+1}^{j+1} \cdot (l_j^{j-1} \cdots l_{i+2}^{i+1} \cdot l_{i+1}^{i-1} \cdots l_3^1(e^2)$ *for* $1 \leq i < j \leq n$ *(here we omit apparent parenthesis for readability).*

Lemma 3. *Let* $s \in \mathcal{B}_\infty$. $\psi(s)$ *contains an edge between the* i*-th and the* j*-th vertices, if, and only if, a canonical form of* s *contains* $e(n,i,j)$.

Sketch of proof of Theorem 3 Let $s,t \in \mathcal{B}_\infty$ such that $\psi(s)$ and $\psi(t)$ are equivalent. Assume that an isomorphism $\alpha : V(\psi(s)) \to V(\psi(t))$ satisfies the conditions in Definition 4. If $|E(\psi(s))| = |E(\psi(t))| = 0$, they have the unique canonical form from Lemma 2 and obviously the theorem holds. We assume $|E(\psi(s))| = |E(\psi(t))| > 0$.

From Lemma 2, we can assume that both s and t are canonical. Let $s = R_{n,k}[P_n[L_1[e^2],\cdots,L_m[e^2]]]$ and $t = R_{n,k}[P'_n[L'_1[e^2],\cdots,L'_m[e^2]]]$ where $n = |V(\psi(s))| = |V(\psi(t))|$ and $m = |E(\psi(s))| = |E(\psi(t))|$. Thus, α can be regarded as the permutation σ on $\{k+1,\cdots,n\}$.

Non-trivial permutation needs at least two elements, so we can assume $k \leq n-2$. Then from (σ4) and (σ5), $r_{k+1}^{k+1}(\cdots r_n^n(\sigma_n^i(t)) = r_{k+1}^{k+1}(\cdots r_n^n(t))$ for $k+1 \leq i \leq n-1$. Since a permutation over $\{k+1,\cdots,n\}$ is generated by σ_n^i's for $k+1 \leq i \leq n-1$, $r_{k+1}^{k+1}(\cdots r_n^n(\sigma(t)) = r_{k+1}^{k+1}(\cdots r_n^n(t))$. Thus, it is enough to show

$$\sigma(P_n[L_1[e^2],\cdots,L_m[e^2]]) =_{\mathcal{E}_{\leq n}} P'_n[L'_1[e^2],\cdots,L'_m[e^2]].$$

Since $\psi(s)$ and $\psi(t)$ are isomorphic, if there is an edge between the i-th and j-th vertices of $\psi(s)$, there is an edge between the $\alpha(i)$-th and $\alpha(j)$-th vertices of $\psi(t)$, and vice versa. Thus, if there is an edge between the i-th and j-th vertices in $\psi(s)$, then, form Lemma 3, there uniquely exist $L_k[e^2]$ and $L'_k[e^2]$ such that $L_k[e^2] =_{\mathcal{E}_{\leq n}} e(n,i,j)$ and $L'_k[e^2] =_{\mathcal{E}_{\leq n}} e(n,\alpha(i),\alpha(j))$.

Since $\sigma(e(n,i,j)) = e(n,\alpha(i),\alpha(j))$,

$$\sigma(P_n[L_1[e^2],\cdots,L_m[e^2]]) =_{\mathcal{E}_{\leq n}} P'_n[L'_1[e^2],\cdots,L'_m[e^2]]$$

holds from (AC1), (AC2), (σ2), and (σ3). ■

Remark 2. The extensions to directed graphs, graphs with multiple edges, and/or graphs with loops are as follows:

– The removal of (E4) in Fig. 3 gives the sound and complete axioms for graphs with multiple edges.

- By adding a constant l^1 as a 1-terminal graph that consists of the unique terminal and the unique edge from the terminal to the terminal itself, we obtain the algebraic construction of graphs with loops. The axioms are preserved for this extension.
- For digraphs, instead of an edge e^2, we use e^2_+ and e^2_-, where e^2_+ is the directed edge from the first terminal to the second, and e^2_- is opposite. Then, the replacement of $\sigma^1_2(e^2) = e^2$ ($\sigma 2$) with $\sigma^1_2(e^2_+) = e^2_-$ and $\sigma^1_2(e^2_-) = e^2_+$ lead the sound and complete axioms for directed graphs.

5 Complete Axiomatization of Graphs: Non-ground Cases

In this section, we extend the result of soundness (Theorem 2) and completeness (Theorem 3) for ground terms to general terms. In this extension, we need additional axioms ($\Sigma 1$) and ($\Sigma 2$) in Fig. 4, which present the *defining relation* of the permutation group [Wey39].

Lemma 4. *[Wey39] For any permutation σ and σ' that are expressed as products of σ^i_k's with $1 \leq i < k$, they are equivalent as a map if and only if $\sigma =_{\mathcal{G}_k} \sigma'$, where $\mathcal{G}_k$ consists of ($\Sigma 1$) and ($\Sigma 2$) axioms in Fig. 4.*

$$\begin{array}{llll} \sigma^i_k \cdot \sigma^i_k(G) & = G & 1 \leq i < k & (\Sigma 1) \\ (\sigma^{i-1}_k \cdot \sigma^i_k)^3(G) & = G & 1 < i < k & (\Sigma 2) \end{array}$$

Fig. 4. Additional axioms $\mathcal{G}_k$ of the algebraic construction of graphs

Example 4. Consider the permutation of 1 and 3 among {1,2,3}

$$\begin{pmatrix} 1\,2\,3 \\ 3\,2\,1 \end{pmatrix}$$

which is represented as $\sigma^2_3 \cdot \sigma^1_3 \cdot \sigma^2_3$ or $\sigma^1_3 \cdot \sigma^2_3 \cdot \sigma^1_3$. This equivalence is obtained by $=_{\mathcal{G}_k}$ as

$$\begin{array}{rl} \sigma^2_3 \cdot \sigma^1_3 \cdot \sigma^2_3 =_{\Sigma 2} & \sigma^2_3 \cdot \sigma^1_3 \cdot (\sigma^1_3 \cdot \sigma^2_3)^3 \cdot \sigma^2_3 \\ = & \sigma^2_3 \cdot (\sigma^1_3 \cdot \sigma^1_3) \cdot \sigma^2_3 \cdot \sigma^1_3 \cdot \sigma^2_3 \cdot \sigma^1_3 \cdot (\sigma^2_3 \cdot \sigma^2_3) \\ =_{\Sigma 1} & (\sigma^2_3 \cdot \sigma^2_3) \cdot \sigma^1_3 \cdot \sigma^2_3 \cdot \sigma^1_3 \\ =_{\Sigma 1} & \sigma^1_3 \cdot \sigma^2_3 \cdot \sigma^1_3 \end{array}$$

Remark 3. For ground terms, ($\Sigma 1$) and ($\Sigma 2$) in Fig. 4 are not required, because the same can be performed by (σ1-d) and ($\sigma 2$) in Fig. 3.

Let X_k be a set of variables with sort B_k. The i-th terminal of x is denoted by $x[i]$. Let $X = \cup_k X_k$. The set of well-sorted terms in

$$T(\{\mathbf{0}, e^2, l_k^i, r_k, \oplus_k, \sigma_k^j \mid 1 \leq i \leq k \leq n,\ 1 \leq j < k\},\ X)$$

is denoted by $\mathcal{B}_\infty(X)$. Define a substitution θ_0 by $x\theta_0 = l_k^k \cdots l_1^1(\mathbf{0})$ for each variable $x \in X_k$.

Definition 9. *For $s, t \in \mathcal{B}_\infty(X)$, s and t are equivalent if, for each ground substitution θ, $\psi(s\theta)$ and $\psi(t\theta)$ are isomorphic.*

The next theorem is immediate.

Theorem 4. (Soundness) *Let s, t be terms in $\mathcal{B}_\infty(X)$. Then s and t are equivalent if $s =_{\mathcal{E}_\infty \cup \mathcal{G}_\infty} t$.*

Difficult part is completeness.

Theorem 5. (Completeness) *Let s, t be terms in $\mathcal{B}_\infty(X)$. Then $s =_{\mathcal{E}_\infty \cup \mathcal{G}_\infty} t$ if s and t are equivalent.*

Similar to the ground case, we first consider a canonical form of a term t. The set of variables that appear in a term t in $\mathcal{B}_\infty(X)$ is denoted by $\mathcal{V}(t)$.

Definition 10. *Let t $(\in \mathcal{B}_\infty(X))$ be a term of sort B_k, $n = |V(\psi(t\theta_0))|$, $m = |E(\psi(t\theta_0))|$, and $\mathcal{V}(t) = \{x_1, \cdots, x_{m'}\}$. t is a canonical form if either*

$$t = r_{k+1}(\cdots r_n(l_n^n(\cdots l_1^1(\mathbf{0})))),$$

or there exist

- $R_{n,k}[\ ] = r_{k+1}(\cdots r_n[\ ])$,
- $P_n[\ , \cdots,\ ]$ *consists of* $\oplus_n$*'s,*
- $L_i[\ ]$ *has the form* $l_n^{u_{i,n-2}}(\cdots l_3^{u_{i,1}}[\ ])$ *with* $u_{i,n-2} > \cdots > u_{i,1}$ *for* $1 \leq i \leq m$,
- $L_{m+i}[\ ]$ *has the form* $l_n^{u'_{i,n-d_i}}(\cdots l_{d_i+1}^{u'_{i,1}}[\ ])$ *with* $u'_{i,n-d_i} > \cdots > u'_{i,1}$ *for* $x_i \in X_{d_i}$ *and* $1 \leq i \leq m'$,
- G_i *is* $\sigma_i(x_i)$ *for some combination* σ_i *of* $\sigma_{d_i}^j$*'s for* $1 \leq i \leq m'$,

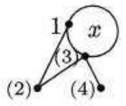
such that

$$t = R_{n,k}[P_n[L_1[e^2],\ \cdots,\ L_m[e^2],\ L_{m+1}[G_1],\ \cdots,\ L_{m+m'}[G_{m'}]]].$$

Define $Center(t) = \psi(R_{n,k}[P_n[L_1[e^2], \cdots, L_m[e^2]]])$. For a ground substitution θ, let $Inner(t, \theta) = V(Center(t))$ and $Outer(t, \theta) = V(\psi(t\theta)) \setminus Inner(t, \theta)$. We say a vertex is inner *if it is in $Inner(t, \theta)$, and* outer *otherwise.*

Lemma 5. *$Center(t)$ is isomorphic to $\psi(t\theta_0)$.*

$$\begin{aligned}
& r_2(e^2 \oplus_2 r_3(l_3^1(e^2 \oplus_2 \underline{l_2^1(r_2}(e^2))) \oplus_3 \sigma_3^2 \cdot \sigma_3^1 \cdot \underline{\sigma_3^2(l_3^2}(x)))) \\
\to^+ & r_2(e^2 \oplus_2 r_3(l_3^1(\underline{e^2 \oplus_2 r_3(l_3^1(e^2))}) \oplus_3 \sigma_3^2 \cdot \underline{\sigma_3^1(l_3^3}(x)))) \\
\to^+ & r_2(e^2 \oplus_2 r_3(\underline{l_3^1(r_3}(l_3^3(e^2) \oplus_3 l_3^1(e^2))) \oplus_3 \underline{\sigma_3^2(l_3^3}(\sigma_2^1(x))))) \\
\to^+ & r_2(e^2 \oplus_2 r_3(r_4(\underline{l_4^1(l_3^3(e^2) \oplus_3 l_3^1(e^2))}) \oplus_3 l_3^2(\sigma_2^1(x)))) \\
\to\ & r_2(e^2 \oplus_2 r_3(\underline{r_4(l_4^1(l_3^3(e^2)) \oplus_4 l_4^1(l_3^1(e^2))) \oplus_3 l_2^3(\sigma_2^1(x))})) \\
\to\ & r_2(\underline{e^2 \oplus_2 r_3(r_4(l_4^1(l_3^3(e^2)) \oplus_4 l_4^1(l_3^1(e^2)) \oplus_4 l_4^4(l_3^2(\sigma_2^1(x))))})) \\
\to^+ & \underbrace{r_2(r_3(r_4}_{R[\ ]}(\underbrace{l_4^4(l_3^3}_{L_1[\ ]}\ \underbrace{(e^2))}\oplus_4 \underbrace{l_4^4(l_3^1}_{L_2[\ ]}\ \underbrace{(e^2))}_{P[\ ]}\oplus_4 \underbrace{l_4^2(l_3^1}_{L_3[\ ]}\ \underbrace{(e^2))}\oplus_4 \underbrace{l_4^4(l_3^2}_{L_4[\ ]}\underbrace{(\sigma_2^1(x))}))))
\end{aligned}$$



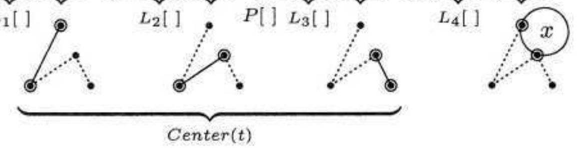


Fig. 5. Example of transformation to a canonical form (non-ground case)

Example 5. Fig. 5 shows the conversion of

$$t = r_2 \cdot p_2(e^2, r_3 \cdot p_3(l_3^1 \cdot p_2(e^2, l_2^1 \cdot r_2(e^2)), \sigma_3^2 \cdot \sigma_3^1 \cdot \sigma_3^2 \cdot l_3^2(x)))$$

to a canonical form. The circle expresses a substitution to a variable x, and the parenthesis for σ_k^i and the commutative associative operator $\oplus_4$ are omitted.

The next lemma is similarly proved as the proof of Lemma 2.

Lemma 6. *For any term $s \in \mathcal{B}_\infty(X)$, there exists a canonical form $t \in \mathcal{B}_n$ such that $s =_{\mathcal{E}_{\leq n}} t$ where $n = |V(Center(t))|$.*

When terms s and t are equivalent, without loss of generality, we can assume that s and t are canonical forms. Let us fix canonical forms s and t.

Lemma 7. *If s and t are equivalent, $\mathcal{V}(s) = \mathcal{V}(t)$.*

Proof. Assume $\mathcal{V}(s) \neq \mathcal{V}(t)$. Without loss of generality, we can assume that $x \in \mathcal{V}(s)$ and $x \notin \mathcal{V}(t)$. From Lemma 5, $Center(s)$ and $Center(t)$ are isomorphic. Let $n = |V(Center(s))| = |V(Center(t))|$.

Consider a ground substitution θ that substitutes a term denoting K_{n+1} (complete graph with $n+1$ vertices) to x, and $l_k^k \cdots l_1^1(\mathbf{0})$ otherwise (for those that in X_k). Then, $|V(\psi(s\theta))| > |V(\psi(t\theta))| = n$, and the contradiction. ■

Lemma 8. *If s and t are equivalent, each variable x occurs the same times both in s and t.*

Proof. Assume that x occurs in s more than in t. Similar to Lemma 7, consider a ground substitution θ that substitutes a term denoting K_{n+1} (complete graph with $n+1$ vertices) to x, and $l_k^k \cdots l_1^1(\mathbf{0})$ otherwise (for those that in X_k). Then, $|V(\psi(s\theta))| > |V(\psi(t\theta))|$, and the contradiction. ■

For notational clarity, we consider conditional linearization of a term.

Definition 11. *Conditional linearization of a term t is obtained by renaming different occurrences of the same variable x to distinct variables $x', x'', \cdots$, associated with the side condition $C = \{x' = x'' = \cdots\}$.*

Example 6. Conditional linearization of a term $p_3(l_3^1(p_2(x,y)), l_3^2(x))$ is

$$p_3(l_3^1(p_2(x',y)), l_3^2(x'')) \text{ with } \{x' = x''\}.$$

From now on, we consider conditional linearization of canonical forms s and t. Let us fix $\mathcal{V}(s)(= \mathcal{V}(t))$ as $\{x_1, \cdots, x_m\}$ with the side condition $C : \{x_i = x_j\}$. Note that from Lemma 7 and 8, such C is well-defined.

Next we define $x_i[t,j]$, which is the vertex in $Center(t)$ that corresponds to the j-th terminal in $\psi(x\theta)$ for each ground substitution θ.

Definition 12. *We borrow the notation from Definition 10. Let t be a canonical form $t = R_{n,k}[P_n[L_1[e^2], \cdots, L_m[e^2], L_{m+1}[G_1], \cdots, L_{m+m'}[G_{m'}]]]$ and let $(v_1, v_2, \cdots, v_n)$ be the tuple of terminals of*

$$\psi(P_n[L_1[e^2], \cdots, L_m[e^2], L_{m+1}[G_1], \cdots, L_{m+m'}[G_{m'}]]\ \theta_0).$$

Assume that a variable x_i in t is of the sort B_{d_i} and let

$$L_{m+i}[G_i] = l_n^{u_{n-d}}(\cdots(l_{d_i+1}^{u_1}[\sigma_i(x_i)]))$$

with $u_{n-d_i} > \cdots > u_1$. Define $x_i[t,j] = v_{\sigma_i^{-1}(w_j)}$ where

$$\{w_1, \cdots, w_{d_i}\} = \{1, \cdots, n\} \setminus \{u_1, \cdots, u_{n-d_i}\}$$

with $w_1 < \cdots < w_{d_i}$.

Example 7. In Example 5, $x[t,1] = v_3$ and $x[t,2] = v_1$.

Below, we define a *marker substitution* $\theta_{\mathcal{M}}$, which distinguishes each terminal $x_i[t,j]$ by the pair of its outer neighborhoods; these neighborhoods are distinguished each other by the number of edges in $\psi(t\ \theta_{\mathcal{M}})$.

Since the number of edges and the neighborhood relation are preserved by an isomorphism, an isomorphism between $\psi(st\ \theta_{\mathcal{M}})$ and $\psi(t\ \theta_{\mathcal{M}})$ induces the isomorphism between $Center(s)$ and $Center(t)$ that maps $x_i[s,j]$ to $x_{i'}[t,j]$ with $x_i = x_{i'} \in C$.

Definition 13. *Let $term_1, \cdots, term_d$ be vertices, and let $ch_0, \cdots, ch_d$ be their children. A rooted tree with the root vertex v and its m children is denoted by $br(v,m)$. For $d \leq h$, a* marker forest *$MF(h,d)$ is a d-terminal graph such that*

$$V(MF(h,d))) = \begin{cases} \phi & \text{if } d = 0 \\ V(br(ch_0, h-d)) \cup (\bigcup_{1 \leq i \leq d} V(br(ch_i, h+2i-2)) \\ \qquad \cup \{term_i\}) & \text{otherwise} \end{cases}$$

and

$$E(MF(h,d)) = \begin{cases} \phi & \text{if } d=0 \\ E(br(ch_0, h-d)) \cup (\bigcup_{1\leq i\leq d} E(br(ch_i, h+2i-2))) & \\ \qquad \cup \{(term_i, ch_{i-1}), (term_i, ch_i), (ch_0, ch_i)\} & \text{otherwise} \end{cases}$$

A marker term $Mt(h,d)$ is a term that denote $MF(h,d)$.

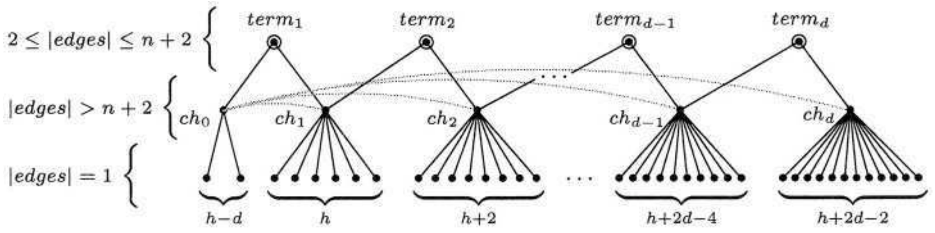


Fig. 6. d-terminal graph $MF(h,d)$

Lemma 9. *In $MF(h,d)$, $h+1 \leq \#e(ch_i) \leq h+2d+1$ for each $0 \leq i \leq d$ and $\#e(ch_i) < \#e(ch_j)$ if $i<j$. Moreprecisely, $\#e(ch_i) = h+2i+1$ for $0 \leq i < d$ and $\#e(ch_d) = h+2d$.*

Definition 14. *Without loss of generality, we can assume that $x_1, \cdots, x_l$ are the representatives under the side condition C of t (i.e., $x_1, \cdots, x_l$ are mutually distinct and for each $x \in \mathcal{V}(t)$ there exists some x_i such that C contains $x = x_i$ with $1 \leq i \leq l$). Let $x_i \in X_{d_i}$.*

Let $n = |V(\psi(t\ \theta_0))|$. The marker substitution $\theta_{\mathcal{M}}$ (see Fig. 6) is a ground substitution such that

$$\begin{cases} x_1\ \theta_{\mathcal{M}} = Mt(n+2,\ d_1) \\ x_{i+1}\ \theta_{\mathcal{M}} = Mt(n+2+\Sigma_{j=1}^{i} d_j,\ d_{i+1}) & \text{for } 1 \leq i < l. \end{cases}$$

Example 8. In Example 5, $x\theta_{\mathcal{M}} = Mt(6,2)$ (see Fig 7).

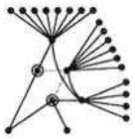


Fig.7. Substitute $MF(6, 2)$ to x in Example 5

Lemma 10. *Let $v \in \psi(t\theta_{\mathcal{M}})$ and $n = |V(Center(t))|$. If v is inner, $2 \leq \#e(v) \leq n+2$. If v is outer, either $\#e(v) = 1$ or $\#e(v) > n+2$.*

Lemma 11. *If s and t are equivalent, an isomorphism α between $\psi(s\ \theta_{\mathcal{M}}))$ and $\psi(t\ \theta_{\mathcal{M}}))$ satisfies :*

- *α is an isomorphism between $Center(s)$ and $Center(t)$.*
- *For each x_i, there exists $x_{i'}$ with $x_i = x_{i'} \in C$, $\alpha(\psi(x_i\theta_{\mathcal{M}})) = \psi(x_{i'}\theta_{\mathcal{M}})$, and $\alpha(x_i[s,j]) = x_{i'}[t,j]$.*

Proof. From Lemma 10, $\alpha(V(Center(s)) = V(Center(t))$.

Let $n = |V(Center(s))|$. For ch_0 in $\psi(x_i\theta_{\mathcal{M}})$, there exists $x_{i'}$ and with $x_i = x_{i'} \in C$ and $\psi(x_{i'}\theta_{\mathcal{M}})$ such that $\alpha(ch_0) = ch'_0$for ch'_0 in $\psi(x_{i'}\theta_{\mathcal{M}})$ by construction. Since the unique neighborhood of ch_0 satisfying $2 \leq \#e(ch_0) \leq n+2$ is $term_1$, $\alpha(term_1) = term'_1$ with $term'_1$ in $\psi(x_{i'}\theta_{\mathcal{M}})$. Since ch_1 is the unique neighborhood of ch_0 that has more then $n+2$ edges, $\alpha(ch_1)$ must be ch'_1. Repeating similar construction, Lemma is proved. ∎

Sketch of proof of Theorem 5 By using the isomorphism α in Lemma 11, similar to the proof of Theorem 3, we obtain the proof of Theorem 5. ∎

6 Related Work

There are many works on algebraic constructions of graphs, including

- [FS96, Erw97] for functional programming,
- [CS92, Has97] from the categorical view point,
- [MSvE94, AA95] for term graphs,
- [Gib95] for directed acyclic graphs, and
- [BC87, ACPS93, OHS03] for graphs with bounded tree width.

Among them, only [BC87, ACPS93, OHS03] characterize the class of graphs with bounded tree width. Bauderon and Courcelle presented the complete axiomatization for ground terms [BC87, Cou90] in their formalization. Their algebraic construction consists of the function symbols

$$\left\{\begin{array}{lll} \oplus_{m,n} & : B_m \times B_n \to B_{m+n}, & e^2 : B_2 \ \text{(edge)}, \\ \theta_{i,j,n} & : B_n \to B_n, & \mathbf{1} : B_1 \ \text{(vertex)}, \\ \sigma_\alpha & : B_m \to B_n, & \mathbf{0} : B_0 \ \text{(empty)}, \end{array}\right.$$

where their interpretation ψ is

- $\psi(\oplus_{m,n}(t_1,t_2))$ is a disjoint union of $\psi(t_1)$ and $\psi(t_2)$,
- $\psi(\theta_{i,j,n}(t))$ fuses i-th and j-th terminals for $1 \leq i < j \leq n$, and
- $\psi(\sigma_\alpha(t))$ renumbers $\alpha(i)$-th terminal as i-th terminal for $\alpha : [1..m] \to [1..n]$.

and their complete axiomatization is shown in Fig. 8.

This paper gives the complete axiomatization for the variation of the algebraic construction given in [ACPS93]. Our choice of formalization comes from its compatibility with SP Term, since SP Term seems the most suitable data structure for programming on graphs with bounded tree width [OHS03]. The idea for the proof of the completeness for ground cases (Section 4) comes from [BC87]; this paper further extends the result to non-ground cases (Section 5).

$$
\begin{array}{rcll}
(s \oplus t) \oplus u & = & s \oplus (t \oplus u) & \text{(R1)} \\
\sigma_\beta \cdot \sigma_\alpha(t) & = & \sigma_{\alpha\cdot\beta}(t) & \text{(R2)} \\
\sigma_{id}(t) & = & t & \text{(R3)} \\
\theta_{i,j,n} \cdot \theta_{i',j',n}(t) & = & \theta_{i',j',n} \cdot \theta_{i,j,n}(t) & \text{(R4-1)} \\
\theta_{i,j,n} \cdot \theta_{j,k,n}(t) & = & \theta_{i,j,n} \cdot \theta_{i,k,n}(t) & \text{(R4-2)} \\
\theta_{i,j,n} \cdot \theta_{j,k,n}(t) & = & \theta_{i,k,n} \cdot \theta_{j,k,n}(t) & \text{(R4-3)} \\
\theta_{i,i,n}(t) & = & t & \text{(R5)} \\
\sigma_\alpha(s) \oplus \sigma_{\alpha'}(t) & = & \sigma_{(\rightarrow_m \cdot \alpha)\oplus(\alpha' \cdot \leftarrow_p)}(t \oplus s) & \text{(R6)} \\
 & & \text{if } \alpha : [p] \to [n], \alpha' : [p'] \to [m] & \\
\theta_{i,j,m}(s) \oplus \theta_{i',j',n}(t) & = & \theta_{i,j,m} \cdot \theta_{m+i',m+j',m+n}(s \oplus t) & \text{(R7)} \\
\theta_{i,n+1,n+1}(t \oplus \mathbf{1}) & = & \sigma_{id\downarrow_{[n]} \oplus (n+1 \mapsto i)}(t) & \text{(R8)} \\
\theta_{i,j,n} \cdot \sigma_\alpha(t) & = & \sigma_\alpha \cdot \theta_{\alpha(i),\alpha(j),n}(t) \qquad \text{if } \alpha : [n] \to [n] & \text{(R9)} \\
\sigma_\alpha \cdot \theta_{i,j,n}(t) & = & \sigma_\beta \cdot \theta_{i,j,n}(t) & \text{(R10)} \\
 & & \text{if } \alpha(m), \beta(m) \in \{i,j\} \text{ or } \alpha(m) = \beta(m) \text{ for each } m. & \\
t \oplus \mathbf{0} & = & t & \text{(R11)}
\end{array}
$$

where $\alpha \cdot \leftarrow_p (i + p) = \alpha(i)$ and $\rightarrow_m \cdot \alpha(j) = m + \alpha(j)$.

Fig. 8. Axioms of algebraic construction of graphs in [BC87, Cou90]

7 Conclusion and Future Work

This paper presents the complete axiomatization for the variation of the algebraic construction given in [ACPS93]. Compared to the original algebraic construction in [ACPS93], we add σ^i_k (which is needed for completeness; the parallel composition p_k has the different infix notation $\oplus_k$ for readability), and omit s_k, which is defined as

$$
s_k(t_1, \cdots, t_k) = \begin{cases} r_2(e^2 \oplus_2 l^1_2(t_1)) & \text{if } k\ = 1, \\ r^{k+1}_{k+1}(l^1_{k+1}(t_1) \oplus_{k+1} \cdots \oplus_{k+1} l^k_{k+1}(t_k)) & \text{if } k \geq 2. \end{cases}
$$

Our final goal is to give the complete (finite) axiomatization of SP Term SP_k [OHS03], which precisely denotes graphs with tree width at most k. SP Term would be the most desirable algebraic construction for writing a functional program on graphs with bounded tree width, because it has only 2 functional constructors: the series composition s_k and the parallel composition $\oplus_k$ (though it has relatively many constants $e_k(i,j)$ and $\mathbf{k}$, which can be treated in a homogeneous way). We will use two approaches, one from rewriting and another from graph theory.

- We already know the complete axioms on $\mathcal{B}_\infty$, which consist of terms constructed from $l^i_k, \oplus_k, r_k, \sigma^i_k, e^2, \mathbf{0}$. We can define $s_k, e_k(i,j), \mathbf{k}$ like "macros". *Can we deduce equations on "macros" from equations on terms constructed from original function symbols ?*

- Minimal separator of a graph is essential for graphs with bounded tree width. We hope that the Menger-like property [Tho90] would help.

Acknowledgments

The author thanks Aart Middeldorp and Masahiko Sakai for information about references, and Philip Wadler for indicating an extension to non-ground cases. Thanks also to Jeremy Gibbons, Zhenjiang Hu, and Aki Takano for fruitful discussions. This research is supported by Special Coordination Funds for Promoting Science and Technology by Ministry of Education, Culture, Sports, Science and Technology, PRESTO by Japan Science and Technology Agency, and Kayamori Foundation of Informational Science Advancement.

References

[AA95] Z.M. Ariola and Arvind. Properties of a first-order functional language with sharing. *Theoretical Computer Science,* 146(1-2):69–108, 1995.

[ACPS93] S. Arnborg, B. Courcelle, A. Proskurowski, and D. Seese. An algebraic theory of graph reduction. *Journal of the Association for Computing Machinery,* 40(5):1134–1164, 1993.

[BC87] M. Bauderon and B. Courcelle. Graph expressions and graph rewritings. *Mathematical System Theory,* 20:83–127, 1987.

[BPT92] R.B. Borie, R.G. Parker, and C.A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica,* 7:555–581, 1992.

[Cou90] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science,* volume B, chapter 5, pages 194–242. Elsevier Science Publishers, 1990.

[CS92] V.-E. Căzănescu and G. Stefănescu. A general result on abstract flowchart schemes with applications to study of accessibility, reduction and minimization. *Theoretical Computer Science,* 99:1–63, 1992.

[Erw97] M. Erwig. Functional programming with graphs. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming, ICFP '97,* pages 52–65. ACM Press, 1997. SIGPLAN Notices 32(8).

[FS96] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions *(or, programs from outer space).* In *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'96,* pages 284–294. ACM Press, 1996.

[Gib95] J. Gibbons. An initial-algebra approach to directed acyclic graphs. In B. Moller, editor, *Mathematics of Program Construction, MPC'95,* pages 282–303, 1995. Lecture Notes in Computer Science, Vol. 947, Springer-Verlag.

[GMT02] J. Gustedt, O.A. Mæhle, and A. Telle. The treewidth of Java programs. In *Proc. 4th Workshop on Algorithm Engineering and Experiments, ALENEX 2002,* pages 86–97, 2002. Lecture Notes in Computer Science, Vol. 2409, Springer-Verlag.

[Has97] M. Hasegawa. Models of sharing graphs, 1997. PhD thesis, University of Edinburgh.

[MSvE94] M.J.Plasmeijer M.R. Sleep and M.C.J.D. van Eekelen, editors. *Term Graph Rewriting, Theory and Practice.* Wiley, 1994.

[OHS03] M. Ogawa, Z. Hu, and I. Sasano. Iterative-free program analysis. In *Proc. 8th ACM SIGPLAN International Conference on Functional Programming, ICFP'03,* pages 111–123. ACM Press, 2003. Uppsala, Sweden.

[RS86] N. Robertson and P.D. Seymour. Graph minors II. algorithmic aspects of tree-width. *Journal of Algorithms,* 7:309–322, 1986.

[SHTO00] I. Sasano, Z. Hu, M. Takeichi, and M. Ogawa. Make it practical: A generic linear-time algorithms for solving maximum-weightsum problems. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming, ICFP'00,* pages 137–149. ACM Press, 2000. Montreal, Canada.

[Tho90] R. Thomas. A Menger-like property of tree-width: The finite case. *Journal of Combinatorial Theory, Series B,* 48:67–76, 1990.

[Tho98] M. Thorup. All structured programs have small tree width and good register allocation. *Information and Computation,* 142:159–181, 1998.

[Wey39] H. Weyl. *Classical groups.* Princeton University Press, 1939.

Sub-Birkhoff

Vincent van Oostrom

Department of Philosophy, Universiteit Utrecht
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
Vincent.vanOostrom@phil.uu.nl

Abstract. For equational specifications validity coincides with derivability in equational logic, which in turn coincides with convertibility generated by the rewrite relation. It is shown that this correspondence, essentially due to Birkhoff, can be generalised in a uniform way to sub-equational logics such as Meseguer's rewriting logic.

1 Introduction

In order to motivate and state our generalisation, we illustrate the essential ingredients of the usual correspondence (see, e.g. Chapter 7 of [1] or Chapter 3 of [2]) between validity, derivability and convertibility by means of the following equational specification $\mathcal{EMul}$ of addition and multiplication:

$$\mathtt{A}(x,\mathtt{0}) \approx x \tag{1}$$
$$\mathtt{A}(x,\mathtt{S}(y)) \approx \mathtt{S}(\mathtt{A}(x,y)) \tag{2}$$
$$\mathtt{M}(x,\mathtt{0}) \approx \mathtt{0} \tag{3}$$
$$\mathtt{M}(x,\mathtt{S}(y)) \approx \mathtt{A}(x,\mathtt{M}(x,y)) \tag{4}$$

and the equation:

$$\mathtt{M}(\mathtt{S}(x),\mathtt{S}(\mathtt{0})) \approx \mathtt{S}(x) \tag{5}$$

On the one hand, (5) is *valid* for the specification $\mathcal{EMul}$ in the sense that it holds in any model. In algebraic semantics, terms are giving meaning by means of an algebra. The algebra is then called a model of the specification if each equation in the latter holds in the former. That is, the meanings of the left- and right-hand side of the equation are *identical,* for any assignment to the variables. For instance, the algebra $\mathcal{N}at$ having the set of natural numbers as carrier, and interpreting 0, S, A and M as zero, successor, addition and multiplication, respectively, is a model of $\mathcal{EMul}$ and one easily verifies that (5) holds in it. For instance, for the assignment α mapping every variable to the natural number 2, its left-hand side $\mathtt{M}(\mathtt{S}(x),\mathtt{S}(\mathtt{0}))$ is mapped to $(2+1) \times (0+1)$, and its right-hand side $\mathtt{S}(x)$ to $2 + 1$, i.e. both sides are mapped to 3.

© Springer-Verlag Berlin Heidelberg 2004

On the other hand, (5) being the conclusion of the proof tree:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{M}(x,\mathtt{S}(y)) \approx \mathtt{A}(x,\mathtt{M}(x,y))}\,(4)}{\mathtt{M}(\mathtt{S}(x),\mathtt{S}(0)) \approx \mathtt{A}(\mathtt{S}(x),\mathtt{M}(\mathtt{S}(x),0))}\,(\sigma) \qquad \dfrac{\dfrac{\dfrac{}{\mathtt{S}(x) \approx \mathtt{S}(x)}\,(\mathrm{ref}) \quad \dfrac{\dfrac{}{\mathtt{M}(x,0) \approx 0}\,(3)}{\mathtt{M}(\mathtt{S}(x),0) \approx 0}\,(\sigma)}{\mathtt{A}(\mathtt{S}(x),\mathtt{M}(\mathtt{S}(x),0)) \approx \mathtt{A}(\mathtt{S}(x),0)}\,(\mathtt{A}) \quad \dfrac{\dfrac{}{\mathtt{A}(x,0) \approx x}\,(1)}{\mathtt{A}(\mathtt{S}(x),0) \approx \mathtt{S}(x)}\,(\sigma)}{\mathtt{A}(\mathtt{S}(x),\mathtt{M}(\mathtt{S}(x),0)) \approx \mathtt{S}(x)}\,(\mathrm{trans})}{\mathtt{M}(\mathtt{S}(x),\mathtt{S}(0)) \approx \mathtt{S}(x)}\,(\mathrm{trans})$$

with substitution σ such that $x \mapsto \mathtt{S}(x)$ and $y \mapsto 0$, shows that it is *derivable* in equational logic (see Table 1).

On the gripping hand, *convertibility* of the sides of (5) is witnessed by:

$$\underline{\mathtt{M}(\mathtt{S}(x),\mathtt{S}(0))} \rightarrow \mathtt{A}(\mathtt{S}(x),\underline{\mathtt{M}(\mathtt{S}(x),0)}) \rightarrow \underline{\mathtt{A}(\mathtt{S}(x),0)} \rightarrow \mathtt{S}(x)$$

a sequence of forwards (and possibly backwards) rewrite steps.

We will refer to the correspondence between validity and derivability as *Birkhoff's theorem* since it is due to [3], and to the correspondence between derivability and convertibility as *logicality* (cf. [4]). Both correspondences are of fundamental importance in the study of programming language foundations, see [5], and can be seen as a justification of term rewriting itself. For instance, they allow for solving uniform word problems by means of complete term rewriting systems.

As argued by Meseguer, e.g. in [6], some specifications should not be considered to be equational. For instance an *equational* specification of a binary choice function ? (selecting either of its arguments) does not make sense, and would result in all terms being identified to one another. Instead an *ordering* specification is appropriate here:

$$?(x,y) \gtrsim x$$
$$?(x,y) \gtrsim y$$

As suggested by the notation, in a model of such an ordering specification each left-hand side should be greater than or equal to the corresponding right-hand side. Then, to salvage the correspondence between validity and derivability, the symmetry rule (sym) should dropped from the proof system of equational logic in Table 1, resulting in *ordering*[1] logic. In order to regain the correspondence between derivability and convertibility, backwards steps should be dropped from the latter. After this is done, both correspondences hold again as shown in [6].

Here we propose to generalise the correspondence as presented above for equational and ordering logic, to so-called *sub-equational* logics obtained by dropping a subset of the inference rules of equational logic. In particular, equational and ordering logic are obtained by dropping nothing (the empty set) and the

[1] Beware: in this paper we will use a systematic naming scheme. For instance, our ordering logic is known in the literature as *rewriting* logic.

$$\frac{}{s \approx t}(s \approx t \in E) \quad \frac{s \approx t}{\sigma(s) \approx \sigma(t)}(\sigma{:}X \to T(\Sigma, X)) \quad \frac{s_1 \approx t_1 \quad \dots \quad s_n \approx t_n}{f(s_1, \dots, s_n) \approx f(t_1, \dots, t_n)}(f \in \Sigma)$$

$$\frac{}{s \approx s}(\text{ref}) \quad \frac{s \approx t}{t \approx s}(\text{sym}) \quad \frac{s \approx t \quad t \approx u}{s \approx u}(\text{trans})$$

Table 1. Equational logic

singleton {(sym)}, respectively. We argue that sub-equational logics are interesting for the same reason that ordering logic is interesting [6]: enforcing too many inference rules would conflate notions which one would like to keep distinct. Just as confusing the forwards and backwards directions (as enforced by symmetry) would be a brutal [6] act in case of the (non-confluent) ordering specification for choice above, to confuse 'not being able to do anything' with 'being able to do a trivial step' (as enforced by reflexivity) is a brutal act in case of a (terminating) specification such as

$$a_{n+1} > a_n$$

Similarly, single-steps should, *a priori,* not be confused with many-steps (as would be enforced by transitivity) in case of a *step* specification (a.k.a. term rewriting system), since by the choice for that form of specification one implicitly specifies that one is interested in individual steps (think e.g. of complexity).

Based on the above ideas, we give a parametrised account of both Birkhoff's theorem (Section 3) and logicality (Section 4) for sub-equational specifications (Section 2). The proofs of our results are simple, as they are just variations on the existing simple proofs for equational logic. The main effort will be in formalising both the results and their proofs in a way which allows for their parametrisation. As a side-effect of this parametrisation the proof structure becomes clearer, which may be of some didactic value. Because of it, we have made an effort to make the paper self-contained.

2 Sub-equational Specifications

A sub-equational specification can be thought of as an equational specification together with a set of inference modes specifying how its equations are to be interpreted (e.g. indeed as equations, or alternatively as rewrite rules, or ...).

Definition 1. *A* signature $(f, g, h \in)\Sigma$ *is a set of* symbols, *each of which comes equipped with a natural number* arity.

The subset of Σ consisting of all symbols of arity n, is denoted by $\Sigma^{(n)}$. Elements of $\Sigma^{(0)}$ are called *constants.* Throughout, we assume $(x, y, z \in)X$ to be a signature disjoint from Σ, consisting of an infinite number of constants called *variables.*

Definition 2. *The set* $(s,t,u \in)T(\Sigma)$ *of* Σ-terms *is inductively defined by:*

– $fs_1 \dots s_n$ *is a term, if* f *is an* n*-ary symbol and* $s_1,\dots,s_n$ *are terms.*

The set $T(\Sigma,X)$ *of* Σ-terms over X *is defined as* $T(\Sigma \cup X)$.

As is customary, we may write $f(s_1,\dots,s_n)$ to denote $fs_1 \dots s_n$.

Example 1. Consider the signature Σ consisting of the nullary symbol `0`, the unary symbol `S` and the binary symbols `A` and `M`. Some Σ-terms are `0`,`S0`,`SS0`, `A00` and `MA0S00`. E.g. the last term is also denoted by `M(A(0, S(0)), 0)`. An example of a Σ-term over X is $\mathtt{A}(x, \mathtt{S}(y))$.

Definition 3. *A* Σ-statement *is a pair of* Σ*-terms.*

Definition 4. *A* (sub-equational) specification *is a quadruple* $\mathcal{S} := \langle \Sigma, X, S, L\rangle$ *with* S *a set of* $\Sigma \cup X$*-statements and* L *a set of* inference modes *which is a subset of* {(embedding), (compatibility), (reflexivity), (symmetry), (transitivity)}.

We will abbreviate the respective inference modes to (emb), (comp), (ref), (sym), and (trans). The idea is that for a sub-equational specification $\mathcal{S} := \langle \Sigma, X, S, L\rangle$, the modes of inference will specify how the pairs in S are to be dealt with, both at the semantical and the syntactical level (both to be presented below).

Example 2. An *equational* specification is a sub-equational specification having EL := {(emb), (comp), (ref), (sym), (trans)} as modes of inferences. A statement (s,t) of such a specification will be called an *equation,* and written as $s \approx t$.

The equational specification $\mathcal{EM}ul$ in the introduction consists of four Σ-equations over X, that is, $\Sigma \cup X$-equations, with Σ as in Example 1.

Example 3. In an *ordering* specification all modes of inference except for (sym) are present RL := {(emb), (comp), (ref), (trans)}.[2] A statement (s,t) of such a specification will be called an *ordering,* and written as $s \gtrsim t$.

The specification of ? in the introduction is an ordering specification.

Similarly term rewriting systems[3] are rendered as sub-equational specifications by taking {(emb), (comp)} as modes of inference. Its statements are written using $\rightarrow$, as usual.[4] The TRS corresponding to $\mathcal{EM}ul$ will be denoted by $\mathcal{RM}ul$.

We will not list all possible sets of inference modes, but only mention one more example, which will be used later.

Example 4. Removing {(ref), (sym)} from the modes of inference of $\mathcal{EM}ul$ yields what we call a *positive* ordering specification $\mathcal{TM}ul$, having as fourth component TL := {(emb), (comp), (trans)}. A statement (s,t) of such a specification will be called a *positive* ordering, and written as $s > t$.

[2] This corresponds to Meseguer's rewriting logic.

[3] To be precise, our term rewriting systems (TRSs) correspond to the pseudo-TRSs of [1, page 36] since we do not impose the usual further restrictions on rules.

[4] Note that although transition system specifications usually employ the $\rightarrow$-notation as well, the (comp)-inference mode is absent for them.

3 Birkhoff

In this section the correspondence between validity (Subsection 3.1) and derivability (Subsection 3.2) for sub-equational specifications is presented in two stages. In Subsection 3.3 we first present a correspondence between *relational* validity and derivability, which is then extended in Subsection 3.4 to a correspondence between validity and derivability by a quotient construction.

3.1 Validity

As usual, algebras are used to give meaning to the terms of a sub-equational specification. However, the notion of validity of a statement (s,t) with respect to a specification $\mathcal{S}$ will now be parametrised over its modes of inference.

Definition 5. *A* Σ-algebra $\mathcal{A}$ *consists of a* carrier *set A, and a mapping that associates with each symbol* $f \in \Sigma^{(n)}$ *a function* $f^{\mathcal{A}}{:}A^n \to A$, *for every* n.

An *assignment* is an X-algebra. For a Σ-algebra $\mathcal{A}$ and an assignment α having the same carrier, $\mathcal{A} \cup \alpha$ denotes the obvious $\Sigma \cup X$-algebra.

Example 5. 1. The algebra $\mathcal{N}at$ of the introduction is a Σ-algebra, for the signature Σ of Example 1. For the same carrier, α of the introduction is an example of an assignment.
2. The Σ*-term* algebra $\mathcal{T}(\Sigma)$ has $T(\Sigma)$ as carrier, and interpretation defined by, for all n, all $f \in \Sigma^{(n)}$, and all $s_1,\dots,s_n \in T(\Sigma)$: $f^{\mathcal{T}(\Sigma)}(s_1,\dots,s_n) := f(s_1,\dots,s_n)$.

Definition 6. *A* Σ-homomorphism h *from a* Σ*-algebra* $\mathcal{A}$ *to a* Σ*-algebra* $\mathcal{B}$, *is a map from the carrier A of* $\mathcal{A}$ *to the carrier B of* $\mathcal{B}$, *such that for all* n, *all* $f \in \Sigma^{(n)}$, *and all* $a_1,\dots,a_n \in A$: $h(f^{\mathcal{A}}(a_1,\dots,a_n)) = f^{\mathcal{B}}(h(a_1),\dots,h(a_n))$.

It is easy to see that $\mathcal{T}(\Sigma)$ is *initial* among Σ-algebras, i.e. for any Σ-algebra $\mathcal{A}$, there is a unique homomorphism from $\mathcal{T}(\Sigma)$ to $\mathcal{A}$, which we denote by $[\![\mathcal{A}]\!]$.

Example 6. 1. The unique homomorphism $[\![\mathcal{N}at \cup \alpha]\!]$ which maps $\mathcal{T}(\Sigma, X)$ to $\mathcal{N}at \cup \alpha$, with $\mathcal{N}at$ and α as in Example 5.1, is concretely defined by:
   - $x \mapsto 2$, for $x \in X$
   - $f(s_1,\dots,s_n) \mapsto f^{\mathcal{N}at}(n_1,\dots,n_n)$, for $f \in \Sigma$ and $s_i \mapsto n_i$

   For instance, $\mathtt{M}(\mathtt{S}(x),\mathtt{S}(\mathtt{0}))$ is mapped to $(2+1) \times (0+1)$, i.e. to 3.
2. A *substitution* is the unique homomorphism $[\![\mathcal{T}(\Sigma,X) \cup \sigma]\!]$ of some assignment σ. For instance, if σ assigns $\mathtt{S}(\mathtt{S}(\mathtt{0}))$ to x, then applying the substitution to $\mathtt{M}(\mathtt{S}(x),\mathtt{S}(\mathtt{0}))$ yields $\mathtt{M}(\mathtt{S}(\mathtt{S}(\mathtt{S}(\mathtt{0}))),\mathtt{S}(\mathtt{0}))$. We will often abbreviate the substitution to just σ.

Definition 7. *Let* $\mathcal{S} := \langle \Sigma, X, S, L\rangle$ *be a specification. A* relational model *of* $\mathcal{S}$ *is pair* $(\mathcal{A}, R)$ *consisting of a* Σ*-algebra* $\mathcal{A}$ *and a relation* R *on the carrier A of the algebra, satisfying each rule* ℓ *in Table 2, for* $\ell \in L$. *Here*

$$\frac{}{s\ R\ t}\,(\text{emb},\ (s,t) \in S) \qquad \frac{a_1,\ldots,a_n =[R]\ b_1,\ldots,b_n}{f^{\mathcal{A}}(a_1,\ldots,a_n)\ R\ f^{\mathcal{A}}(b_1,\ldots,b_n)}\,(\text{comp},\ f \in \Sigma)$$

$$\frac{}{a\ R\ a}\,(\text{ref}) \quad \frac{a\ R\ b}{b\ R\ a}\,(\text{sym}) \quad \frac{a\ R\ b \quad b\ R\ c}{a\ R\ c}\,(\text{trans})$$

Table 2. Relational models

- *$s\ R\,t$ expresses that for all assignments α, it holds $[\![\mathcal{A} \cup \alpha]\!](s)\ R\ [\![\mathcal{A} \cup \alpha]\!](t)$.*
- *=[R] expresses that corresponding components of $a_1,\ldots,a_n$ and $b_1,\ldots,b_n$ are identical, except for one index, say i, for which $a_i\ R\ b_i$.*

(s,t) is relationally valid *in $\mathcal{S}$, $\models\!\!\models s\ \mathcal{S}\ t$, if $s\ R\ t$ holds in every relational model $(\mathcal{A},R)$ of $\mathcal{S}$.*

Remark 1. Since $s\ R\,t$ depends on $\mathcal{A}$ as well, formally we should consider it to be an abbreviation of $s\ R_{\mathcal{A}}\ t$. One may think of relational models as models of a predicate logic with one binary predicate symbol.

The (comp)-rule is a direct generalisation of the usual compatibility rules found in mathematics and rewriting. In the following examples, the relational models for equational, ordering, and positive ordering specifications are characterised. To that end, recall that a relation R is a *congruence* relation for an algebra $\mathcal{A}$, if it is an equivalence relation which is *preserved* by the *operations* of $\mathcal{A}$, i.e. such that for every n-ary operation ϕ, if $a_1\ R\ b_1,\ldots,a_n\ R\ b_n$, then $\phi(a_1,\ldots,a_n)\ R\ \phi(b_1,\ldots,b_n)$.[5] In each example, we will assume the relational model to be $(\mathcal{A},R)$.

Example 7. . In case of an equational specification, R is seen to be a congruence relation as follows. Since {(ref), (sym), (trans)} $\subseteq$ EL, R is an equivalence relation. To see that it is a congruence relation, suppose ϕ is an n-ary operation in $\mathcal{A}$ and $a_1\ R\ b_1,\ldots a_n\ R\ b_n$, then we conclude from

$$\begin{array}{l} \phi(a_1,\ldots,a_n)\ R\ \phi(b_1,\ldots,a_n) \\ \qquad\qquad\quad \vdots \qquad\quad \ddots \\ \qquad\qquad\quad R\ \phi(b_1,\ldots,b_n) \end{array}$$

using {(comp), (trans)} $\subseteq$ EL.

Models of equational specifications in the standard sense of the introduction give rise to relational models, just by pairing them up with the identity relation

[5] Hence the distinction between compatibility and congruence is that the latter requires all corresponding premisses to be related, whereas the former requires exactly one pair of corresponding premisses to be related (and the rest to be identical).

id. For instance, $(\mathcal{N}at, \mathsf{id})$, is a relational model of $\mathcal{E}\mathcal{M}ul$. id is trivially a congruence relation, and (emb) is forced to hold by the assumption that $\mathcal{N}at$ is a model, in the standard sense, of $\mathcal{E}\mathcal{M}ul$.

However, note that R is in general *not* forced to be the identity relation. For instance, an example of a relational model for the equational specification $\mathcal{E}\mathcal{M}ul$ consists of its term algebra $\mathcal{T}(\Sigma, X)$ and the convertibility relation $\leftrightarrow^*_{\mathcal{E}\mathcal{M}ul}$ (see Example 16).

Example 8. In case of an ordering specification, R *is* seen to be an operation-preserved quasi-order. That it is a quasi-order, i.e. reflexive and transitive, follows since $\{(\text{ref}), (\text{trans})\} \subseteq \mathrm{E}L$. That it is operation-preserved follows as in the previous item.

Example 9. In case of a positive ordering specification, the relation R is an operation-preserved transitive relation.

Example 10. Any relational model for the equational specification $\mathcal{E}\mathcal{M}ul$ is is automatically a relational model for its associated TRS $\mathcal{R}\mathcal{M}ul$. Of course, this does not hold the other way around. For instance, combining the polynomial interpretation of [1, Example 6.2.13] with the natural order $>$ on the natural numbers yields a relational model of $\mathcal{R}\mathcal{M}ul$, but not of $\mathcal{E}\mathcal{M}ul$, because $>$ is not symmetric. Although symmetry is lacking, transitivity is not, hence this interpretation *is* a model of the positive ordering specification $\mathcal{T}\mathcal{M}ul$.

As the first example shows, there is a mismatch between the notion of a model and that of a relational model. It is analagous to the difference between the notions of model of predicate logic with and without equality: in the former the interpretation of the binary equality predicate is fixed to the identity relation, whereas in the latter its interpretation can in principle be any relation (possibly satisfying some constraints). That is, there are many more relational models than there are models. This mismatch will be overcome in Subsection 3.4.

3.2 Derivability

Definition 8. *The judgment that a statement* (s, t) *is derivable by means of sub-equational logic, for a given sub-equational specification* $\mathcal{S}$*, is denoted by* $\vdash s\ \mathcal{S}\ t$*. The axioms and rules of* sub-equational logic *are the ones listed in Table 3. The* theory $\underline{\mathcal{S}}$ *of* $\mathcal{S}$ *is the relation on terms, consisting of all derivable pairs.*

Here derivability of a statement means that it is the conclusion of some proof tree built from the inference rules, as usual. Note that an inference rule only applies when it is an allowed mode of inference, according to the specification. Furthermore, not all modes of inference of standard equational logic as presented in Table 1 are (directly) at our disposal in sub-equational logic, not even for an equational sub-equational specification, where all modes of inference are

$$\frac{}{\sigma(s)\ \underline{\mathcal{S}}\ \sigma(t)}\ (\text{emb}, (s,t)\in S, \sigma{:}X\to\mathcal{T}(\Sigma,X)) \qquad \frac{s_1,\dots,s_n =[\underline{\mathcal{S}}]\ t_1,\dots,t_n}{f(s_1,\dots,s_n)\ \underline{\mathcal{S}}\ f(t_1,\dots,t_n)}\ (\text{comp}, f\in\Sigma)$$

$$\frac{}{s\ \underline{\mathcal{S}}\ s}\ (\text{ref}) \quad \frac{s\ \underline{\mathcal{S}}\ t}{t\ \underline{\mathcal{S}}\ s}\ (\text{sym}) \quad \frac{s\ \underline{\mathcal{S}}\ t \quad t\ \underline{\mathcal{S}}\ u}{s\ \underline{\mathcal{S}}\ u}\ (\text{trans})$$

Table 3. Sub-equational logic for sub-equational specification $\mathcal{S} := \langle \Sigma, X, S, L\rangle$

available. The reason is that the standard inference rules of equational logic exhibit some dependencies which we have avoided here, in order to make the connexion between syntax and semantics smoother. In particular, the equation- and substitution-rule have been merged into the (emb)-rule. Furthermore, the (comp)-rule allows one to relate only one argument at the time whereas the standard presentation has a congruence-rule. Nevertheless, the two presentations are easily seen to be equivalent as illustrated by the following example.

Example 11. Redrawing the proof tree of the introduction for the sub-equational specification corresponding to $\mathcal{EMul}$, omitting parentheses and $\underline{\mathcal{EMul}}$ to save space, yields

$$\dfrac{\dfrac{}{(\mathtt{M}(\mathtt{S}x,\mathtt{S0}),\mathtt{A}(\mathtt{S}x,\mathtt{M}(\mathtt{S}x,\mathtt{0})))}\,(\sigma) \quad \dfrac{\dfrac{\dfrac{}{(\mathtt{M}(\mathtt{S}x,\mathtt{0}),\mathtt{0})}\,(\sigma)}{(\mathtt{A}(\mathtt{S}x,\mathtt{M}(\mathtt{S}x,\mathtt{0})),\mathtt{A}(\mathtt{S}x,\mathtt{0}))}\,(\mathtt{comp},\ \mathtt{A}) \quad \dfrac{}{(\mathtt{A}(\mathtt{S}x,\mathtt{0}),\mathtt{S}x)}\,(\sigma)}{(\mathtt{A}(\mathtt{S}x,\mathtt{M}(\mathtt{S}x,\mathtt{0})),\mathtt{S}x)}\,(\mathtt{trans})}{(\mathtt{M}(\mathtt{S}x,\mathtt{S0}),\mathtt{S}x)}\,(\mathtt{trans})$$

More precisely, for a given equational specification $\mathcal{E}$, its derivability in equational logic $\mathcal{E} \vdash s \approx t$ coincides with its derivability $\vdash s\ \mathcal{E}\ t$ in equational sub-equational logic.

3.3 Relational Term Model

Derivability can be related to relational validity, by constructing a so-called relational term model for a specification.

Lemma 1 (Term Model). *$\vdash s\ \mathcal{S}\ t$ iff $\models s\ \mathcal{S}\ t$, for any specification $\mathcal{S}$.*

Proof. Define the relational *term* model $\mathcal{M}(\mathcal{S})$ of a sub-equational specification $\mathcal{S} := \langle \Sigma, X, S, L\rangle$ as the pair $(\mathcal{T}(\Sigma,X), \underline{\mathcal{S}})$, where $\mathcal{T}(\Sigma,X)$ is the term algebra and $\underline{\mathcal{S}}$ the theory of $\mathcal{S}$.

To prove the if-direction (completeness), it suffices to prove that $\mathcal{M}(\mathcal{S})$ is a relational model for $\mathcal{S}$, by the choice of the theory of $\mathcal{S}$ as the relation of $\mathcal{M}(\mathcal{S})$. Since $\mathcal{T}(\Sigma,X)$ is a $\Sigma \cup X$-algebra by Example 5, it certainly is a Σ-algebra. Hence to verify that $\mathcal{M}(\mathcal{S})$ is indeed a relational model for $\mathcal{S}$, it remains to show that rule ℓ holds in theory $\underline{\mathcal{S}}$, for each $\ell \in L$. Intuitively, this will hold by the 1–1 correspondence between the rules of relational models in Table 2 and the inference rules of sub-equational logics in Table 3. For a proof, we distinguish cases for the rules.

(emb) Suppose $(s,t) \in S$. By the (emb)-rule of Table 2, we must verify that for any assignment α, it holds that $[\![\mathcal{T}(\Sigma,X) \cup \alpha]\!](s)$ is related by $\underline{S}$ to $[\![\mathcal{T}(\Sigma,X) \cup \alpha]\!](t)$. By the definition of substitution, this is just the same as saying that $\sigma(s)$ is related by $\underline{S}$ to $\sigma(t)$ for any substitution σ. Which holds by the (emb)-inference rule of the logic.

(comp), (ref), (sym), (trans) Each rule in Table 2 directly follows from the corresponding rule of Table 3, where (comp) also uses that symbols are interpreted as themselves in relational term models.

To prove the only-if-direction (soundness), it suffices to prove by induction on derivations (proof trees) that pairs in the theory $\underline{S}$, are related in any relational model $(\mathcal{A}, R)$ of S. The proof is by cases on the modes of inference in L, showing that the statement holds for a proof whose conclusion uses inference rule ℓ, by using rule ℓ of Table 2.

(emb) Suppose $(s,t) \in S$ and let σ be some subtitution. We have to show $[\![\mathcal{A} \cup \alpha]\!](\sigma(s))\ R\ [\![\mathcal{A} \cup \alpha]\!](\sigma(t))$, for any assignment α. Suppose we can show the so-called *semantic* substitution lemma:

$$[\![\mathcal{A} \cup \alpha]\!](\sigma(u)) = [\![\mathcal{A} \cup \alpha_\sigma]\!](u) \tag{6}$$

where the assignment α_σ maps a variable $x \in X$ to the value of $\sigma(x)$ in the algebra $\mathcal{A}$ under the assignment α, i.e. to $[\![\mathcal{A} \cup \alpha]\!](\sigma(x))$.
Then we are done, since

$$[\![\mathcal{A} \cup \alpha]\!](\sigma(s)) = [\![\mathcal{A} \cup \alpha_\sigma]\!](s)\ R\ [\![\mathcal{A} \cup \alpha_\sigma]\!](t) = [\![\mathcal{A} \cup \alpha]\!](\sigma(t))$$

by (emb) of Table 2, and the semantic substitution lemma (twice).
It remains to show (6), which is proven by induction on $u \in \mathcal{T}(\Sigma,X)$.
(variable)

$$\begin{aligned} [\![\mathcal{A} \cup \alpha]\!](\sigma(x)) &= \alpha_\sigma(x) \\ &= [\![\alpha_\sigma]\!](x) \\ &= [\![\mathcal{A} \cup \alpha_\sigma]\!](x) \end{aligned}$$

(symbol) For all n, all $f \in \Sigma^{(n)}$, and all $s_1,\ldots,s_n \in \mathcal{T}(\Sigma,X)$:

$$\begin{aligned} [\![\mathcal{A} \cup \alpha]\!](\sigma(f(s_1,\ldots,s_n))) &= [\![\mathcal{A} \cup \alpha]\!](f(\sigma(s_1),\ldots,\sigma(s_n))) \\ &= f^{\mathcal{A}\cup\alpha}([\![\mathcal{A} \cup \alpha]\!](\sigma(s_1)),\ldots,[\![\mathcal{A} \cup \alpha]\!](\sigma(s_n))) \\ &=_{\mathrm{IH}} f^{\mathcal{A}\cup\alpha}([\![\mathcal{A} \cup \alpha_\sigma]\!](s_1),\ldots,[\![\mathcal{A} \cup \alpha_\sigma]\!](s_n)) \\ &= f^{\mathcal{A}}([\![\mathcal{A} \cup \alpha_\sigma]\!](s_1),\ldots,[\![\mathcal{A} \cup \alpha_\sigma]\!](s_n)) \\ &= f^{\mathcal{A}\cup\alpha_\sigma}([\![\mathcal{A} \cup \alpha_\sigma]\!](s_1),\ldots,[\![\mathcal{A} \cup \alpha_\sigma]\!](s_n)) \\ &= [\![\mathcal{A} \cup \alpha_\sigma]\!](f(s_1,\ldots,s_n)). \end{aligned}$$

Which concludes the proof of the semantic substitution lemma.
(comp), (ref), (sym), (trans) As for the other direction, these are trivial. □

Note that what we have really is a term model, i.e. terms are interpreted as terms (even stronger: as themselves), unlike the standard term models where terms are interpreted as equivalence classes of terms. The latter will be constructed in the following subsection.

3.4 Quotienting Out a Maximal Congruence

To overcome the mismatch between relational models and models observed above, we show that any relational model can be turned into a model, by quotienting out a maximal congruence relation. Quotienting out a congruence relation $\cong$ consists in taking $\cong$-equivalence classes of elements as new elements.

Definition 9. *Let $\mathcal{M} := (\mathcal{A}, R)$ be a relational model of $\mathcal{S} := \langle \Sigma, X, S, L \rangle$ and let $\cong$ be a congruence relation on the carrier A of $\mathcal{A}$. The* quotient $\mathcal{M}/_{\cong}$ of $\mathcal{M}$ *by $\cong$ is the pair $(\mathcal{A}/_{\cong}, R/_{\cong})$ defined by:*

- *The* quotient *algebra $\mathcal{A}/_{\cong}$ of $\mathcal{A}$ by $\cong$ is defined by:*
  - *The carrier $A/_{\cong}$ of $\mathcal{A}/_{\cong}$ consists of the $\cong$-equivalence classes $[a]_{\cong}$ for $a \in A$.*
  - *The interpretation of symbols is given by: for all n, for all $f \in \Sigma^{(n)}$, and all $a_1, \ldots, a_n \in A$*

$$f^{\mathcal{A}/_{\cong}}([a_1]_{\cong}, \ldots [a_n]_{\cong}) := [f^{\mathcal{A}}(a_1, \ldots, a_n)]_{\cong}$$

- *The relation $R/_{\cong}$ on the carrier $A/_{\cong}$ of $\mathcal{A}/_{\cong}$, is defined by:*

$$[a]_{\cong} \; R/_{\cong} \; [b]_{\cong} := a \cong ; R ; \cong b, \text{ where } ; \text{ denotes relation composition}$$

Neither the definition of the quotient algebra nor of the quotient relation depends on the choice of the representatives, because $\cong$ is a congruence relation. Under some constraints, taking quotients preserves and 'reflects' modelhood.

Lemma 2 (Quotient). *Let $\mathcal{S} := \langle \Sigma, X, S, L \rangle$ be a specification, $\mathcal{M} := (\mathcal{A}, R)$ be a relational model of $\mathcal{S}$, and $\cong$ a congruence relation on the carrier A of $\mathcal{A}$.*

- *If $\cong \subseteq R^*$, then $\mathcal{M}/_{\cong}$ is a relational model of $\mathcal{S}$ again.*
- *If moreover* (trans) $\in L$*, then $[a]_{\cong} \; R/_{\cong} \; [b]_{\cong}$ implies $a \; R \; b$.*

Proof. We first show the first item. Let $\mathcal{S} := \langle \Sigma, X, S, L \rangle$. We must verify for each $\ell \in L$, that if R satisfies the inference rule ℓ for $\mathcal{M}$, then $R/_{\cong}$ does so for $\mathcal{M}/_{\cong}$. Except for the (emb) rule all cases are easy:

(comp) We have to show that $[a_1]_{\cong}, \ldots [a_n]_{\cong} =[R/_{\cong}] \; [b_1]_{\cong}, \ldots [b_n]_{\cong}$ implies $f^{\mathcal{A}/_{\cong}}([a_1]_{\cong}, \ldots [a_n]_{\cong}) \; R/_{\cong} \; f^{\mathcal{A}/_{\cong}}([b_1]_{\cong}, \ldots [b_n]_{\cong})$. By the assumption it holds $a_i \cong b_i$, except say for j, for which $a_j \cong ; R ; \cong b_j$. By (comp) for R and congruence of $\cong$, we obtain $f^{\mathcal{A}}(a_1, \ldots, a_n) \cong ; R ; \cong f^{\mathcal{A}}(b_1, \ldots, b_n)$, from which the claim follows by definition of $f^{\mathcal{A}/_{\cong}}$.

(ref) If R is reflexive, then $\cong ; R ; \cong$ is reflexive by the assumption that $\cong$ is a congruence relation hence reflexive, so $R/_{\cong}$ is reflexive as well.

(sym) If R is symmetric, then $\cong ; R; \cong$ is symmetric by the assumption that $\cong$ is a congruence relation hence symmetric, so $R/_{\cong}$ is symmetric as well.
(trans) If R is transitive, then $\cong ; R; \cong$ is transitive by the assumption that $\cong$ is contained in the reflexive–transitive closure of R, so $R/_{\cong}$ is transitive as well.

It remains to verify the (emb) rule holds for $R/_{\cong}$ under the assumption that it holds for R. So suppose $(s,t) \in S$. We have to show

$$[\![\mathcal{A}/_{\cong} \cup \beta]\!](s)\ R/_{\cong}\ [\![\mathcal{A}/_{\cong} \cup \beta]\!](t)$$

for any assignment β of $\cong$-equivalence classes of A, to variables. We will show the so-called *syntactic* substitution lemma:

$$[\![\mathcal{A}/_{\cong} \cup \beta]\!](u) = [[\![\mathcal{A} \cup \alpha]\!](u)]_{\cong} \tag{7}$$

for any assignment α 'picking' elements from those classes, i.e. such that α maps each variable x to an element of x^{β}. Then we conclude, by definition of $R/_{\cong}$:

$$[\![\mathcal{A}/_{\cong} \cup \beta]\!](s) = [[\![\mathcal{A} \cup \alpha]\!](s)]_{\cong}\ R/_{\cong}\ [[\![\mathcal{A} \cup \alpha]\!](t)]_{\cong} = [\![\mathcal{A}/_{\cong} \cup \beta]\!](t)$$

using the assumption that $[\![\mathcal{A} \cup \alpha]\!](s)\ R\ [\![\mathcal{A} \cup \alpha]\!](t)$ for any α. It remains to show (7) for all $u \in T(\Sigma, X)$, which we prove by induction on u.

(variable) Since α was assumed to pick elements from β:

$$[\![\mathcal{A}/_{\cong} \cup \beta]\!](x) = x^{\beta} = [x^{\alpha}]_{\cong} = [[\![\mathcal{A} \cup \alpha]\!](x)]_{\cong}.$$

(symbol) For all n, all $f \in \Sigma^{(n)}$, and all $u_1,\ldots,u_n \in T(\Sigma, X)$:

$$\begin{aligned}
[\![\mathcal{A}/_{\cong} \cup \beta]\!](f(u_1,\ldots,u_n)) &= f^{\mathcal{A}/_{\cong}\cup\beta}([\![\mathcal{A}/_{\cong} \cup \beta]\!](u_1),\ldots,[\![\mathcal{A}/_{\cong} \cup \beta]\!](u_n)) \\
&=_{\mathrm{IH}} f^{\mathcal{A}/_{\cong}\cup\beta}([[\![\mathcal{A} \cup \alpha]\!](u_1)]_{\cong},\ldots,[[\![\mathcal{A} \cup \alpha]\!](u_n)]_{\cong}) \\
&= f^{\mathcal{A}/_{\cong}}([[\![\mathcal{A} \cup \alpha]\!](u_1)]_{\cong},\ldots,[[\![\mathcal{A} \cup \alpha]\!](u_n)]_{\cong}) \\
&= [f^{\mathcal{A}}([\![\mathcal{A} \cup \alpha]\!](u_1),\ldots,[\![\mathcal{A} \cup \alpha]\!](u_n))]_{\cong} \\
&= [f^{\mathcal{A}\cup\alpha}([\![\mathcal{A} \cup \alpha]\!](u_1),\ldots,[\![\mathcal{A} \cup \alpha]\!](u_n))]_{\cong} \\
&= [[\![\mathcal{A} \cup \alpha]\!](f(u_1,\ldots,u_n))]_{\cong}.
\end{aligned}$$

Showing the second item is easy: by definition $[a]_{\cong}\ R/_{\cong}\ [b]_{\cong}$ iff $a \cong ; R ; \cong b$. By the assumption $\cong \subseteq R^*$, this implies $a\ R^* ; R ; R^*\ b$, from which $a\ R\ b$ follows by the assumption (trans) $\in L$. □

Example 12. Consider the relational model $(T(\Sigma, X), \leftrightarrow^*_{\mathcal{EM}ul})$ of $\mathcal{EM}ul$ of Example 7. Taking for $\cong$ the convertibility relation $\leftrightarrow^*_{\mathcal{EM}ul}$, we see that it satisfies the first condition of Lemma 2, hence that the convertibility relation itself can be quotiented out. As one easily checks this yields a relational model having the classes of convertible terms as elements, and having the identity relation id as relation. Note that the first component of the resulting model, is a model in the sense of the previous section. That is, we have constructed a model from a relational model.

The construction in the example can be generalised in the sense that if the relation of a relational model contains a non-trivial congruence it can be quotiented out. In fact, we take this as the defining property of a model.

Definition 10. *A* model *of $\mathcal{S}$ is a congruence-free relational model. Here, a relational model $(\mathcal{A}, R)$ of a specification $\mathcal{S}$ is* congruence-free *if the reflexive–transitive closure R^* of R contains no congruence relations other than the identity relation* id. *We say (s,t) is valid, written $\models s \mathrel{\mathcal{S}} t$, incase $s \mathrel{R} t$ in all models $(\mathcal{A}, R)$ of $\mathcal{S}$.*

Hence, validity is obtained from relational validity by restricting the relational models to models.

Proposition 1. *For any relational model $\mathcal{M} := (\mathcal{A}, R)$, $\mathcal{M}/_{\cong}$ is a model where $\cong$ is a maximal congruence relation $\cong$, such that $\cong \subseteq R^*$.*

Proof. That a maximal congruence relation exists follows from Kuratowski's Lemma, since the union of the congruence relations in a chain is easily seen to be a congruence relation again. That the quotient $\mathcal{M}/_{\cong}$ is a relational model follows from Lemma 2, and that it is congruence-free holds, since otherwise the 'offending' congruence $\cong'$ could have been composed with $\cong$ right away. More precisely, in such a case, defining a to be related to b iff $[a]_{\cong} \cong' [b]_{\cong}$, would have given a congruence relation on A still contained in R^*, but larger than $\cong$ contradicting the latter's maximality. □

Let R be the relation of a relational model for $\mathcal{S}$.

Example 13. As seen above, R itself is the maximal congruence in the case of an equational specification, and models, i.e. congruence-free relational models, are in 1–1 correspondence with the models of the introduction. That is, for an equational specification $\mathcal{E}$, the standard and sub-equational notions of validity coincide.

Generalising the example, one notes that if R is both transitive and operation-preserved, such as is the case for (positive) ordering logic, then the reflexive closure of $R \cap (R^{-1})$ is the largest congruence relation contained in R^*. This maximal congruence just identifies all objects in strongly connected components of R. (Note that if R is terminating, then quotienting does nothing.) Hence models of ordering and positive ordering specifications have *partial* orders (reflexive, transitive and anti-symmetric relations) and *positive* orders (transitive and anti-symmetric relations) respectively, as relations.

Example 14. The models of ordering specifications are better known as quasi-models [1, Definition 6.5.30].

By the quotient construction, checking validity on relational models can be restricted to checking validity on models in case of transitive specifications, that is, which have (trans) as mode of inference.

Lemma 3. *$\models s\,\mathcal{S}\,t$ iff $\models s\,\mathcal{S}\,t$, for transitive specifications $\mathcal{S}$.*

Proof. The only-if-direction holds since models are a special case of relational models. The if-direction follows, since by Proposition 1, any relational model of $\mathcal{S}$ gives rise to a model, in which s and t are related by the assumption $\models s\,\mathcal{S}\,t$, but then s and t were related in the relational model as well, by the second item of the Quotient Lemma 2 using the assumption that $\mathcal{S}$ is transitive. □

Theorem 1 (Birkhoff). *$\vdash s\,\mathcal{S}\,t$ iff $\models s\,\mathcal{S}\,t$, for transitive $\mathcal{S}$.*

Proof. By Lemmas 3 and 1. □

Example 15. For equational specifications this is just Birkhoff's theorem [3].

For ordering specifications, the theorem states the correspondence between validity w.r.t. Zantema's quasi-models and derivability in Meseguer's rewriting logic (using their own terminology), a result originally due to [6].

4 Logicality

We present a uniform method to define convertibility relations for sub-equational logics (Subsection 4.1) and show their `logicality` [4] (Subsection 4.2), i.e. show that convertibility coincides with derivability for sub-equational specifications.

4.1 Convertibility

Definition 11. *Let $\mathcal{S}$ be a sub-equational specification with modes of inference L. Its* sub-convertibility *relation $\mathcal{S}(\rightarrow)$ is obtained by starting with the empty relation and closing under the inference rule ℓ of sub-equational logic if $\ell \in L$, in the order:* (emb), (comp), (ref), (sym), (trans).

Let $\mathcal{S}$ be a sub-equational specification. Of course, in case of a rewriting specification, having {(emb), (comp)} as modes of inference, $\mathcal{S}(\rightarrow)$ is just the rewrite (step) relation generated by the rules. Other examples are:

Example 16. 1. For an equational specification $\mathcal{S}(\rightarrow)$ is convertibility $\leftrightarrow^*_{\mathcal{S}}$.
2. For an ordering specification $\mathcal{S}(\rightarrow)$ is rewritability/reachability $\rightarrow^*_{\mathcal{S}}$.
3. For a positive ordering specification $\mathcal{S}(\rightarrow)$ is positive reachability $\rightarrow^+_{\mathcal{S}}$.

Further examples one could think of are e.g. head steps ({(emb)}) for modelling process calculi, *Identity* ({(ref)}) then $\mathcal{S}(\rightarrow)$ is just syntactic identity, or *Empty* ($\emptyset$) for which $\mathcal{S}(\rightarrow)$ is the empty relation.

4.2 Closure

We prove that derivability coincides with convertibility for a given sub-equational specification. As convertibility is defined as a special case of derivability, i.e. by applying the inference rules in the order as given in Definition 11, it is clearly contained in it. To show the other inclusion it suffices to prove that closing under an inference mode preserves closure under inference modes earlier in the order, since then the generated relation must coincide with derivability as the latter is the *least* relation closed under each inference mode. We illustrate this by means of an example.

Example 17. Suppose the relation R is compatible and we take its symmetric closure yielding $R \cup R^{-1}$. We must show that compatibility is preserved. That is, we must prove that $f(s_1,\ldots,s_n)\ R \cup R^{-1}\ f(t_1,\ldots,t_n)$ holds, under the assumption $s_1,\ldots,s_n =[R \cup R^{-1}]\ t_1,\ldots,t_n$. We distinguish cases according to whether compatibility is due to R or R^{-1} holding between two premisses.

- If compatibility is due to R, then the result follows by (comp) for R.
- If the assumption is due to R^{-1}, then $t_1,\ldots,t_n =[R]\ s_1,\ldots,s_n$, hence by (comp) $f(t_1,\ldots,t_n)\ R\ f(s_1,\ldots,s_n)$, hence by (sym) $f(s_1,\ldots,s_n)\ R \cup R^{-1}\ f(t_1,\ldots,t_n)$.

Checking preservation for all other combinations is as easy.

Proposition 2. *Closing relations in the order of Definition 11 preserves the properties/inference rules earlier in the order.*

Proof. First, note that all operations are monotonic in the sense that they may generate new conclusions, but preserve all existing ones. As the (emb)and (ref)inference rules have empty premisses, monotonocity explains the corresponding rows in the following table, which displays vertically the property which is to be preserved under closing with respect to the horizontally indicated inference mode.

	(emb)	(comp)	(ref)	(sym)	(trans)
(emb)	X	mon	mon	mon	mon
(comp)	X	X	(ref)	(sym)	(trans)
(ref)	X	X	X	mon	mon
(sym)	X	X	X	X	(trans)
(trans)	X	X	X	X	X

No closures are taken after (trans), which explains the last row. Preservation in the (comp)- and (sym)-rows follows by easy structural manipulations, using the inference rule given in the table in the end. For instance, the proof that (comp) is preserved under (sym) employs (sym) as final rule, as shown in Example 17. The other entries are dealt with an analogous way. □

Remark 2. Alternatively, one could permute any two consecutive inference rule in a derivation which are in the 'wrong' order. One easily shows that permutation is always possible, that the process terminates (use e.g. recursive path orders), and that the resulting derivation (the normal form) is a conversion.

Theorem 2 (Logicality). $\vdash s\;\mathcal{S}\;t$ *iff* $s\;\mathcal{S}(\rightarrow)\;t$, *for specifications* $\mathcal{S}$.

Proof. ($\Rightarrow$) It suffices to verify that the term algebra $\mathcal{T}(\Sigma, X)$ with relation $\mathcal{S}(\rightarrow)$ constitutes a relational model. It follows directly from Proposition 2.
($\Leftarrow$) Trivial, since $\mathcal{S}(\rightarrow)$ is constructed by successively closing under the inference rules which are also part of the sub-equational specification $\mathcal{S}$. □

As a final application combining the Birkhoff and Logicality theorems consider the following result due to Zantema [1, Theorem 6.2.2]:

Theorem 3. *A TRS is terminating if and only if it admits a compatible well-founded monotone algebra.*

Proof. View the TRS as a positive ordering specifications. From the above we then have that $\rightarrow^+$ is sound and complete w.r.t. positively ordered models. If the order is required to be well-founded such models coincide with compatible well-founded monotone algebras. Hence the if-direction follows from the existence of such a model by soundness. The only-if direction follows by the relational term model construction, and the observation made above that quotienting a terminating relation does nothing.

Note that for this example to work it was necessary to drop (ref), i.e. one could work with neither equational nor rewriting logic. Also, building-in transitivity in the order of a monotone algebra would not have been necessary; working with a terminating relation instead would be fine as well. More generally, often a *big step* semantics can easily be replaced by a *small step* semantics without problems.

5 Conclusion

We have given a uniform presentation of Birkhoff-style sound- and completeness results for various sub-equational logics. Moreover, we have given a uniform proof of logicality of rewriting for each of them. Although the results are not very surprising, we have not seen such a uniform presentation before. Moreover we do think the resulting presentation is elegant and the analysis required and performed is useful.

Acknowledgments Thanks are due to ETL, Tsukuba and the Institute of Information Sciences and Electronics, of Tsububa University for travel support, to the referees for constructive criticism, and to J. Joosten, A. Visser, and the attendants of the ETL-rewrite day, for feedback.

References

[1] Terese: Term Rewriting Systems. Volume 55 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2003)

[2] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
[3] Birkhoff, G.: On the structure of abstract algebras. Proceedings of the Cambridge Philosophical Society **31** (1935) 433–454
[4] Yamada, T., Avenhaus, J., Loría-Sáenz, C., Middeldorp, A.: Logicality of conditional rewrite systems. TCS **236** (2000) 209–232
[5] Mitchell, J.C.: Foundations for Programming Languages. Foundations of Computing series. The MIT Press (1996)
[6] Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. TCS **96** (1992) 73–155
[7] Zantema, H.: Termination of term rewriting: interpretation and type elimination. JSC **17** (1994) 23–50

Relaxing the Value Restriction

Jacques Garrigue

Research Institute for Mathematical Sciences,
Kyoto University, Sakyo-ku, Kyoto 606-8502
garrigue@kurims.kyoto-u.ac.jp

Abstract. Restricting polymorphism to values is now the standard way to obtain soundness in ML-like programming languages with imperative features. While this solution has undeniable advantages over previous approaches, it forbids polymorphism in many cases where it would be sound. We use a subtyping based approach to recover part of this lost polymorphism, without changing the type algebra itself, and this has significant applications.

1 Introduction

Restricting polymorphism to values, as Wright suggested [1], is now the standard way to obtain soundness in ML-like programming languages with imperative features. The long version of this paper explains in detail how this conclusion was reached [2]. This solution's main advantages are its utter simplicity (only the generalization rule is changed from the original Hindley-Milner type system), and the fact it avoids distinguishing between applicative and imperative type variables, giving identical signatures to pure and imperative functions. This property is sometimes described as *implementation abstraction.*

Of course, this solution is sometimes more restrictive than previous ones. In particular, all previous solutions, both those based on weak type variables [3,4,5,6] and those based on more refined approaches using effects [7] or closure typing [8,9], were conservative: typing was only restricted for functions actually constructing imperative data. The value restriction is not conservative: by assuming that all functions may be imperative, lots of polymorphism is lost. However, this extra polymorphism appeared to be of limited practical use, and experiments have shown that the changes needed to adapt ML programs typechecked using stronger type systems to the value only polymorphism type system were negligible.

Almost ten years after the feat, it might be useful to check whether this is still true. Programs written ten years ago were not handicapped by the value restriction, but what about programs we write now, or programs we will write in the future?

In his paper, Wright considers 3 cases of let-bindings where the value restriction causes a loss of polymorphism.

1. Expressions that never return. They do not appear to be really a problem, but he remarks that in the specific case of $\forall\alpha.\alpha$, it would be sound to keep the stronger type.

© Springer-Verlag Berlin Heidelberg 2004

2. Expressions that compute polymorphic procedures.
   This amounts to a partial application. Analysis of existing code showed that their evaluation was almost always purely applicative, and as a result one could recover the polymorphism through eta-expansion of the whole expression, except when the returned procedure is itself embedded in a data structure.
3. Expressions that return polymorphic data structures. A typical example is an expression returning always the empty list. It should be given the polymorphic type α `list`, but this is not possible under the value restriction if the expression has to be evaluated.

Of these 3 cases, the last one, together with the data-structure case of the second one, are most problematic: there is no workaround to recover the lost polymorphism, short of recomputing the data structure at each use. This seemed to be a minor problem, because existing code made little use of this kind of polymorphism inside a data structure. However we can think of a number of cases where this polymorphism is expected, sometimes as a consequence of extensions to the type system.

1. Constructor and accessor functions. While algebraic datatype constructors and pattern matching are handled specially by the type system, and can be given a polymorphic type, as soon as we define functions for construction or access, the polymorphism is lost. The consequence is particularly bad for abstract datatypes and objects [10], as one can only construct them through functions, meaning that they can never hold polymorphic values.
2. Polymorphic variants [11]. By nature, a polymorphic variant is a polymorphic data structure, which can be seen as a member of many different variant types. If it is returned by a function, or contains a computation in its argument, it looses this polymorphism.
3. Semi-explicit polymorphism [12]. This mechanism allows to keep principality of type-checking in the presence of first-class polymorphism. This is done through adding type variable markers to first-class polymorphic types, and checking their polymorphism. Unfortunately, value restriction looses this polymorphism. A workaround did exist, but the resulting type system was only "weakly" principal.

We will review these cases, and show how the value restriction can be relaxed a little, just enough for many of these problems to be leveled. As a result, we propose a new type system for ML, with *relaxed value restriction,* that is strictly more expressive (it types more programs) than ML with the usual value restriction.

The starting point is very similar to the original observation about $\forall\alpha.\alpha$: in some cases, polymorphic types are too generic to contain any value. As such they can only describe empty collections, and it is sound to allow their generalization.

Our basic idea is to use the structural rules of subtyping to recover this polymorphism: by subsumption, if a type appears at a covariant position inside the type of a value, it shall be safe to replace it with any of its superrypes. From a set-theoretic point of view, if this type is not inhabited, then it is a subtype of all other types (they all contain the empty set). If it can be replaced by any type, then we can make it a polymorphic variable. This basically means that, even for side-effecting expressions, it is sound to generalize non-contravariant type variables.

Unfortunately, this model-based reasoning cannot be translated into a direct proof: we are aware of no set theoretic model of ML extended with references. Neither can we use a direct syntactic proof, as our system, while being sound, does not enjoy the subject reduction. Nonetheless this intuition will lead us to an indirect proof, by translation into a stronger type system with subtyping.

This paper is organized as follows. We first describe our approach in more detail, and apply it to simple cases. Then we show how it helps solving the problems described above. In section 4 we answer some concerns. In section 5 we formalize our language and type system, and prove its soundness using semantic types in section 6, before concluding. More details and extra proofs can be found in the accompanying technical report [2].

2 Polymorphism from Subtyping

Before entering into the details of our method, let us define our intent.

We follow the value restriction, and keep its principles: simplicity and abstraction. That is, we do not distinguish at the syntactic level between *applicative* and *imperative* type variables; neither do we introduce different points of quantification, as in rank-1 polymorphism [13]. All type variables in any function type are to be seen as imperative: by default, they become non-generalizable in the let-binding of a non-value (*i.e.* a term containing a function application), on a purely syntactical criterion.

However we can analyze the semantic properties of types, independently of the implementation. By distinguishing between covariant and contravariant variables in types we are able to partially lift this restriction when generalizing: as before, variables with contravariant occurrences in the type of an expansive expression cannot be generalized, but variables with only covariant occurrences can be generalized.

The argument goes as follows. We introduce a new type constructor, `zero` , which is kept empty. We choose to instantiate all non-contravariant variables in let-bound expressions by `zero`. In a next step we coerce the type of the let-bound variable to a type where all `zero` 's are replaced by (independent) fresh type variables. Since the coercion of a variable is a value, in this step we are no longer limited by the value restriction, and these type variables can be generalized.

To make explanations clear, we will present our first two examples following the same pattern: first give the non-generalizable type scheme as by the value restriction (typed by Objective Caml 3.06 [14]), then obtain a generalized version by explicit subtyping. However, as explained in the introduction, our real intent is to provide a replacement for the usual value restriction, so we will only give the generalized version —as Objective Caml 3.07 does—, in subsequent examples. Here is our first example.

```
let l =
  let r = ref [] in !r
val l : '_a list = []
```

The type variable `'_a` is not generalized: it will be instantiated when used, and fixed afterwards. This basically means that `l` is now of a fixed type, and cannot be used in polymorphic contexts anymore.

$$V^-(\alpha) = \emptyset \qquad \begin{array}{l} V^-(\tau\,\mathtt{ref}) = FTV(\tau) \\ V^-(\tau\,\mathtt{list}) = V^-(\tau) \end{array} \qquad \begin{array}{l} V^-(\tau_1 \to \tau_2) = FTV(\tau_1) \cup V^-(\tau_2) \\ V^-(\tau_1 \times \tau_2) = V^-(\tau_1) \cup V^-(\tau_2) \end{array}$$

Fig. 1. Dangerous variables

Our idea is to recover polymorphism through subtyping.

```
let l = (l : zero list :> 'a list)
val l : 'a list = []
```

A coercion $(e : \tau_1 :> \tau_2)$ makes sure that e has type τ_1, and that τ_1 is a subtype of τ_2. Then, it can safely be seen as having type τ_2. Since `l` is a value, and the coercion of a value is also a value, this is a value binding, and the new `'a` in the type of the coerced term can be generalized.

Why is it sound? Since we assigned an empty list to `r`, and returned its contents without modification, `l` can only be the empty list; as such it can safely be assigned a polymorphic type.

Comparing with conservative type systems, Leroy's closure-based typing [8] would indeed infer the same polymorphic typing, but Tofte's imperative type variables [3] would not: since the result is not a closure, with Leroy's approach the fact `[]` went through a reference cell doesn't matter; however, Tofte's type system would force its type to be imperative, precluding any further generalization when used inside a non-value binding.

The power of this approach is even more apparent with function types.

```
let f =
  let r = ref [] in fun () -> !r
val f : unit -> '_a list
```

which we can coerce again

```
let f = (f : unit -> zero list :> unit -> 'a list)
val f : unit -> 'a list
```

This result may look more surprising, as actually `r` is kept in the closure of `f`. But since there is no way to modify its contents, `f` can only return the empty list. This time, even Leroy's closure typing and Talpin&Jouvelot's effect typing [7] cannot meet the mark.

This reasoning holds as long as a variable does not appear in a contravariant position. Yet, for type inference reasons we explain in section 5, we define a set of dangerous variables (figure 1) including all variables appearing on the left of an arrow, which is more restrictive than simple covariance. In a non-value binding, we will generalize all local variables except those in $V^-(\tau)$, assuming the type before generalization is τ. This definition is less general than subtyping, as a covariant type variable with multiple occurences will be kept shared. For instance, subtyping would allow `('_a * '_a) list` to be coerced to `('a * 'b) list`, but type inference will only give the less general `('a * 'a) list`.

Of course, our approach cannot recover all the polymorphism lost by the value restriction. Consider for instance the partial application of `map` to the identity function.

```
let map_id = List.map (fun x -> x)
val map_id : '_a list -> '_a list
```

Since `'_a` also appears in a contravariant position, there is no way this partial application can be made polymorphic. Like with the strict value restriction, we would have to eta-expand to obtain a polymorphic type.

However, the relaxed value restriction becomes useful if we fully apply `map`, a case where eta-expansion cannot be used.

```
let l = List.map (fun id -> id) []
val l : 'a list
```

Note that all the examples presented in this section cannot be handled by rank-1 polymorphism. This is not necessarily the case for examples in the next section, but this suggests that improvements by both methods are largely orthogonal.

While our improvements are always conceptually related to the notion of empty container, we will see in the following examples that it can show up in many flavors, and that in some cases we are talking about concrete values, rather than empty ones.

3 Application Examples

In this section, we give examples of the different problems described in the introduction, and show how we improve their typings.

3.1 Constructor and Accessor Functions

In ML, we can construct values with data constructors and extract them with pattern matching.

```
let empty2 = ([],[])
val empty2 : 'a list * 'b list = ([], [])
let (_,l2) = empty2
val l2 : 'a list = []
```

As you can see here, since neither operations use functions, the value restriction does not come in the way, and we obtain a polymorphic result. However, if we use a function as accessor, we loose this polymorphism.

```
let l2 = snd empty2
val l2 : '_a list = []
```

Moreover, if we define custom constructors, then polymorphism is lost in the original data itself. Here `pair` assists in building a Lisp-like representation of tuples.

```
let pair x y = (x, (y, ()))
val pair : 'a -> 'b -> 'a * ('b * unit)
let empty2' = pair [] []
val empty2' : '_a list * ('_b list * unit) = (..)
```

The classical workaround to obtain a polymorphic type involves eta-expansion, which means code changes, extra computation, and is incompatible with side-effects, for instance if we were to count the number of cons-cells created.

If the parameters to the constructor have covariant types, then the relaxed value restriction solves all these problems.

```
let l2 = snd empty2
val l2 : 'a list = []
let empty2' = pair [] []
val empty2' : 'a list * ('b list * unit) = (..)
```

This extra polymorphism allows one to share more values throughout a program.

3.2 Abstract Datatypes

This problem is made more acute by abstraction. Suppose we want to define an abstract datatype for bounded length lists. This can be done with the following signature:

```
module type BLIST = sig
  type +'a t
  val empty : int -> 'a t
  val cons : 'a -> 'a t -> 'a t
  val list : 'a t -> 'a list
end
module Blist : BLIST = struct
  type 'a t = int * 'a list
  let empty n = (n, [])
  let cons a (n, l) =
    if n > 0 then (n-1, a::l) else raise (Failure "Blist.cons")
  let list (n, l) = l
end
```

The + in `type +'a t` is a variance annotation, and is available in Objective Caml since version 3.01. It means that `'a` appears only in covariant positions in the definition of `t`. This additional information was already used for explicit subtyping coercions (between types including objects or variants), but with our approach we can also use it to automatically extract more polymorphism.

The interesting question is what happens when we use `empty`. Using the value restriction, one would obtain:

```
let empty5 = Blist.empty 5
val empty5 : '_a Blist.t = <abstract>
```

Since the type variable is monomorphic, we cannot reuse this `empty5` as *the* empty 5-bounded list; we have to create a new empty list for each different element type. And this time, we cannot get the polymorphism by building the value directly from data constructors, as abstraction has hidden the type's structure.

Just as for the previous example, relaxed valued restriction solves the problem: since *'_a* is not dangerous in *'_a Blist.t*, we shall be able to generalize it.

```
val empty5 : 'a Blist.t = <abstract>
```

With the relaxed value restriction, abstract constructors can be polymorphic as long as their type variables are covariant inside the abstract type.

3.3 Object Constructors

As one would expect from its name, Objective Caml sports object-oriented features. Programmers are often tempted by using classes in place of algebraic datatypes. A classical example is the definition of lists.

```
class type ['a] list = object
  method empty : bool
  method hd : 'a
  method tl : 'a list
end
class ['a] nil : ['a] list = ...
class ['a] cons a b : ['a] list = ...
```

This looks all nice, until you realize that you cannot create a polymorphic empty list: an object constructor is seen by the type system as a function. Again, as `'a` is covariant in `'a list`, it is generalizable, and the relaxed value restriction allows a polymorphic type.

```
let nil : 'a list = new nil
val nil : 'a list = <obj>
```

3.4 Polymorphic Variants

Polymorphic variants [11,15] are another specific feature of Objective Caml. Their design itself contradicts the assumption that polymorphic data structures are rare in ML programs: by definition a polymorphic variant can belong to any type that includes its tag.

```
let one = `Int 1
val one : [> `Int of int] = `Int 1
let two = `Int (1+1)
val two : _[> `Int of int] = `Int 2
```

Again the value restriction gets in our way: it's enough that the argument is not a value to make the variant constructor monomorphic (as shown by the "_" in front of the type). And of course, any variant returned by a function will be given a monomorphic type. This means that in all previous examples, you can replace the empty list by any polymorphic variant, and the same problem will appear.

Again, we can use our coercion principle[1]:

```
let two = (two : [`Int of int] :> [> `Int of int])
val two : [> `Int of int] = `Int 2
```

This makes using variants in multiple contexts much easier. Polymorphic variants profit considerably from this improvement. One would like to see them simply as the dual of polymorphic records (or objects), but the value restriction has broken the duality.

[1] zero amounts here to an empty variant type, and if we show the internal row extension variables the coercion would be `(two : [`Int of int | zero] :> [`Int of int | 'a])`, meaning that in one we case we allow no other constructor, and in the other case we allow any other constructor.

For polymorphic records, it is usually enough to have polymorphism of functions that accept a record, but for polymorphic variants the dual would be polymorphism of variants themselves, including results of computations, which the value restriction did not allow. While Objective Caml allowed polymorphism of functions that accept a variant, there were still many cases where one had to use explicit subtyping, as the same value could not be used in different contexts by polymorphism alone. For instance consider the following program:

```
val all_results :
  [ `Bool of bool | `Float of float | `Int of int] list ref
val num_results : [ `Float of float | `Int of int] list ref
let div x y =
  if x mod y = 0 then `Int(x/y) else `Float(float x/.float y)
val div : int -> int -> [> `Float of float | `Int of int]
let comp x y =
  let z = div x y in
  all_results := z :: !all_results;
  num_results := z :: !num_results
val comp : int -> int -> unit
```

Since `all_results` and `num_results` are toplevel references, their types must be ground. With the strict value restriction, `z` would be given a monomorphic type, which would have to be equal to the types of both references. Since the references have different types, this is impossible. With the relaxed value restriction, `z` is given a polymorphic type, and distinct instances can be equal to the two reference types.

3.5 Semi-explicit Polymorphism

Since version 3.05, Objective Caml also includes an implementation of semi-explicit polymorphism [12], which allows the definition of polymorphic methods in objects.

The basic idea of semi-explicit polymorphism is to allow universal quantification anywhere in types (not only in the prefix), but to restrict instantiation of these variables to cases where the first-class polymorphism is *known* at the instantiation point. To obtain a principal notion of *knowledge,* types containing quantifiers are marked by type variables (which are only used as *markers*), and a quantified type can only be instantiated when its marker variable is generalizable. Explicit type annotations can be used to force markers to be polymorphic.

We will not explain here in detail how this system works, but the base line is that inferred polymorphism can be used to enforce principality. While this idea works very well with the original Hindley-Milner type system, problems appear with the value restriction.

We demonstrate here Objective Caml's behavior. The marker variable ϵ on the type $\texttt{poly}^{\epsilon}$ is hidden in the surface language.

```
class poly : object method id : 'a. 'a -> 'a end
let f (x : poly) = (x#id 1, x#id true)
val f : poly -> int * bool = <fun>
let h () = let x = new poly in (x#id 1, x#id true)
val h : unit -> int * bool = <fun>
```

f is a valid use of polymorphism: the annotation is on the binding of x and can be propagated to all its uses, *i.e.* the type of x is $\forall\epsilon.\mathtt{poly}^{\epsilon}$. But h would not be accepted under the strict value restriction, because new poly is not a value, so that the type $\mathtt{poly}^{\epsilon}$ of x is not generalizable. Since refusing cases like h would greatly reduce the interest of type inference, it was actually accepted, arguing that markers have no impact on soundness. A system allowing this is formalized in [12], yet it replaces full blown principality by a notion of principality among maximal derivations, which is a weaker property.

By using our scheme of generalizing type variables that do not appear in dangerous positions, we can recover full principality, with all its theoretical advantages, and accept h “officially”.

Note also that since these markers may appear in types that otherwise have no free type variables, this boosts the number of data structures containing polymorphic (marker) variables. That is, semi-explicit polymorphism completely invalidates the assumption that polymorphic values that are not functions are rare and not essential to ML programming.

4 Concerns

This section addresses some natural concerns about the relaxed value restriction.

4.1 Typing Power and Usefulness

A first question is how powerful the relaxed value restriction is, compared to the value restriction and other known systems, and whether its improvements are genuinely useful or not. If we considered only benchmarks proposed in the literature [7,8], we would come to the conclusion that the relaxed value restriction adds no power: its results exactly matches those of the strict value restriction. This is because all examples in the literature are only concerned with polymorphic procedures, not polymorphic data.

In the previous section we have given a fair number of examples handling polymorphic data. They demonstrate the additional power of our system. Compared with system predating the value restriction, we are in general less powerful, with some exceptions as shown in section 2. However, in practice implementation abstraction matters more than pure typing power, and on this side we keep the good properties of the value restriction.

Our examples with constructor functions and abstract datatypes were expressible in systems predating the value restriction, and are refused by the strict value restriction. This makes one wonder why this didn’t cause more problems during the transition. These idioms were apparently rarely used then. However, the author believes he is not alone in having experienced exactly those problems on newly written code. And there have been explicit reports of polymorphism problems with objects and variants, justifying the need for such an improvement.

4.2 Abstraction

While we claim that our scheme is not breaking implementation abstraction, one may remark that we require variance annotations for abstract datatype definitions. Aren't these annotations breaking abstraction?

Clearly, specifying a variance reduces the generality of an interface, and as such it is reducing its abstraction degree. However we claim that this does not mean that we are breaking implementation abstraction. We give here a concrete example, defining covariant vectors on top of nonvariant mutable arrays.

```
type +'a vector = {get: int -> 'a; length: int}
let make len f =
  let arr = if len = 0 then [||] else Array.create len (f 0) in
  for i = 1 to len-1 do arr.(i) <- f i done;
  {get=Array.get arr; length=len}
val make : int -> (int -> 'a) -> 'a vector
let map f vect = make vect.length (fun i -> f (vect.get i))
val map : ('a -> 'b) -> 'a vector -> 'b vector
```

What this example demonstrates, is that variance is not limited by the implementation. By changing the superficial definition, while keeping the same internal implementation, we may improve the variance of a datatype. This situation is to be compared with imperative type variables, or equality type variables, whose specificity must be propagated through any definition they are used in, making it impossible to abstract from the implementation.

To be fully honest, there are cases where an overspecified variance results in making some implementations impossible. But this should be seen as a problem of bad design, and the above example gives a natural criterion for proper variance of an abstract datatype: this should at most be the variance of the minimal set of operations which cannot be defined without access to the implementation.

4.3 Ease of Use

Does the introduction of variance make the language harder to use? There are actually two problems: understanding the new typing rule, and having to write variance annotations for abstract datatypes.

Seeing that the value restriction itself is rather hard to grasp —notwithstanding the simplicity of its definition—, one might argue that any improvement of polymorphism (when it does not involve changes in the type algebra itself) is good, as it is going to avoid some non-intuitive type errors. Moreover, once you understand the typing algorithm, the relaxed value restriction introduces no leap in complexity.

More disturbing may be the need for variance annotations. For Objective Caml, they were already there, as the language allows explicit subtyping. So we are just exploiting an existing feature. But even if it were to be newly added, keep in mind that explicit annotations are only needed for abstract datatype definitions, and that there is a good semantic criterion as to what they should be. Of course this information is only optional: at worst, we are still as powerful as the value restriction.

4.4 Compilation

A last concern is with compilation, in particular for compilers using type information during compilation or at runtime. These compilers often involve a translation to an explicitly typed second-order lambda-calculus, which does not seem to be a good target for our system since, as we will see in the next sections, our type soundness seems to require subtyping.

A first remark is that the problem lies not so much in our approach as in the inadequation between polymorphic data structures and second-order lambda-calculus. While there can be no value whose type is a covariant variable inside the data structure, second-order lambda-calculus would have us pass its (useless) type around.

The answer is simple enough: we just have to extend the target type system with the needed subtyping, knowing that this will not impact later stages of compilation as there are no values of type `zero` anyway. To gain full profit of our remark, we may even replace all purely covariant type variables with `zero` —in value bindings too—, so as to minimize the type information passed around.

While `zero` is not a problem, compilation is one of the reasons we have stopped short of exploiting the dual observation: that assuming a "type of all values" `top`, the monomorphic type variables that appear only in contravariant positions are generalizable too. This would have had an extra advantage: this should alleviate the principality problem, which had us restrict generalizability to type variables of rank 0. Only variables that appear both in covariant and contravariant position would not be generalizable. However, the existence of `top` would require all values to be represented in a uniform way. This is just what type-passing implementations want to avoid. Actually, even Objective Caml, which has only a very conservative use of type information, does not satisfy this property[2].

5 Formalization and Type System

In this section we fully formalize our language, and propose a type system where the extra polymorphism described in previous examples is recovered automatically (without the need for explicit coercions). Yet this type system, which we call the *relaxed value restriction,* enjoys the principal type property.

We base ourselves on Wright and Felleisen's formalization of Reference ML [16]. For our results to be meaningful, we need to handle more varied data, so we also add pairs and lists, as they do not incur any difficulty in typing.

Expressions distinguish between values and non-values. The store is introduced by the $\rho\theta.e$ binder and is handled explicitly. Two kinds of contexts are defined for reduction rules: R-contexts, used in store operations, and E-contexts, in evaluation.

[2] The function `Obj.repr` can be seen as a coercion to `top`(*aka* `Obj.t`), but it is unsafe.

```
let l = Array.create 2 (Obj.repr 1.0)
val l : Obj.t array = [|<abstr>; <abstr>|]
l.(1) <- Obj.repr 1
Segmentation fault
```

In one sentence: arrays of float values have a special representation, and operations on arrays are not semantically correct when float and int values are mixed —which is of course impossible using the existing type system and safe operations.

$e ::= v \mid e_1\, e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \rho\theta.e$
$v ::= x \mid \mathsf{Y} \mid \lambda x.e \mid \mathsf{ref} \mid\ ! \mid := \mid := v \mid (v,v) \mid \pi_1 \mid \pi_2 \mid \mathsf{nil} \mid \mathsf{cons}\ v \mid \mathsf{uncons}\ v\ v$
$\theta ::= \{\langle x, v\rangle\}^*$
$R ::= [\,] \mid R\, e \mid v\, R \mid \mathsf{let}\ x = R\ \mathsf{in}\ e$
$E ::= [\,] \mid E\ e \mid v\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid \rho\theta.E$

As in Reference ML, both := and := v are values, reflecting the fact := can only be reduced when given two arguments.

Reduction rules are given in figure 2. They are those of Reference ML, with a few innocuous additions. We define one-step reduction as $E[e] \longrightarrow E[e']$ whenever $e \longrightarrow e'$, and multi-step reduction as $e_1 \overset{*}{\longrightarrow} e_n$ whenever $e_1 \longrightarrow e_2 \ldots \longrightarrow e_n$. Reduction does not produce badly-formed expressions.

Lemma 1. *If* e *is a well-formed expression (i.e. no non-value appears at a value position), and* $e \longrightarrow e'$*, then* e' *is well-formed.*

Types are the usual monotypes and polytypes.

$$\tau ::= \alpha \mid \tau\ \mathtt{ref} \mid \tau \times \tau \mid \tau\ \mathtt{list}$$
$$\sigma ::= \tau \mid \forall\bar{\alpha}.\tau$$

An instantiation order $\succ$ is defined on polytypes by $\forall\bar{\alpha}.\tau \succ \forall\bar{\beta}.\tau'$ iff $\bar{\beta} \cap FTV(\forall\bar{\alpha}.\tau) = \emptyset$ and there is a vector $\bar{\tau}$ of monotypes such that $[\bar{\tau}/\bar{\alpha}]\tau = \tau'$.

We type this language using typing rules in figure 3. Those rules are again taken from Reference ML, assuming all type variables to be imperative (which is equivalent to applying the value restriction, *cf* [1]page 6). The only exception is the LET$_e$ rule, which generalizes some variables. In the value case, $Close(\tau_1, \Gamma) = \forall FTV(\tau_1)\backslash FTV(\Gamma).\tau_1$ as usual, but in the non-value case we still generalize safe variables: $CovClose(\tau_1, \Gamma) = \forall FTV(\tau_1) \setminus V^-(\tau_1) \setminus FTV(\Gamma).\tau_1$, with V^- the set of dangerous variables defined in figure 1. The definition of V^- captures more variables than the usual definition of contravariant occurrences. We deem dangerous all occurrences appearing in a contravariant branch of a type. While this is not necessary to ensure type soundness, we need it to keep principality of type inference. For instance, consider the following function.

$$\mathsf{let}\ f = \mathsf{let}\ r = \mathsf{ref\ nil\ in}\ \lambda k.\mathsf{Y}\ (\lambda f.f)\ !r$$

As the type of Y $(\lambda f.f)$ is $\forall\alpha\beta.\alpha \to \beta$, we expect the principal type of f to be $\forall\beta.\gamma \to \beta$, with γ a non generalizable variable. However, if we were to generalize covariant variables at ranks higher than 0, then $\forall\beta\delta.(\delta \to \gamma) \to \beta$ would be another acceptable type for f, and neither of the two is an instance of the other. *i.e.* we would have lost principality.

As we explained in section 3.5, rule LET$_e$ does not unshare covariant type variables, as it would be sound to do, but only allows for more type variables to be generalized. Unsharing variables would break even the partial subject reduction we define lower.

$$\begin{array}{llll}
(\beta_v) & (\lambda x.e)\, v \rightarrow e[v/x] & (\pi_1) & \pi_1\,(v_1,v_2) \rightarrow v_1 \\
(let) & \mathsf{let}\; x = v\; \mathsf{in}\; e \rightarrow e[v/x] & (\pi_2) & \pi_2\,(v_1,v_2) \rightarrow v_2 \\
(\mathsf{Y}) & \mathsf{Y}\, v \rightarrow v\, (\lambda x.\mathsf{Y}\, v\, x) & (un_1) & \mathsf{uncons}\; v_1\; v_2\; \mathsf{nil} \rightarrow v_1\; \mathsf{nil} \\
(ref) & \mathsf{ref}\; v \rightarrow \rho\langle x,v\rangle.x & (un_2) & \mathsf{uncons}\; v_1\; v_2\; (\mathsf{cons}\; v) \rightarrow v_2\; v \\
(deref) & \rho\theta\langle x,v\rangle.R[!\,x] \rightarrow \rho\theta\langle x,v\rangle.R[v] & & \\
(assign) & \rho\theta\langle x,v_1\rangle.R[:=x\; v_2] \rightarrow \rho\theta\langle x,v_2\rangle.R[v_2] & & \\
(\rho_{merge}) & \rho\theta_1.\rho\theta_2.e \rightarrow \rho\theta_1\theta_2.e & & \\
(\rho_{lift}) & R[\rho\theta.e] \rightarrow \rho\theta.R[e] & & \text{if } R \neq [\,] \\
(\rho_{drop}) & \rho\theta.e \rightarrow e & & \text{if } dom(\theta) \cap FV(e) = \emptyset
\end{array}$$

Fig. 2. Reduction rules

$$\textsc{Var}\quad \frac{\Gamma(x) \succ \tau}{\Gamma \vdash x : \tau} \qquad \textsc{App}\quad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1} \qquad \textsc{Abs}\quad \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$$

$$\textsc{Let}_v\quad \frac{\Gamma \vdash v : \tau_1 \quad \Gamma[x \mapsto Close(\tau_1,\Gamma)] \vdash e : \tau_2}{\Gamma \vdash \mathsf{let}\; x = v\; \mathsf{in}\; e : \tau_2} \qquad \textsc{Pair}\quad \frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash (v_1,v_2) : \tau_1 \times \tau_2}$$

$$\textsc{Let}_e\quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto CovClose(\tau_1,\Gamma)] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\; x = e_1\; \mathsf{in}\; e_2 : \tau_2} \qquad \textsc{Cons}\quad \frac{\Gamma \vdash v : \tau \times \tau\;\mathtt{list}}{\Gamma \vdash \mathsf{cons}(v) : \tau\;\mathtt{list}}$$

$$\textsc{Rho}\quad \frac{\Gamma[x_j \mapsto \tau_j\;\mathtt{ref}]_1^n \vdash e : \tau \quad \Gamma[x_j \mapsto \tau_j\;\mathtt{ref}]_1^n \vdash v_i : \tau_i \;\; (1 \leq i \leq n)}{\Gamma \vdash \rho\langle x_1,v_1\rangle \ldots \langle x_n,v_n\rangle.e : \tau}$$

$\textsc{Axioms}$

$$\begin{array}{lll}
\Gamma \vdash \mathsf{Y} : ((\tau_1 \rightarrow \tau_2) \rightarrow \tau_1 \rightarrow \tau_2) \rightarrow \tau_1 \rightarrow \tau_2 & & \\
\Gamma \vdash \mathsf{ref} : \tau \rightarrow \tau\;\mathtt{ref} & \Gamma \vdash\; ! : \tau\;\mathtt{ref} \rightarrow \tau & \Gamma \vdash\; := : \tau\;\mathtt{ref} \rightarrow \tau \rightarrow \tau \\
\Gamma \vdash \pi_1 : \tau_1 \times \tau_2 \rightarrow \tau_1 & \Gamma \vdash \pi_2 : \tau_1 \times \tau_2 \rightarrow \tau_2 & \Gamma \vdash \mathsf{nil} : \tau\;\mathtt{list} \\
\multicolumn{3}{l}{\Gamma \vdash \mathsf{uncons} : (\tau_1\;\mathtt{list} \rightarrow \tau_2) \rightarrow (\tau_1 \times \tau_1\;\mathtt{list} \rightarrow \tau_2) \rightarrow \tau_1\;\mathtt{list} \rightarrow \tau_2}
\end{array}$$

Fig. 3. Typing rules

$$\textsc{b-Sub}\quad \frac{\Gamma \models e : t \quad t \leq t'}{\Gamma \models e : t'} \qquad \textsc{b-Let}_v\quad \frac{(\forall t \in s)\, \Gamma \models v : t \quad \Gamma[x \mapsto s] \models e : t'}{\Gamma \models \mathsf{let}\; x = v\; \mathsf{in}\; e : t'} \qquad \textsc{b-Abs}\quad \frac{\Gamma[x \mapsto \uparrow t_1] \models e : t_2}{\Gamma \models \lambda x.e : t_1 \rightarrow t_2}$$

$$\textsc{b-Var}\quad \frac{t \in \Gamma(x)}{\Gamma \models x : t} \qquad \textsc{b-App}\quad \frac{\Gamma \models e_1 : \tau_2 \rightarrow t_1 \quad \Gamma \models e_2 : t_2}{\Gamma \models e_1\, e_2 : t_1} \qquad \textsc{b-Let}_e\quad \frac{\Gamma \models e_1 : t_1 \quad \Gamma[x \mapsto \uparrow t_1] \models e_2 : t_2}{\Gamma \models \mathsf{let}\; x = e_1\; \mathsf{in}\; e_2 : t_2}$$

$$\textsc{b-Rho}\quad \frac{\Gamma[x_j \mapsto \uparrow(t_j\;\mathtt{ref})]_1^n \models e : t \quad \Gamma[x_j \mapsto \uparrow(t_j\;\mathtt{ref})]_1^n \models v_i : t_i \;\; (1 \leq i \leq n)}{\Gamma \models \rho\langle x_1,v_1\rangle \ldots \langle x_n,v_n\rangle.e : t}$$

Fig. 4. Typing rules for B(T)

We include the RHO typing rule for completeness, but we cannot use it to obtain full subject reduction. We can see this on the following example[3].

$$\begin{aligned}&\textsf{let}\ f = (\textsf{let}\ r = \textsf{ref nil in}\ \lambda x.!r)\ \textsf{in}\ (\textsf{cons}(\textsf{nil}, f\ \textsf{nil}), \textsf{cons}(\textsf{ref nil}, f\ \textsf{nil}))\\ \rightarrow\ &\rho\langle r, \textsf{nil}\rangle.(\textsf{cons}(\textsf{nil}, (\lambda x.!r)\ \textsf{nil}), \textsf{cons}(\textsf{ref nil}, (\lambda x.!r)\ \textsf{nil}))\end{aligned}$$

In the first line, f can be given the polymorphic type $\forall\alpha.\ \beta\ \texttt{list} \rightarrow \alpha\ \texttt{list}$, with β a non-generalized type variable. When we apply f to nil we may get any list. The type of the whole expression is $(\tau_1\ \texttt{list list} \times \tau_2\ \texttt{list ref list})$. However, after reduction, r can only be given a monomorphic type, and its two occurrences appear in incompatible type contexts.

In the absence of direct subject reduction, we must prove type soundness in an indirect way. Following our intuition, we could recover subject reduction in a stronger system, by adding a subsumption rule,

$$\frac{\Gamma \vdash e : \tau[\overline{\texttt{zero}}/\bar{\alpha}]}{\Gamma \vdash e : \tau}\ \bar{\alpha} \cap V^{-}(\tau) = \emptyset$$

Rather than doing this directly, and bearing the burden of proof, we will do this in the next section by translating our derivations into a known type system validating this rule. We believe that an appropriate form of subsumption (direct or indirect) is essential to proofs of subject reduction for type systems validating our LET$_e$ rule.

On the other hand, principality is a static property of terms, and we can prove it easily by trivially modifying the inference algorithm W, using *CovClose* in place of *Close* for non-values. This is clearly sound: this is our rule. This is also complete: *CovClose* is monotonic with respect to the instantiation order $\succ$, that is, for any type substitution S, we have $\mathit{CovClose}(\tau, \Gamma) \succ \mathit{CovClose}(S(\tau), S(\Gamma))$.

Proposition 1 (principality). *If, for a given pair (Γ, e) there is a τ_0 such that $\Gamma \vdash e : \tau_0$ is derivable, then there exists a σ such that for any τ, $\Gamma \vdash e : \tau$ iff $\sigma \succ \tau$.*

We can also verify a partial form of subject reduction, limited to non side-effecting reductions, but allowing those reductions to happen anywhere in a term. While insufficient to prove type soundness, this property is useful to reason about program transformations.

$$\begin{aligned}C ::= &\ [\,] \mid C\ e \mid e\ C \mid \textsf{let}\ x = C\ \textsf{in}\ e \mid \textsf{let}\ x = e\ \textsf{in}\ C\\ &\mid \rho\theta\langle x, C\rangle.e \mid \rho\theta.C \mid \lambda x.C \mid (C, v) \mid (v, C)\end{aligned}$$

Proposition 2 (partial subject reduction). *Non side-effecting reductions, i.e. rules (β_v),(let),(Y), (π_i), (un_i) preserve typing: for any context C, if $\Gamma \vdash C[e] : \tau$ and $e \rightarrow_f e'$, then $\Gamma \vdash C[e'] : \tau$.*

The proof can be easily transposed from any proof of subject reduction for applicative ML. We only need to verify that the substitution lemma still holds in presence of our distinction between LET$_v$ and LET$_e$.

Lemma 2 (substitution). *If $\Gamma[x \mapsto \sigma_1] \vdash e : \tau$ and $\Gamma \vdash v : \tau_1$ and $\mathit{Close}(\tau_1, \Gamma) \succ \sigma_1$, then $\Gamma \vdash e[v/x] : \tau$.*

[3] For sake of conciseness we use pairs of expressions, rather than an expanded form where pairs contain only values; and we write e_1 ; e_2 as a shorthand for let $i = e_1$ in e_2 (i fresh). This has no impact on typing.

6 Type Soundness

Rather than extending our own type system with subsumption, we will reuse one that already has the required combination of polymorphism, imperative operations, and subtyping. A good choice is Pottier's B(T) [17], as its typing rules closely match ours. B(T) was originally developed as an intermediate step in the proof of type soundness for HM(X), a constraint-based polymorphic type system [18]. B(T) is particular by its extensional approach to polymorphism: polytypes are not expressed syntactically, but as (possibly infinite) sets of ground monotypes. For us, its main advantages are its simplicity (no need to introduce constraints as in HM(X)), and the directness of the translation of typing derivations.

We give here a condensed account of the definition of B(T), which should be sufficient to understand how a typing derivation in our system can be mapped to a typing derivation in an instance of B(T).

The T in B(T) represents a universe of monotypes, equipped with a subtyping relation $\leq$, serving as parameter to the type system. Monotypes in T are denoted by t. $\rightarrow$ should be a total function from $T \times T$ into T, such that $t_1 \rightarrow t_2 \leq t'_1 \rightarrow t'_2$ implies $t'_1 \leq t_1$ and $t_2 \leq t'_2$. `ref` should be a total function from T to T, such that t `ref` $\leq t'$ `ref` implies $t = t'$. Moreover $t_1 \rightarrow t_2 \leq t$ `ref` and t `ref` $\leq t_1 \rightarrow t_2$ should both be false for any t, t_1, t_2 in T. Polytypes s are upward-closed subsets of T (*i.e. if* $t \in s$ and $t \leq t'$ then $t' \in s$). We write $\uparrow t$ for the upward closure of a monotype (the set of all its supertypes).

The terms and reduction rules in B(T) are identical to those in our system (excluding pairs and lists). While Pottier's presentation uses a different syntax for representing and updating the store, the presentations are equivalent, ours requiring only more reduction steps. We will stick to our presentation.

Typing judgments are written $\Gamma \models e : t$ with Γ a polytype environments (mapping identifiers to upward-closed sets of monotypes) and t a monotype. Typing rules[4] are given in figure 4. They are very similar to ours, you just have to transpose all τ's into t's and all $\vdash$ into $\models$. The only changes are that B-LET$_e$ is now monomorphic (this is the strict value restriction), subsumption B-SUB is added, and polymorphism is handled semantically in B-VAR and B-LET. AXIOMS for references are included.

The following theorem is proved in [17], section 3, for any $(T, \leq)$ satisfying the above requirements.

Theorem 1 (Subject Reduction). *If $e \rightarrow e'$, where e, e' are closed, then $\Gamma \models e : t$ implies $\Gamma \models e' : t$.*

For our purpose, we choose T as the set of all types generated by the type constructors `zero`, `int`, $\rightarrow$, `ref`, $\times$, `list` and the set of all type variables $\{\alpha, \beta, \ldots\}$. The variables are introduced here as type constants, to ease the translation, but they are unrelated to polymorphism: there is no notion of variable quantification in B(T). `zero` is an extra type constructor, which need not be included in our original language. The subtyping relation is defined as `zero` $\leq t$ and $t \leq t$ for any t in T, and extended through

[4] In Pottier's presentation, a judgment writes $\Gamma, M \models e : t$; we have merged Γ and M (M only mapping to monotypes), as our syntax for references permits. B-RHO merges B-STORE and B-CONF from the original presentation.

constructors, all covariant in their parameters, except `ref` which is non-variant, and $\rightarrow$ which is contravariant in its first parameter and covariant in its second one. This conforms to the requirements for B(T), meaning that subject reduction holds in the resulting system. We also extend the language, reduction and typing rules with PAIR, CONS and AXIOMS about Y, pairs and lists. Extending subject reduction to these features presents no challenge; the concerned reader is invited to check this (and other details of formalization), on the remarkably short proof in [17].

The progress lemma depends more directly on the syntax of expressions, and we cannot reuse directly Pottier's proof. However, our reduction and typing rules are basically the same as in [16].

Lemma 3 (Progress). *For any closed* e*, if for all* e' *such that* $e \xrightarrow{*} e'$ *there is* Γ *and* t *such that* $\Gamma \models e' : t$*, then reducing* e *either diverges or leads to a value.*

Combining the above subject reduction and progress, our instance of B(T) is sound.

We present now the translation itself. First we must be able to translate each component of a typing judgment. The expression part is left unchanged. Types are translated under a substitution $\xi : V \rightarrow T$.

$$[\![\alpha]\!]\xi = \xi(\alpha) \qquad [\![\tau_1 \times \tau_1]\!]\xi = [\![\tau_1]\!]\xi \times [\![\tau_2]\!]\xi$$
$$[\![\tau\ \mathtt{ref}]\!]\xi = [\![\tau]\!]\xi\ \mathtt{ref} \qquad [\![\tau\ \mathtt{list}]\!]\xi = [\![\tau]\!]\xi\ \mathtt{list}$$

This translation is extended to polytypes appearing in typing environments.

$$[\![\forall\alpha_1 \ldots \alpha_n.\tau]\!]\xi = \{t \mid (t_1, \ldots, t_n) \in T^n, [\![\tau]\!](\xi[\alpha_1 \mapsto t_1, \ldots, \alpha_n \mapsto t_n]) \leq t\}$$

Before going on to translate full derivations, we state a lemma about the single subsumption step we need.

Lemma 4. *Let* $\bar{\alpha}$ *be a set of type variables that appear only covariantly in* τ_1*. Let* ξ *be any translation substitution. Then* $[\![\forall\bar{\alpha}.\tau_1]\!]\xi = \uparrow[\![\tau_1]\!](\xi[\bar{\alpha} \mapsto \overline{\mathtt{zero}}])$.

Finally the derivation is translated by induction on its structure, transforming $\Gamma \vdash e : \tau$ into $[\![\Gamma]\!]\xi \models e : [\![\tau]\!]\xi$ for any ξ.

- if the last rule applied is LET$_e$ and $CovClose(\tau_1, \Gamma) = \forall\bar{\alpha}.\tau_1$ then it is translated into
$$\frac{[\![\Gamma]\!]\xi' \models e_1 : [\![\tau_1]\!]\xi' \quad [\![\Gamma[x \mapsto \forall\bar{\alpha}.\tau_1]]\!]\xi \models e_2 : [\![\tau_2]\!]\xi}{[\![\Gamma]\!]\xi \models \textsf{let}\ x = e_1\ \textsf{in}\ e_2 : [\![\tau_2]\!]\xi}(\text{B-LET}_e)$$
where $\xi' = \xi[\bar{\alpha} \mapsto \overline{\mathtt{zero}}]$. By Lemma 4, we have $[\![\forall\bar{\alpha}.\tau_1]\!]\xi = \uparrow[\![\tau_1]\!]\xi'$, and this is an instance of rule B-LET$_e$. Note also that $[\![\Gamma]\!]\xi' = [\![\Gamma]\!]\xi$ as $\alpha_i \cap FTV(\Gamma) = \emptyset$.
- if the last rule applied is LET$_v$ and $Close(\tau_1, \Gamma) = \forall\alpha_1 \ldots \alpha_n.\tau_1$, then it becomes
$$\frac{\dfrac{[\![\Gamma]\!]\xi' \models v : [\![\tau_1]\!]\xi'}{[\![\Gamma]\!]\xi' \models v : t}(\text{B-SUB}) \quad [\![\Gamma]\!]\xi[x \mapsto s] \models e : [\![\tau_2]\!]\xi}{[\![\Gamma]\!]\xi \models \textsf{let}\ x = v\ \textsf{in}\ e : [\![\tau_2]\!]\xi}(\text{B-LET}_v)$$
where $s = [\![\forall\alpha_1 \ldots \alpha_n.\tau_1]\!]\xi$, t ranges over all elements of s, and $\xi' = \xi[\alpha_1 \mapsto t_1, \ldots, \alpha_n \mapsto t_n]$ is such that $[\![\tau_1]\!]\xi' \leq t$. Here again $[\![\Gamma]\!]\xi' = [\![\Gamma]\!]\xi$.

- if the last rule applied is VAR, it becomes $\dfrac{[\![\tau]\!]\xi \in ([\![\Gamma]\!]\xi)(x)}{[\![\Gamma]\!]\xi \models x : [\![\tau]\!]\xi}$ (B-VAR).
- other cases are trivial induction.

From this construction we can obtain the following proposition.

Proposition 3. *If $\Gamma \vdash e : \tau$ is derivable in ML with the relaxed value restriction, then $[\![\Gamma]\!]\xi \models e : [\![\tau]\!]\xi$ is derivable in B(T) for any ξ.*

Now, suppose that we restrict ourselves to closed expressions whose types do not contain references nor function types. Normal forms of such expressions can only be data of the form:

$$d ::= \mathsf{nil} \mid (d, d) \mid \mathsf{cons}\ d$$

For such normal forms, type derivations in B(T) coincide with our system.

From this and type soundness for our instance of B(T) we can deduce the type soundness of ML with the relaxed value restriction, as stated below.

Theorem 2 (Type Soundness). *If $\emptyset \vdash e : \delta$ with δ any type of the form $\delta ::= \alpha \mid \delta \times \delta \mid \delta\ \mathtt{list}$, then reducing e either diverges or leads to a normal form d, and $\emptyset \vdash d : \delta$.*

7 Conclusion

Thanks to a small observation on the relation between polymorphism and subtyping —that zero in a covariant position is equivalent to a universally quantified type variable—, we have been able to smooth some of the rough edges of the value restriction, while keeping all of its advantages. This is a useful result, which has already been integrated in the Objective Caml 3.07 compiler. Hopefully this should make the use of polymorphic data structures easier.

Notwithstanding our achievements, this paper does nothing to solve the fundamental problem of the value restriction, namely that by assuming all functions to be imperative, it is overly pessimistic. We have been able to rescue some cases that were probably not even considered when it was introduced. But there is no easy solution for more involved cases, with polymorphic function types in the data.

The triviality of this result brings another question: why wasn't it discovered earlier? Actually, this specific use of subtyping is not new: the fact has not attracted very much attention, but our LET$_e$ rule is already admissible in HM(X). This could give yet another way to prove type soundness for our system: by defining it as a subsystem of a sufficiently feature-rich instance of HM(X) as found in [19]. We preferred B(T) for its robustness, and the lightness of its definition and proof, but this last approach would be purely syntactic.

Acknowledgments

I want to thank here Didier Rémy and other members of the Cristal team at INRIA for discussions and encouragements.

References

1. Wright, A.K.: Simple imperative polymorphism. Lisp and Symbolic Computation **8** (1995)
2. Garrigue, J.: Relaxing the value restriction (extended version). RIMS Preprint 1444, Research Institute for Mathematical Sciences, Kyoto University (2004) URL: `http://wwwfun.kurims.kyoto-u.ac.jp/~garrigue/papers/`.
3. Tofte, M.: Type inference for polymorphic references. Information and Computation **89** (1990) 1–34
4. Milner, R., Tofte, M., Harper, R.: The Definition of Standard ML. MIT Press, Cambridge, Massachusetts (1990)
5. Greiner, J.: SML weak polymorphism can be sound. Technical Report CMU-CS-93-160R, Canegie-Mellon University, School of Computer Science (1993)
6. Hoang, M., Mitchell, J., Viswanathan, R.: Standard ML-NJ weak polymorphism and imperative constructs. In: Proc. IEEE Symposium on Logic in Computer Science. (1993) 15–25
7. Talpin, J.P., Jouvelot, P.: The type and effect discipline. In: Proc. IEEE Symposium on Logic in Computer Science. (1992) 162–173
8. Leroy, X., Weis, P.: Polymorphic type inference and assignment. In: Proc. ACM Symposium on Principles of Programming Languages. (1991) 291–302
9. Leroy, X.: Polymorphic typing of an algorithmic language. Research report 1778, INRIA (1992)
10. Rémy, D., Vouillon, J.: Objective ML: A simple object-oriented extension of ML. In: Proc. ACM Symposium on Principles of Programming Languages. (1997) 40–53
11. Garrigue, J.: Programming with polymorphic variants. In: ML Workshop, Baltimore (1998)
12. Garrigue, J., Rémy, D.: Extending ML with semi-explicit higher order polymorphism. Information and Computation **155** (1999) 134–171
13. Ohori, A., Yoshida, N.: Type inference with rank 1 polymorphism for type-directed compilation of ML. In: Proc. International Conference on Functional Programming, ACM Press (1999)
14. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system release 3.06, Documentation and user's manual. Projet Cristal, INRIA. (2002)
15. Garrigue, J.: Simple type inference for structural polymorphism. In: The Ninth International Workshop on Foundations of Object-Oriented Languages, Portland, Oregon (2002)
16. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115** (1994) 38–94
17. Pottier, F.: A semi-syntactic soundness proof for HM(X). Research Report 4150, INRIA (2001)
18. Odersky, M., Sulzmann, M., Wehr, M.: Type inference with constrained types. Theory and Practice of Object Systems **5** (1999) 35–55
19. Skalka, C., Pottier, F.: Syntactic type soundness for HM(X). In: Proceedings of the 2002 Workshop on Types in Programming (TIP'02). Volume 75 of Electronic Notes in Theoretical Computer Science., Dagstuhl, Germany (2002)

Rigid Mixin Modules

Tom Hirschowitz

ENS Lyon

Abstract. Mixin modules are a notion of modules that allows cross-module recursion and late binding, two features missing in ML-style modules. They have been well defined in a call-by-name setting, but in a call-by-value setting, they tend to conflict with the usual static restrictions on recursive definitions. Moreover, the semantics of instantiation has to specify an order of evaluation, which involves a difficult design choice. Previous proposals [14, 16] rely on the dependencies between components to compute a valid order of evaluation. In such systems, mixin module types must carry some information on the dependencies between their components, which makes them verbose. In this paper, we propose a new, simpler design for mixin modules in a call-by-value setting, which avoids this problem.

1 Introduction

1.1 The Problem

For programming "in the large", it is desirable that the programming language offers linguistic support for the decomposition and structuring of programs into modules. A good example of such linguistic support is the ML module system and its powerful notion of parameterized modules. Nevertheless, this system is weak on two important points.

(Mutual recursion) Mutually recursive definitions cannot be split across separate modules. There are several cases where this hinders modularization [6].
(Modifiability) The language does not propose any mechanism for incremental modification of an already-defined module, similar to inheritance and overriding in object-oriented languages.

Class-based object-oriented languages provide excellent support for these two features. Classes are naturally mutually recursive, and inheritance and method overriding answer the need for modifiability. However, viewed as a module system, classes have two weaknesses: they do not offer a general parameterization mechanism (no higher-order functions on classes), and the mechanisms they offer to describe pre-computations (initialization of static and instance variables) lack generality, since a module system should allow to naturally alternate function definitions with computational definitions using these functions.

Mixin modules [4] (hereafter simply called "mixins") provide an alternative approach to modularity that combines some of the best aspects of classes and

© Springer-Verlag Berlin Heidelberg 2004

ML-style modules. Mixins are modules with “holes” (not-yet-defined components), where the holes can be plugged later by composition with other mixins, following a late-binding semantics. However, the handling of pre-computations and initializations in mixins is still problematic. Most of the previous work on mixins, notably by Ancona and Zucca [2] and Wells and Vestergaard [20], is better suited to a call-by-name evaluation strategy. This strategy makes it impossible to trigger computations at initialization time (see Sect. 6 for more details).

The choice of a call-by-value setting raises the following two issues.

(Recursive definitions) Since mixin components are not necessarily functions, arbitrary recursive definitions can appear dynamically by composition. For instance, consider the following two mixins, (in an informal concrete syntax)

```
mixin A = mix                    mixin B = mix
  ? x : int                        ? y : int
  ! y = x + 1         and          ! x = y * 2
end                              end
```

Each of these mixins declares the missing value (marking it with `?`) and defines the other one (marking it with `!`). The composition of `A` and `B` involves the mutually recursive definition `x = y * 2` and `y = x + 1`.

In most call-by-value languages, recursive definitions are statically restricted, in order to be more efficiently implementable [3, 15], and to avoid some ill-founded definitions. Obviously, our system should not force language designers to abandon these properties, and thus needs guards on recursive definitions, at the level of both static and dynamic semantics.

(Order of evaluation) In our system, mixins will contain arbitrary, unevaluated definitions, whose evaluation will be triggered by instantiation. Because these definitions are arbitrary, the order in which they will be evaluated matters. For instance, in a mixin `A` defining `x = 0` and `y = x + 1`, `x` must be evaluated before `y`. Thus, the semantics of instantiation must define an order of evaluation. Moreover, mixins can be built by composition, so the semantics of composition must also take the order of definitions into account.

From the standpoint of dynamic semantics, the second issue involves a design decision. From the standpoint of typing, it reduces to the first issue, since the existence of a valid order of evaluation is governed by the absence of invalid recursive definitions.

1.2 Instantiation-Time Ordering: Flexible Mixin Modules

The *MM* language of call-by-value mixins [14, 16,12] is designed as follows. Mixins contain unordered definitions. Only at instantiation does the system compute an order for them, according to their inter-dependencies [14, 16, 12], and to programmer-supplied annotations that fix some bits of the final order [16, 12]. This solution, which we call *flexible mixins* is very expressive w.r.t. code reuse, since components can be re-ordered according to the context. However, it appears too complex in some respects.

(Instantiation) In particular, instantiation is too costly, since it involves computing the strongly-connected components of a graph whose size is quadratic in the input term, plus a topological sort of the result.

(Type safety) As explained above, when recursive definitions are restricted, the type system must prevent invalid ones. In *MM,* mixin types contain some information about the dependencies between definitions. Nevertheless, this makes mixin types verbose, and also over-specified, in the sense that any change in the dependencies between components can force the type of the mixin to change, which is undesirable.

The first problem is not so annoying in the context of a module system: it only has to do with linking operations, and thus should not affect the overall efficiency of programs. The second problem makes the proposed language impractical without dedicated graph support.

1.3 Early Ordering: Rigid Mixin Modules

In this paper, we propose a completely different approach, from scratch. We introduce *Mix,* a new language of call-by-value mixins, where mixin components are ordered, in a rigid way. They can be defined either as single components (briefly called “singles”) or as blocks of components. Blocks contain mutually recursive definitions, and are restricted to a certain class of values. Conversely, singles can contain arbitrary, non-recursive computations. Composition preserves the order of both of its arguments, and instantiation straightforwardly converts its argument into a module.

In *Mix,* the components of a mixin are ordered once for all at definition time, so *Mix* is less expressive than *MM.* Yet, it has other advantages. First, with respect to side effects, annotations are no longer needed, since side effects always respect the syntactic order. Moreover, instantiation is less costly than in *MM,* since it runs in $O(n \; log \; n)$, where n is the size of the input. Concerning typing, mixin types have the same structure as mixins themselves: they are sequences of specifications, which can be either singles or blocks. They avoid the use of explicit graphs, which improves over *MM.* Compared to ML module types, the only differences are that the order matters and that mutually recursive specifications must be explicitly grouped together. Finally, the meta theory of *Mix* is much simpler than the one of *MM,* which makes it more likely to scale up to a fully-featured language. In summary, we propose *Mix* as a good trade-off between expressiveness and user-friendliness for incorporating mixins into a practical programming language like ML.

The rest of the paper is organized as follows. Section 2 presents an informal overview of *Mix* by example. Section 3 formally defines *Mix* and its dynamic semantics. Section 4 defines a sound type system for *Mix*. Finally, sections 5 and 6 review related and future work, respectively. The proofs are omitted from this paper for lack of space, but a longer version including them is available as a research report [13].

2 Intuitions

As a simplistic introductory example, consider a program that defines two mutually recursive functions for testing whether an integer is even or odd, and then tests whether 56 is even, and whether it is odd. Assume now that it is conceptually obvious that everything concerning oddity must go into one program fragment, and everything concerning evenness must go into another, clearly distinct fragment. Here is how this can be done in an informal programming language based on *Mix,* with a syntax mimicking OCaml [17].

First, define two mixins Even and Odd as follows.

```
mixin Even = mix
  recblock ? odd : int -> bool
       and ! even x = x = 0 or odd (x-1)
  ! even56 = even 56
end

mixin Odd = mix
  recblock ? even : int -> bool
       and ! odd x = x > 0 and even (x-1)
  ! odd56 = odd 56
end
```

Each of these mixins declares the missing function (marking it with ?) and defines the other one (marking it with !), inside a **recblock** which delimits a recursive block. Then, outside of this block, each mixin performs one computation.

In order to link them, and obtain the desired complete mixin, one composes Even and Odd, by writing **mixin** OpenNat = Odd >> Even. Intuitively, composition connects the missing components of Even to the corresponding definitions of Odd, and *vice versa,* preserving the order of both mixins. Technically, composition somehow passes Odd through Even, with Even acting as a filter, stopping the components of Odd when they match one of its own components. This filtering is governed by some rules: the components of Odd go through Even together, until one of them, say component *c*, matches some component of Even. Then, the components of Odd defined to the left of *c* are stuck at the current point. The other components continue their way through Even. Additionally, when two components match, they are merged into a single component.

In our example, odd and even both stop at Even's recursive block mentioning them, so the two recursive blocks are merged. Further, odd56 remains is unmatched, so it continues until the end of Even. The obtained mixin is thus equivalent to

```
mixin OpenNat = mix
  recblock ! even x = x = 0 or odd (x-1)
       and ! odd x = x > 0 and even (x-1)
  ! even56 = even 56
  ! odd56 = odd 56
end
```

Note that composition is asymmetric. This mixin remains yet to be instantiated, in order to trigger its evaluation. This is done by writing `module Nat = close OpenNat`, which makes `OpenNat` into a module equivalent to

```
module Nat = struct
  let rec even x = x = 0 or odd (x-1)
          odd x = x > 0 and even (x-1)
  let even56 = even 56
  let odd56 = odd 56
end
```

which evaluates to the desired result. For comparison, in *MM*, the final evaluation order would be computed upon instantiation, instead of composition. It would involve a topological sort of the strongly-connected components of the dependency graph of `OpenNat`. Incidentally, in *MM*, in order to ensure that `even56` is evaluated before `odd56`, the definition of `odd56` should better explicitly state it. From the standpoint of typing, explicitly grouping possibly recursive definitions together allows to get rid of dependency graphs in the types, thus greatly simplifying the type system.

3 The *Mix* Language and Its Dynamic Semantics

3.1 Syntax

Pre-terms. Figure 1 defines the set of *pre-terms* of *Mix*. It distinguishes names X from variables x, following Harper and Lillibridge [11]. It includes a standard record construct $\{s\}$, where $s ::= (X_1 = e_1 \dots X_n = e_n)$, and selection $e.X$. It features two constructs for value binding, letrec for mutually recursive definitions, and let for single, non-recursive definitions. Finally, the language provides four mixin constructs. Basic mixins consist of *structures* $m = (c_1 \dots c_n)$, wich are lists of *components*. A component c is either a *single*, or a *block*. A single u is either a named *declaration* $X \rhd x = \bullet$, or a *definition* $L \rhd x = e$, where L is a *label*. Labels can be names or the special *anonymous* label, written _, in which case the definition is also said anonymous. Finally, a block q is a list of singles. The other constructs are composition $(e_1 \gg e_2)$, instantiation (close e), and deletion of a name X, written $(e_{|-X})$.

Terms. Proper terms are defined by restricting the set of pre-terms, as follows. We define *Mix values* v by $v ::= \{s^v\} \mid \langle m \rangle$, where $s^v ::= (X_1 = v_1 \dots X_n = v_n)$. Then The system is parameterized by the set *RecExp* of *valid recursive expressions,* which must contain only values, and be closed under substitution.

Definition 1 (Terms)
A term *of Mix is a pre-term such that: records do not define the same name twice ; bindings do not define the same variable twice ; structures define neither*

x	$\in$	*Vars*	Variable
X	$\in$	*Names*	Name
L	$\in$	*Names* $\cup \{\_\}$	Label
Expression:			
e	$::=$	x	Variable
	$\mid$	$\{X_1 = e_1 \ldots X_n = e_n\}$	Record
	$\mid$	$e.X$	Selection
	$\mid$	letrec $x_1 = e_1 \ldots x_n = e_n$in e	letrec
	$\mid$	let $x = e_1$ in e_2	let
	$\mid$	$\langle c_1 \ldots c_n \rangle$	Structure
	$\mid$	$e_1 \gg e_2$	Composition
	$\mid$	close e	Instantiation
	$\mid$	$e_{\mid -X}$	Deletion
Definition:			
c	$::=$	u	Single definition
	$\mid$	$[u_1 \ldots u_n]$	Block
Single definition:			
u	$::=$	$L \rhd x = e \mid X \rhd x = \bullet$	

Fig. 1. Syntax

the same name twice nor the same variable twice ; and the right-hand sides of letrec *and block definitions belong to RecExp.*

The restriction of letrec and block definitions to valid recursive expressions both simplifies the semantics of letrec, and models the restrictions put by standard call-by-value languages on recursive definitions [17, 19]. Typically, recursive definitions can only be functions.

Records, bindings and structures are respectively considered as finite maps from names to terms, variables to terms, and pairs of a label and a variable to terms or $\bullet$. Thus, given a structure m, the restriction of $dom(m)$ (the domain of m) to pairs of a name and a variable can be seen as an injective finite map from names to variables, which we call $VofN(m)$.

Terms are considered equivalent modulo proper [20] renaming of bound variables and modulo the order in blocks and letrec. We denote by $DV(m)$and $DV(b)$ the sets of variables defined by m and b, respectively, and by $DN(m)$ and $DN(s)$ the sets of names defined by m and s, respectively.

3.2 Dynamic Semantics

The semantics of *Mix* is defined as a reduction relation on pre-terms in figure 2, using notions defined in figure 3. It is compatible with bound variable renaming and preserves well-formedness, so we extend it to terms.

Figure 3 defines *evaluation contexts,* which enforce a deterministic, call-by-value strategy. We can now examine the rules, from the most interesting to the most standard.

$$
\boxed{\begin{array}{rll}
\langle m_1 \rangle \gg \langle m_2 \rangle & \to \langle \mathit{Add}(m_1, m_2, \varepsilon) \rangle \text{ if } m_1 \asymp m_2 & (\textsc{Compose}) \\
\mathsf{close}\ \langle m^c \rangle & \to \mathit{Bind}(m^c, \{\mathit{Record}(m^c)\}) & (\textsc{Close}) \\
\langle m \rangle_{|-X} & \to \langle \mathit{Del}(m, X) \rangle & (\textsc{Delete}) \\
\mathsf{letrec}\ b\ \mathsf{in}\ e & \to \{x \mapsto \mathsf{letrec}\ b\ \mathsf{in}\ b(x) \mid x \in \mathit{dom}(b)\}(e) & (\textsc{LetRec}) \\
\mathsf{let}\ x = v\ \mathsf{in}\ e & \to \{x \mapsto v\}(e) & (\textsc{Let}) \\
\{s^v\}.X & \to s^v(X) & (\textsc{Select}) \\
\mathbb{E}[e] & \to \mathbb{E}[e'] \text{ if } e \to e' & (\textsc{Context})
\end{array}}
$$

Fig. 2. Reduction rule

Composition. Rule COMPOSE describes mixin composition. In order to be composed, the structures must be made *compatible* by α-conversion. Namely, we say that two structures m_1 and m_2 are compatible, and write $m_1 \asymp m_2$, iff $DV(m_1) \cap FV(\langle m_2 \rangle) = DV(m_2) \cap FV(\langle m_1 \rangle) = \emptyset$, and for any $x \in DV(m_1) \cap DV(m_2)$, there exists a name X such that $\mathit{VofN}(m_1)(X) = \mathit{VofN}(m_2)(X) = x$. This basically says that both structures agree on the names of variables.

Then, their composition $\langle m_1 \rangle \gg \langle m_2 \rangle$ is $\mathit{Add}(m_1, m_2, \varepsilon)$, where *Add* is defined by induction on m_1 by

$$
\begin{array}{rcl}
\mathit{Add}(\varepsilon, m_1, m_2) & = & m_1, m_2 \\
\mathit{Add}((m_1, c), m_2, m_3) & = & \mathit{Add}(m_1, m_2, (c, m_3)) \\
& & \quad \text{if } \mathit{DN}(c) \cap \mathit{DN}(m_2, m_3) = \emptyset \\
\mathit{Add}((m_1, c_1), (m_2^1, c_2, m_2^2), m_3) & = & \mathit{Add}(m_1, m_2^1, (c_1 \otimes c_2, m_2^2, m_3)) \\
\multicolumn{3}{r}{\text{if } \mathit{DN}(c_1) \cap \mathit{DN}(m_2^1, m_2^2, m_3) = \emptyset \text{ and } \mathit{DN}(c_1) \cap \mathit{DN}(c_2) \neq \emptyset}
\end{array}
$$

Given three arguments m_1, m_2, m_3, Add roughly works as follows. If m_1 is empty, it returns the concatenation of m_2 and m_3. If the last component c of m_1 defines names that are not defined in m_2 or m_3, then c is pushed at the head of m_3. Finally, when the last component c_1 of m_1 defines a name also defined by some c_2 in m_2, so that $m_2 = (m_2^1, c_2, m_2^2)$, then the third argument becomes $(c_1 \otimes c_2, m_2^2, m_3)$, where $c_1 \otimes c_2$ is the *merging* of c_1 and c_2, which is defined by

$$
\begin{array}{rcll}
c_1 \otimes c_2 & = & c_2 \otimes c_1 & \\
(X \triangleright x = \bullet) \otimes c & = & c & \text{if } \mathit{VofN}(c)(X) = x \\
[q_1] \otimes [q_2] & = & [q_1, q_2] & \text{if } \mathit{DN}(q_1) \cap \mathit{DN}(q_2) = \emptyset \\
[X \triangleright x = \bullet, q_1] \otimes [q_2] & = & [q_1] \otimes [q_2] & \text{if } \mathit{VofN}(q_2)(X) = x
\end{array}
$$

This definition is not algorithmic, but uniquely defines the merging of two components, and an algorithm is easy to derive from it: one has to apply rules 2 and 4 as long as possible, then commute the arguments and apply rules 2 and 4 as long as possible again, and then finally apply rule 3. Technically, as soon as a declaration is matched, it is removed, and when two blocks have no more common defined names, their merging is their union. Note that initially, only components with common defined names are merged, but that the union takes place after all the common names have been reduced.

$$\begin{aligned}\mathbb{E} ::= &\ \{s^v, X = \square, s\} \mid \square.X \\ &\mid \mathsf{let}\ x = \square\ \mathsf{in}\ e \\ &\mid \square \gg e \mid v \gg \square \\ &\mid \mathsf{close}\ \square \mid \square_{|-X}\end{aligned}$$

Fig. 3. Evaluation contexts

Example 1. Assuming that *Mix* is extended with functions, integers and booleans, the mixin Even described in section 2 is written

$$\begin{aligned} even = \langle\, [\, &Odd \rhd odd = \bullet, \\ &\quad Even \rhd even = \lambda x.(x = 0)\ \mathsf{or}\ odd\ (x - 1)\ ], \\ &Even56 \rhd even56 = even\ 56\ \rangle \end{aligned}$$

During composition with the mixin corresponding to Odd, the component *Odd56* traverses the whole structure to go to the rightmost position, then the two blocks defining *Even* and *Odd* are merged, which gives the expected block.

Instantiation. Rule CLOSE describes the instantiation of a *complete* basic mixin. A structure is said *complete* iff it does not contain declarations. We denote complete structures, components, singles, and blocks by m^c, c^c, u^c, and q^c, respectively. Given a complete basic mixin $\langle m^c \rangle$, instantiation first generates a series of bindings, following the structure of m^c, and then stores the results of named definitions in a record. Technically, close $\langle m^c \rangle$ reduces to $Bind(m^c, \{Record(m^c)\})$, where Record makes m^c into a record and *Bind* makes m^c into a binding:

$Record(m^c)$ is defined on singles by

$$Record(X \rhd x = e) = (X = x) \qquad \text{and} \qquad Record(\_ \rhd x = e) = \varepsilon,$$

naturally extended to components and structures by concatenation,

and $Bind(m^c, e)$ is defined inductively over m^c by

$$\begin{aligned} Bind(\varepsilon, e) &= e \\ Bind(([{u^c}_1 \ldots {u^c}_n], m^c), e) &= \mathsf{letrec}\ \lfloor {u^c}_1 \rfloor \ldots \lfloor {u^c}_n \rfloor\ \mathsf{in}\ Bind(m^c, e) \\ Bind((u^c, m^c), e) &= \mathsf{let}\ \lfloor u^c \rfloor\ \mathsf{in}\ Bind(m^c, e), \\ \text{with}\ \lfloor L \rhd x = e \rfloor &= (x = e). \end{aligned}$$

For each component, *Bind* defines a letrec (if the component is a block) or a let (if the component is a single), by extracting bindings $x = e$ from singles $L \rhd (x = e)$.

Other rules. Rule DELETE describes the action of the deletion operation. Given a basic mixin $\langle m \rangle$, $\langle m \rangle_{|-X}$ reduces to $\langle Del(m, X) \rangle$, where $Del(m, X)$ denotes m, where any definition of the shape $X \rhd x = e$ is replaced with $X \rhd x = \bullet$.

Type:

$$\begin{aligned}\tau \in \mathit{Types} &::= \{S\} \mid \langle C_1 \ldots C_n \rangle \\ C &::= U \mid [U_1 \ldots U_n] \\ U &::= \delta X : \tau \\ \delta &::= \,! \mid ? \\ S &\in \mathit{Names} \xrightarrow{\text{fin}} \mathit{Types}\end{aligned}$$

Environment:

$$\Gamma \in \mathit{Vars} \xrightarrow{\text{fin}} \mathit{Types}$$

Fig. 4. Types

The next two rules, LETREC, LET, handle value binding. The only non-obvious rule is LETREC, which enforces the following behavior. The idea is that the rule applies when the considered binding is fully evaluated, which is always the case for proper terms. A pre-term letrec b in e reduces to e, where each $x \in dom(b)$ is replaced with a kind of closure representing its definition, namely letrec b in $b(x)$. Note the notation for capture-avoiding substitution.

Finally, rule SELECT defines record selection, and rule CONTEXT extends the rule to any evaluation context.

4 Static Semantics

We now define a sound type system for *Mix* terms. Defining it on terms rather than pre-terms means that the considered expressions are well-formed by definitions. Types are defined in figure 4. A *Mix* type τ can be either a record type or a mixin type. A mixin type has the shape $\langle M \rangle$, where M is a *signature.* A signature is a list of *specifications* C, which can be either *single specifications* U or *block specifications* Q. A single specification has the shape $\delta X : \tau$ where δ is a flag indicating whether the considered name is a declaration of a definition. It can be either ?, for declarations, or !, for definitions. A block specification is a list of single specifications. Record types are finite maps from names to types. Types are identified modulo the order of specifications in blocks. Environments Γ are finite maps from variables to types. The disjoint union of two environments Γ_1 and Γ_2 is written $\Gamma_1 + \Gamma_2$ (which applies only if their domains are disjoint).

Figure 5 presents our type system for *Mix.*

Basic mixin modules and enriched specifications. Let us begin with the typing of basic mixins. Rule T-STRUCT simply delegates the typing of a basic mixin $\langle m \rangle$ to the rules for typing structures. These rules basically give each component c an *enriched specification,* which is a specification, enriched with the corresponding variable. Formally, single enriched specifications have the shape $\delta L \triangleright x : \tau$, and enriched block specifications are finite sets of these. Notably, this allows to type anonymous definitions (using enriched specifications like $\delta_{\_} \triangleright x : \tau$), and also to recover a typing environment (namely $\{x \mapsto \tau\}$) for typing the next components.

Expressions

T-Struct
$$\frac{\Gamma \vdash c_1 \ldots c_n : M^e}{\Gamma \vdash \langle c_1 \ldots c_n \rangle : \langle \mathit{Sig}(M^e) \rangle}$$

T-Compose
$$\frac{\Gamma \vdash e_1 : \langle M_1 \rangle \qquad \Gamma \vdash e_2 : \langle M_2 \rangle}{\Gamma \vdash e_1 \gg e_2 : \langle \mathit{Add}(M_1, M_2, \varepsilon) \rangle}$$

T-Close
$$\frac{\Gamma \vdash e : \langle M^c \rangle}{\Gamma \vdash \mathsf{close}\ e : \{\mathit{Record}(M^c)\}}$$

T-Delete
$$\frac{\Gamma \vdash \langle m \rangle : \langle M \rangle}{\Gamma \vdash \langle m \rangle_{|-X} : \langle \mathit{Del}(M, X) \rangle}$$

T-Var
$$\Gamma \vdash x : \Gamma(x)$$

T-Record
$$\frac{\mathit{dom}(s) = \mathit{dom}(S) \qquad \forall X \in \mathit{dom}(s), \Gamma \vdash s(X) : S(X)}{\Gamma \vdash \{s\} : \{S\}}$$

T-Select
$$\frac{\Gamma \vdash e : \{S\}}{\Gamma \vdash e.X : S(X)}$$

T-LetRec
$$\frac{\Gamma + \Gamma_b \vdash b : \Gamma_b \qquad \Gamma + \Gamma_b \vdash e : \tau}{\Gamma \vdash \mathsf{letrec}\ b\ \mathsf{in}\ e : \tau}$$

T-Let
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma + \{x \mapsto \tau_1\} \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

Singles

T-Some
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (L \rhd x = e) : (!L \rhd x : \tau)}$$

T-None
$$\Gamma \vdash (X \rhd x = \bullet) : (?X \rhd x : \tau)$$

Structures and bindings

T-Empty
$$\Gamma \vdash \varepsilon : \varepsilon$$

T-Single
$$\frac{\Gamma \vdash u : U^e \qquad \Gamma + \mathit{Env}(U^e) \vdash m : M^e}{\Gamma \vdash (u, m) : (U^e, M^e)}$$

T-Block
$$\frac{\Gamma + \Gamma_q \vdash m : M^e \qquad \Gamma_q = \biguplus_{u \in q} \mathit{Env}({U^e}_u) \qquad \forall u \in q, u \in \mathit{RecExp}_? \text{ and } \Gamma + \Gamma_q \vdash u : {U^e}_u}{\Gamma \vdash ([q], m) : ([\biguplus_{u \in q} {U^e}_u], M^e)}$$

T-Binding
$$\frac{\mathit{dom}(b) = \mathit{dom}(\Gamma_b) \qquad \forall x \in \mathit{dom}(b), b(x) \in \mathit{RecExp} \text{ and } \Gamma_b \vdash b(x) : \Gamma_b(x)}{\Gamma \vdash b : \Gamma_b}$$

Fig. 5. Type system

Enriched single specifications, block specifications, and signatures are denoted by U^e, Q^e, and M^e, respectively. Once the structure m has been given such an enriched signature M^e, this result is converted to a proper signature $M = \mathit{Sig}(M^e)$, assigning to the basic mixin $\langle m \rangle$ the type $\langle M \rangle$. The *Sig* function merely forgets variables and anonymous definitions of its argument: it is defined by straightforward extension of

$$\mathit{Sig}(\delta X \rhd x : \tau) = \delta X : \tau \qquad \mathit{Sig}(\delta_{\_} \rhd x : \tau) = \varepsilon.$$

Here is how structures are given such enriched signatures. By rule T-SOME, a single definition $L \rhd x = e$ is given the enriched single specification $!L \rhd x : \tau$ if e has type τ. By rule T-NONE, a single declaration $X \rhd x = \bullet$ can be given any enriched specification of the shape $?X \rhd x : \tau$.

Given this, we can define the typing of structures. By rule T-EMPTY, an empty structure is given the empty signature. By rule T-SINGLE, a structure of the shape (u, m) is typed as follows. First, u is typed, yielding an enriched specification U^e. This U^e is made into an environment by the *Env* function from enriched signatures to environments. This function associates to any enriched single specification $\delta L \rhd x : \tau$ the finite map $\{x \mapsto \tau\}$, and is straightforwardly extended to signatures. The obtained environment is added to the current environment for typing the remaining components inductively, yielding an enriched signature M^e. The type of the whole structure is (U^e, M^e).

By rule T-BLOCK, a structure of the shape $([q], m)$ is typed as follows. An enriched single specification ${U^e}_u$ is guessed for each single u of q. Then, the set of these enriched single specifications is converted into an environment $\Gamma_q = \biguplus_{u \in q} \mathit{Env}({U^e}_u)$. This environment Γ_q is added to the current environment. Then, it is checked that each single u indeed has the enriched specification ${U^e}_u$. Additionally, it is checked that each single u of q is defined by a valid recursive expression or is a declaration. By abuse of notation, we write this $u \in \mathit{RecExp}_?$. Finally, the structure m is typed, yielding an enriched signature M^e, which is concatenated to $[\biguplus_{u \in q} {U^e}_u]$.

Composition. The typing of composition, defined by rule T-COMPOSE, recalls its dynamic semantics. The type of the composition of two mixins of types $\langle M_1 \rangle$ and $\langle M_2 \rangle$, respectively, is $\langle \mathit{Add}(M_1, M_2, \varepsilon) \rangle$, where Add is defined by

$$\begin{array}{rcl}
\mathit{Add}(\varepsilon, M_1, M_2) & = & M_1, M_2 \\
\mathit{Add}((M_1, C), M_2, M_3) & = & \mathit{Add}(M_1, M_2, (C, M_3)) \\
 & & \quad \text{if } \mathit{DN}(C) \cap \mathit{DN}(M_2, M_3) = \emptyset \\
\mathit{Add}((M_1, C_1), (M_2^1, C_2, M_2^2), M_3) & = & \mathit{Add}(M_1, M_2^1, (C_1 \otimes C_2, M_2^2, M_3)) \\
\multicolumn{3}{r}{\text{if } \mathit{DN}(C_1) \cap \mathit{DN}(M_2^1, M_2^2, M_3) = \emptyset \text{ and } \mathit{DN}(C_1) \cap \mathit{DN}(C_2) \neq \emptyset}
\end{array}$$

which does the same as *Add* on structures. The merging of two specifications is similarly defined by

$$\begin{array}{rcll}
C_1 \otimes C_2 & = & C_2 \otimes C_1 & \\
(?X : \tau) \otimes C & = & C & \text{if } C(X) = \tau \\
[Q_1] \otimes [Q_2] & = & [Q_1, Q_2] & \text{if } \mathit{DN}(Q_1) \cap \mathit{DN}(Q_2) = \emptyset \\
[?X : \tau, Q_1] \otimes [Q_2] & = & [Q_1] \otimes [Q_2] & \text{if } Q_2(X) = \tau
\end{array}$$

It differs from component merging, because it checks that the types of matching specifications are the same.

Example 2. When some mutually recursive definitions are grouped together in blocks, rule T-BLOCK ensures that they are all defined by valid recursive expressions. Let us now show how the type system rules out mutually recursive definitions that are not in blocks. Assume given mixins with types $e_1 : \langle ?Y : \tau_Y, !X : \tau_X \rangle$ and $e_2 : \langle ?X : \tau_X, !Y : \tau_Y \rangle$. When typing their composition $e_1 \gg e_2$, the component X in e_1, by the third rule in the definition of *Add*, is merged with its counterpart in e_2. This pushes the component Y of e_2 to the right, so that we obtain the triple $(?Y : \tau_Y, \varepsilon, (!X : \tau_X, !Y : \tau_Y))$, to which no rule applies.

Other rules. Rule T-CLOSE types instantiation. Following previous notation, we let M^c denote complete signatures. Given a complete mixin of type $\langle M^c \rangle$, close makes it into a record. The type of the result is $\{Record(M^c)\}$, which is obtained by flattening the blocks in M^c, forgetting the ! flags.

Rule T-DELETE types deletion. For a mixin e of type $\langle M \rangle$, the rule gives $e_{|-X}$ the type $\langle Del(M, X) \rangle$, in which *Del(M, X)* denotes *M,* where any declaration of the shape $!X : \tau$ is replaced with $?X : \tau$.

The other typing rules are straightforward.

Soundness. The type sytem is sound, in the sense that the following results hold.

Lemma 1 (Subject reduction)
If $\Gamma \vdash e : \tau$ *and* $e \longrightarrow e'$, *then* $\Gamma \vdash e' : \tau$.

Lemma 2 (Progress)
If $\emptyset \vdash e : \tau$, *then either* e *is a value, or there exists* e' *such that* $e \longrightarrow e'$.

Theorem 1 (Soundness)
If $\emptyset \vdash e : \tau$, *then either* e *reduces to a value, or its evaluation does not terminate.*

5 Related Work

Kernel calculi with mixin modules. The idea of mixin modules comes from that of mixins, introduced by Bracha [4] as a model of inheritance. In this model, called Jigsaw, classes are represented by *mixins,* which are equipped with a powerful set of modularity operations, and can be instantiated into *objects.* Syntactically, mixins may contain only values, which makes them as restrictive as classes. What differentiates them from classes is their cleaner design, which gave other authors the idea to generalize them to handle modules as well as objects.

Ancona and Zucca [2] propose a call-by-name module system based on some of Bracha's ideas, called *CMS.* As *Mix, CMS* extends Jigsaw by allowing any kind of expressions as mixin definitions, not just values. Unlike in *Mix,* in *CMS,* there is no distinction between modules and mixin modules, which makes sense in call-by-name languages, since the contents of modules are not evaluated until selection. In call-by-value, the contents of a module are eagerly evaluated, so they cannot have a late binding semantics. Thus, modules must be dinstinguished from mixin modules, and so *CMS* is not a suitable model. From the standpoint of typing, *CMS,* unlike *Mix,* but consistently with most call-by-name languages, does not control recursive definitions.

The separation between mixin modules and modules, as well as late binding, can be encoded in Wells and Vestergaard's **m**-calculus [20], which however is untyped, and does not provide programmer control over the order of evaluation.

In a more recent calculus [1], Ancona et al. separate mixin modules from modules, and handle side effects as a monad. However, they do not attempt to statically reject faulty recursive definitions. Moreover, in their system, given a composition $e_1 \gg e_2$, the monadic (i.e., side-effective) definitions of e_1 are necessarily evaluated before those of e_2, which is less flexible than our proposal.

Language designs with mixin modules. Duggan and Sourelis [8] propose an extension of ML with mixin modules, where mixin modules are divided into a *prelude,* a *body,* and an *initialization section.* Only definitions from the body are concerned by mixin module composition, the other sections being simply concatenated (and disjoint). Also, the body is restricted to functions and data-type definitions, which prevents illegal recursive definitions from arising dynamically. This is less flexible than *Mix,* since it considerably limits the alternation of functional and computational definitions.

Flatt and Felleisen [10] introduce the closely related notion of *units,* in the form of (1) a theoretical extension to Scheme and ML and (2) an actual extension of their PLT Scheme implementation of Scheme [9]. In their theoretical work, they only permit values as unit components, except for a separate initialization section. This is more restrictive than *Mix,* in the same way as Duggan and Sourelis. In the implementation, however, the semantics is different. Any expression is allowed as a definition, and instantiation works in two phases. First, all fields are initialized to `nil`; and second, they are evaluated and updated, one after another. This yields both unexpected behavior (consider the definition `x = cons(1, x)`), and dynamic type errors (consider `x = x + 1`), which do not occur in *Mix.* Finally, units do not feature late binding, contrarily to *Mix.*

Linking calculi. Other languages that are close to mixin modules are linking calculi [5, 18]. Generally, they support neither nested modules nor late binding, which significantly departs from *Mix.* Furthermore, among them, Cardelli's proposal [5] does not restrict recursion at all, but the operational semantics is sequential in nature and does not appear to handle cross-unit recursion. As a result, the system seems to lack the progress property. Finally, Machkasova and Turbak [18] explore a linking calculus with a very rich equational theory, but which does not restrict recursion and is untyped.

Flexible mixin modules. In the latest versions of *MM* [12], a solution to the problem of dependency graphs in types is proposed. Instead of imposing that the graph in a mixin module type exactly reflect the dependencies of the considered mixin module, it is seen as a bound on its dependencies, thanks to an adequate notion of subtyping. Roughly, it ensures that the considered mixin module has no more dependencies than exposed by the graph. This allows two techniques for preventing *MM* mixin module types from being verbose and over-specified. First, the interfaces of a mixin module e can be given more constrained dependency graphs than that of e. This makes interfaces more robust to later changes. Second, a certain class of dependency graphs is characterized, that bear a convenient syntactic description, thus avoiding users to explicitly write graphs by hand. In fact, this syntactic sugar allows to write *MM* types exactly as *Mix* types. We call MM^2 the language obtained by restricting *MM* to such types (in a way that remains to be made precise, for instance by insertion of implicit coercions).

The expressive power of typed MM^2 w.r.t. reordering lies between *MM* and *Mix*. Intuitively, the order in *Mix* mixins is fixed at definition time, while MM^2 allows later reordering of input components. For instance, the *Mix* basic mixin $e_1 = \langle X \rhd x = \bullet, Y \rhd y = \bullet \rangle$ cannot be composed with $e_2 = \langle !Y \rhd y = 0, !X \rhd x = 0 \rangle$, which is unfortunate. The equivalent composition is well-typed in MM^2.

The importance of this loss of flexibility has to be further investigated. Intuitively, it can only be annoying when a mixin is reused in an unexpected way, which makes the initial order incorrect. Unfortunately, the classical examples using mixins modules (and modules in general) generally show the modular decomposition of programs, not really the reuse of existing code, with possibly badly ordered components. This comes from the lack of extensive practice of any system with mixin modules, which is one of our priorities for future work.

6 Future Work

Type components and subtyping. Before to incorporate mixin modules into a practical language, we have to refine our type system in at least two respects. First, we have to design an extended version of *Mix* including ML-style user-defined type components and data types. This task should benefit from recent advances in the design of recursive module systems [6, 7]. Second, we have to enrich our type system with a notion of subtyping over mixin modules. Indeed, it might turn out too restrictive for a module system to require, as *Mix* does, a definition filling a declaration of type τ to have exactly type τ.

Compilation. The *Mix* language features anonymous definitions, and thus the compilation scheme for mixin modules proposed by Hirschowitz and Leroy [14] does not apply. A possible extension of this scheme to anonymous definitions is sketched in later work [12], but not formalized. This extension might apply to *Mix*. However, it should be possible to do better than this, by taking advantage of the more rigid structure of *Mix* mixin modules.

Other definitions of composition. The composition operator of *Mix* is somewhat arbitrary. This gives the idea to explore other definitions, perhaps more flexible. Ideally, the system should be parameterized over the notion of composition.

References

[1] D. Ancona, S. Fagorzi, E. Moggi, E. Zucca. Mixin modules and computational effects. In *Int'l Col. on Automata, Lang. and Progr.,* 2003.

[2] D. Ancona, E. Zucca. A calculus of module systems. *J. Func. Progr.,* 12(2), 2002.

[3] A. W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[4] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, 1992.

[5] L. Cardelli. Program fragments, linking, and modularization. In *24th symp. Principles of Progr. Lang.* ACM Press, 1997.

[6] K. Crary, R. Harper, S. Puri. What is a recursive module? In *Prog. Lang. Design and Impl.* ACM Press, 1999.

[7] D. R. Dreyer, R. Harper, K. Crary. Toward a practical type theory for recursive modules. Technical Report CMU-CS-01-112, Carnegie Mellon University, Pittsburgh, PA, 2001.

[8] D. Duggan, C. Sourelis. Mixin modules. In *Int. Conf. on Functional Progr.* ACM Press, 1996.

[9] M. Flatt. PLT MzScheme: language manual. Technical Report TR97-280, Rice University, 1997.

[10] M. Flatt, M. Felleisen. Units: cool modules for HOT languages. In *Prog. Lang. Design and Impl.* ACM Press, 1998.

[11] R. Harper, M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st symp. Principles of Progr. Lang.* ACM Press, 1994.

[12] T. Hirschowitz. *Modules mixins, modules et récursion étendue en appel par valeur.* PhD thesis, University of Paris VII, 2003.

[13] T. Hirschowitz. Rigid mixin modules. Research report RR2003-46, ENS Lyon, 2003.

[14] T. Hirschowitz, X. Leroy. Mixin modules in a call-by-value setting. In D. Le Métayer, ed., *Europ. Symp. on Progr.,* vol. 2305 of *LNCS,* 2002.

[15] T. Hirschowitz, X. Leroy, J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *Princ. and Practice of Decl. Prog.* ACM Press, 2003.

[16] T. Hirschowitz, X. Leroy, J. B. Wells. A reduction semantics for call-by-value mixin modules. Research report RR-4682, INRIA, 2003.

[17] X. Leroy, D. Doligez, J. Garrigue, J. Vouillon. The Objective Caml system. Software and documentation available on the Web, http://caml.inria.fr/, 1996–2003.

[18] E. Machkasova, F. A. Turbak. A calculus for link-time compilation. In *Europ. Symp. on Progr.,* vol. 1782 of *LNCS.* Springer-Verlag, 2000.

[19] R. Milner, M. Tofte, D. MacQueen. *The Definition of Standard ML.* The MIT Press, 1990.

[20] J. B. Wells, R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Europ. Symp. on Progr.,* vol. 1782 of *LNCS.* Springer-Verlag, 2000.

Semantics of Linear Continuation-Passing in Call-by-Name

Masahito Hasegawa[1,2]

[1] Research Institute for Mathematical Sciences, Kyoto University
hassei@kurims.kyoto-u.ac.jp
[2] PRESTO, Japan Science and Technology Agency

Abstract. We propose a semantic framework for modelling the linear usage of continuations in typed call-by-name programming languages. On the semantic side, we introduce a construction for *categories of linear continuations,* which gives rise to cartesian closed categories with "linear classical disjunctions" from models of intuitionistic linear logic with sums. On the syntactic side, we give a simply typed call-by-name $\lambda\mu$-calculus in which the use of names (continuation variables) is restricted to be linear. Its semantic interpretation into a category of linear continuations then amounts to the call-by-name continuation-passing style (CPS) transformation into a linear lambda calculus with sum types. We show that our calculus is sound for this CPS semantics, hence for models given by the categories of linear continuations.

1 Introduction

1.1 Linearly Used Continuations

Recent work on *linearly used continuations* by Berdine, O'Hearn, Reddy and Thielecke [7,8] points out the advantage of looking at the linear usage of *continuations* in programming languages. They observe:

> … in the many forms of control, continuations are used *linearly.* This is true for a wide range of effects, including procedure call and return, exceptions, `goto` statements, and coroutines.

They then propose linear type systems (based on a version of intuitionistic linear logic [13,2,3]) for capturing the linear usage of continuations, where the linear types are used for typing the target codes of continuation-passing style (CPS) transforms, rather than the source (ML or Scheme, for example) programs. Several "good" examples are shown to typecheck, while examples which duplicate continuations do not. An instance of such situations is found in a recent work on axiomatizing delimited continuations [19] where the linear usage of metacontinuations is crucial.

Motivated by Berdine et al.'s work, in a previous paper [14] we have developped a semantic framework for linearly used continuations (and more generally linearly used effects) in typed call-by-value (CBV) programming languages in

© Springer-Verlag Berlin Heidelberg 2004

terms of models of linear type theories. In particular, the CBV CPS transformation is naturally derived as an instance of general monadic transformation into the linear lambda calculus in this framework, and we have shown that the CPS transformation enjoys good properties, most notably the full completeness ("no-junk property"). Further results including a fully abstract game semantics have been given by Laird [20]. Thus the semantic analysis on linear CPS in the CBV setting has been shown fruitful and successful to some extent.

The present paper proposes an analogous approach for linearly used continuations in *call-by-name* setting. Thus we first seek for the semantic construction which gives a model capturing the linearity of the usage of continuations from a model of linear type theory, and then extract the call-by-name CPS transformation into the linear lambda calculus from the construction. In this way we provide sound models of the call-by-name $\lambda\mu$-calculus [23] in which the use of names (continuation variables) is restricted to be linear. Proof theoretically, this restriction prohibits us to write programs (proofs) of many of "classical" types, because the disjunction type is used only linearly. We still have the excluded middle $\neg A \vee A$ (because it is isomorphic to $A \Rightarrow A$ in this world), but not the double-negation elimination $\neg\neg A \Rightarrow A$ (equivalently $\neg\neg\neg A \vee A$) in general. This means that the typing for linearly used continuations is placed somewhere between the intuitionistic and classical ones [1].

1.2 Semantic Construction: Categories of Linear Continuations

The central semantic construction in this work, though rather simple and possibly folklore among specialists, is that of *categories of linear continuations,* which can be considered as a generalization of two well-known constructions of cartesian closed categories:

1. The semantic counterpart of the *(call-by-name) double-negation translation from classical logic to intuitionistic logic:* we construct a cartesian closed category from a cartesian closed category with sums as the opposite of the Kleisli category of the "continuation monad" $((-) \rightarrow R) \rightarrow R$, also known as the *category of continuations* [16,26].
2. The semantic counterpart of the *Girard translation from intuitionistic logic to linear logic:* we construct a cartesian closed category from a model of linear logic as the co-Kleisli category of the comonad $!(-) = ((-) \rightarrow \bot) \multimap \bot$ (where we assume the presence of products) — equivalently as the opposite of the Kleisli category of the monad $?(-) = ((-) \multimap \bot) \rightarrow \bot$ (where we need sums).

The view of regarding modalities ! and ? as expressing "linearly used continuations" and "linearly defined continuations" has been emphasized in our previous work [15] (and also implicit in Filinski's work [12]), and it helps us to understand these two situations as instances of a single setting. Starting from a model of linear logic with sums, we construct a cartesian closed category as the opposite of the Kleisli category of the ?-like monad $T(-) = ((-) \multimap R) \rightarrow R$.

One technically interesting point is that monads of this form are in general *not* strong — they only have "strength with respect to !" [10]. Thus they seem less useful in the call-by-value setting (because a monad needs to be strong for interpreting a reasonable "notion of computation" in the sense of Moggi [22]). This also implies that the induced operators on objects for the "linear classical disjunction"does not form a premonoidal structure [24] — the object function $A \vee (-)$ does not extend to a functor.

1.3 Organization of This Paper

This article is organized as follows. In Sect. 2 we recall the semantics and syntax of the linear lambda calculus which serves as the target of our CPS transformation. Sect. 3 introduces the construction of categories of linear continuations. In Sect. 4 we consider the $\lambda\mu$-calculus with linear controls, and spell out the CPS transformation derived from the semantic construction of the last section. Sect. 5 concludes the paper. Appendices summarize the linear lambda calculus DILL and the notion of !-strong monads.

2 Preliminaries

2.1 Categorical Models of Linear Logic

We describe models of linear logic in terms of symmetric monoidal closed categories with additional structure – suitable comonad for modelling the modality "of course"!, and finite products/coproducts for modelling additives, and a dualising object (hence $*$-autonomous structure [4,5,25]) for modelling the duality of classical linear logic. For reference, we shall give a compact description of the comonads to be used below, due to Hyland and Schalk (which is equivalent to Bierman's detailed definition [9], and also to the formulation based on symmetric monoidal adjunctions [6,3]).

Definition 1 (linear exponential comonad [17]). *A symmetric monoidal comonad* $! = (!, \varepsilon, \delta, m_{A,B}, m_I)$ *on a symmetric monoidal category* $\mathcal{C}$ *is called a* linear exponential comonad *when the category of its coalgebras is a category of commutative comonoids – that is:*

- *there are specified monoidal natural transformations* $e_A : !A \to I$ *and* $d_A : !A \to !A \otimes !A$ *which form a commutative comonoid* $(!A, e_A, d_A)$ *in* $\mathcal{C}$ *and also are coalgebra morphisms from* $(!A, \delta_A)$ *to* (I, m_I) *and* $(!A \otimes !A, m_{!A,!A} \circ (\delta_A \otimes \delta_A))$ *respectively, and*
- *any coalgebra morphism from* $(!A, \delta_A)$ *to* $(!B, \delta_B)$ *is also a comonoid morphism from* $(!A, e_A, d_A)$ *to* $(!B, e_B, d_B)$.

2.2 Dual Intuitionistic Linear Logic

The target calculus we will make use of is the multiplicative exponential fragment of intuitionistic linear logic, formulated as a linear lambda calculus summarized in Appendix. Our presentation is based on a dual-context type system for intuitionistic linear logic (called DILL) due to Barber and Plotkin [2,3]. In this formulation of the linear lambda calculus, a typing judgement takes the form $\Gamma\ ;\ \Delta \vdash M : \tau$ in which Γ represents an intuitionistic (or additive) context whereas Δ is a linear (multiplicative) context. It has been shown that symmetric monoidal closed categories with linear exponential comonad provide a sound and complete class of categorical models of DILL [3].

In the sequel, it turns out to be convenient to introduce syntax sugars for "intuitionistic" or "non-linear" function type

$$\begin{array}{rcl} \tau_1 \to \tau_2 & \equiv & !\tau_1 \multimap \tau_2 \\ \lambda x^{\tau}.M & \equiv & \lambda y^{!\tau}.\mathsf{let}\ !x^{\tau}\ \mathsf{be}\ y\ \mathsf{in}\ M \\ M^{\tau_1 \to \tau_2} @ N^{\tau_1} & \equiv & M\,(!N) \end{array}$$

which enjoy the following typing derivations.

$$\frac{\Gamma, x : \tau_1\ ;\ \Delta \vdash M : \tau_2}{\Gamma\ ;\ \Delta \vdash \lambda x^{\tau_1}.M : \tau_1 \to \tau_2} \qquad \frac{\Gamma\ ;\ \Delta \vdash M : \tau_1 \to \tau_2 \quad \Gamma\ ;\ \emptyset \vdash N : \tau_1}{\Gamma\ ;\ \Delta \vdash M @ N : \tau_2}$$

As one expects, the usual $\beta\eta$-equalities $(\lambda x.M) @ N = M[N/x]$ and $\lambda x.M @ x = M$ (with x not free in M) are easily provable from the axioms of DILL. (In fact it is possible to have them as primitives rather than derived constructs [7,8,15].)

In addition to the constructs of DILL, we need to deal with (additive) sum types. Here we employ the fairly standard syntax:

$$\frac{\Gamma\ ;\ \Delta \vdash M : 0}{\Gamma\ ;\ \Delta \vdash \mathsf{abort}_{\sigma} M : \sigma}\ (0\,\mathrm{E})$$

$$\frac{\Gamma\ ;\ \Delta \vdash M : \sigma}{\Gamma\ ;\ \Delta \vdash \mathsf{inl}_{\sigma,\tau} M : \sigma \oplus \tau}\ (\oplus \mathrm{I}_L) \qquad \frac{\Gamma\ ;\ \Delta \vdash N : \tau}{\Gamma\ ;\ \Delta \vdash \mathsf{inr}_{\sigma,\tau} N : \sigma \oplus \tau}\ (\oplus \mathrm{I}_R)$$

$$\frac{\Gamma\ ;\ \Delta_1 \vdash L : \sigma \oplus \tau \quad \Gamma\ ;\ \Delta_2, x : \sigma \vdash M : \theta \quad \Gamma\ ;\ \Delta_2, y : \tau \vdash N : \theta}{\Gamma\ ;\ \Delta_1 \sharp \Delta_2 \vdash \mathsf{case}\ L\ \mathsf{of}\ \mathsf{inl}\, x^{\sigma} \mapsto M \,\|\, \mathsf{inr}\, y^{\tau} \mapsto N : \theta}\ (\oplus \mathrm{E})$$

3 Categories of Linear Continuations

Let $\mathcal{C}$ be a symmetric monoidal closed category with a linear exponential comonad ! and finite coproducts (we write 0 for an initial object and $\oplus$ for binary coproducts). Fix an object R, and define a category $R^{\mathcal{C}}$ as follows: $R^{\mathcal{C}}$'s objects are the same as those of $\mathcal{C}$, and arrows are given by

$$R^{\mathcal{C}}(A, B) = \mathcal{C}(!(A \multimap R), B \multimap R).$$

The identities and compositions in $R^{\mathcal{C}}$ are inherited from the co-Kleisli category $\mathcal{C}_!$ of the comonad ! (so, up to equivalence, $R^{\mathcal{C}}$ can be considered as the full subcategory of $\mathcal{C}_!$ whose objects are of the form $A \multimap R$).

As easily seen, $R^{\mathcal{C}}(A,B) \simeq \mathcal{C}(B, !(A \multimap R) \multimap R)$, thus $R^{\mathcal{C}}$ is isomorphic to the opposite of the Kleisli category of the monad $TX = !(X \multimap R) \multimap R$ on $\mathcal{C}$. Note that this monad is *not* necessarily strong – but it is strong with respect to ! (i.e., has a restricted form of strength $!A \otimes TX \to T(!A \otimes X)$) – see Appendix for the definition of !-strong monads. This notion is introduced by Blute et al. [10] for axiomatising the exponential "why not" ?. A !-strong monad may not be strong, though it induces a strong monad on the co-Kleisli category $\mathcal{C}_!$ [14].

Proposition 1. *The monad* $TX = !(X \multimap R) \multimap R$ *on* $\mathcal{C}$ *is* !*-strong.*

In terms of DILL, the !-strength is represented as follows.

$$a : A \; ; \; m : (X \multimap R) \to R \vdash \lambda k^{(!A \otimes X) \multimap R}.m \, @ \, (\lambda x^X.k\,(!a \otimes x)) \\ : ((!A \otimes X) \multimap R) \to R$$

We shall note the non-linear use of the variable $a : A$.

Note that the exponential $?(-) = !((-) \multimap \bot) \multimap \bot$ is a particular instance of this — see Example 2 below. Another typical example of !-strong (but not necessarily strong) monads is the exception monad $X \mapsto X \oplus E$.

3.1 Cartesian Closure

A category of linear continuations has sufficient structures for modelling the simply typed lambda calculus [21]:

Proposition 2. $R^{\mathcal{C}}$ *is a cartesian closed category.*

Proof: Define

$$\top = 0 \quad A \wedge B = A \oplus B \quad A \Rightarrow B = !(A \multimap R) \otimes B$$

We shall see that $\top$ is a terminal object, $A \wedge B$ is a binary product of A and B, and $A \Rightarrow B$ is an exponential of B by A in $R^{\mathcal{C}}$.

$$\begin{array}{rcl} R^{\mathcal{C}}(A,\top) & = & \mathcal{C}(!(A \multimap R), 0 \multimap R) \\ & \simeq & \mathcal{C}(0, !(A \multimap R) \multimap R) \\ & \simeq & 1 \end{array}$$

$$\begin{array}{rcl} R^{\mathcal{C}}(A, B \wedge C) & = & \mathcal{C}(!(A \multimap R), (B \oplus C) \multimap R) \\ & \simeq & \mathcal{C}(B \oplus C, !(A \multimap R) \multimap R) \\ & \simeq & \mathcal{C}(B, !(A \multimap R) \multimap R) \times \mathcal{C}(C, !(A \multimap R) \multimap R) \\ & \simeq & \mathcal{C}(!(A \multimap R), B \multimap R) \times \mathcal{C}(!(A \multimap R), C \multimap R) \\ & = & R^{\mathcal{C}}(A,B) \times R^{\mathcal{C}}(A,C) \end{array}$$

$$\begin{array}{rcl} R^{\mathcal{C}}(A \wedge B, C) & = & \mathcal{C}(!((A \oplus B) \multimap R), C \multimap R) \\ & \simeq & \mathcal{C}(!(A \multimap R) \otimes !(B \multimap R), C \multimap R) \\ & \simeq & \mathcal{C}(!(A \multimap R), (!(B \multimap R) \otimes C) \multimap R) \\ & = & R^{\mathcal{C}}(A, B \Rightarrow C) \end{array}$$

Here we use an isomorphism $!((A\oplus B)\multimap R)\simeq !(A\multimap R)\otimes !(B\multimap R)$ which can be thought as an instance of the "Seely isomorphism"$!(X\&Y)\simeq !X\otimes !Y$ [25,9] as we have a product diagram $A\multimap R\xleftarrow{\mathsf{inl}\multimap R}(A\oplus B)\multimap R\xrightarrow{\mathsf{inr}\multimap R}B\multimap R$. Since a linear exponential comonad ! arises from a symmetric monoidal adjunction from a category with finite products [9,6,3], it follows that ! sends a product of X and Y (if exists) to $!X\otimes !Y$ up to coherent isomorphism. □

3.2 Disjunctions

It is natural to define the (linear) disjunctions on $R^{\mathcal{C}}$ as

$$\bot = I \qquad A\vee B = A\otimes B$$

which satisfy the isomorphisms one would expect for "classical" disjunctions:

Proposition 3. *The following isomorphisms exist:*

$$\begin{array}{rcl} A\Rightarrow B & \simeq & (A\Rightarrow\bot)\vee B \\ A\Rightarrow(B\vee C) & \simeq & (A\Rightarrow B)\vee C \end{array}$$

However, they are *not* even premonoidal (because the monad T is *not* strong). The functor $A\otimes(-)$ on $\mathcal{C}$ does not give rise to a functor on $R^{\mathcal{C}}$.

We also note that these linear disjunctions do not give weak coproducts in general. For instance $\bot$ is not a weak initial object:

$$R^{\mathcal{C}}(\bot,X) = \mathcal{C}(!(I\multimap R),X\multimap R) \simeq \mathcal{C}(X,!R\multimap R) = \mathcal{C}(X,R\to R)$$

Hence we can define the canonical map from $\bot$ to only objects of the form $!X$.

3.3 Examples

As mentioned in the introduction, the categories of linear continuations subsume two well-known constructions of cartesian closed categories: one for the call-by-name double-negation translation from classical logic to intuitionistic logic, and the other for (the dual of) the Girard translation from intuitionistic logic to linear logic. For the former, it suffices to simply trivialize the linearity. For the latter, we let the response object R be the dualising object (linear falsity type) $\boldsymbol{\bot}$.[3]

Example 1 (Categories of continuations). Let $\mathcal{C}$ be a cartesian closed category with finite coproducts and an object R. By taking the identity comonad as the linear exponential comonad, we have a sufficient structure for constructing a category $R^{\mathcal{C}}$ of (linear) continuations. Its objects are the same as $\mathcal{C}$, with $R^{\mathcal{C}}(A,B)=(R^A,R^B)$, together with the terminal object 0, binary product $A\oplus B$ and exponential $R^A\times B$. This is exactly the category of continuations [16,26]. Note that, in this case, the monad T is the standard continuation monad and is strong, hence the classical disjunction is premonoidal.

[3] This should not be confused with the classical falsity $\bot$ introduced in the last section. In this paper we use $\bot$ and $\boldsymbol{\bot}$ for the classical falsity and linear falsity (dualising object) respectively.

Example 2 (Girard translation). Suppose that $\mathcal{C}$ is $*$-autonomous [4], thus has a dualising object $\bot$ and can model classical linear logic [25,5]. We note that its opposite $\mathcal{C}^{\mathrm{op}}$ is also $*$-autonomous, with a linear exponential comonad $?X = !(X \multimap \bot) \multimap \bot$. By letting R be $\bot$, we have

$$\bot^{\mathcal{C}^{\mathrm{op}}}(A,B) \simeq \mathcal{C}^{\mathrm{op}}(?(A \multimap \bot), B \multimap \bot) \simeq \mathcal{C}^{\mathrm{op}}(B, !A) = \mathcal{C}(!A, B) = \mathcal{C}_!(A,B).$$

Thus the derivation of the cartesian closure of $\bot^{\mathcal{C}^{\mathrm{op}}}$ is exactly the well-known "decomposition" $A \Rightarrow B = !A \multimap B$ (given a symmetric monoidal closed category $\mathcal{C}$ with a linear exponential comonad ! and finite products, the co-Kleisli category $\mathcal{C}_!$ is cartesian closed). Sec.4.5 gives a syntactic interpretation of this observation.

Of course, there are many categories of linear continuations which do not fall into these two extreme cases. For instance:

Example 3. Let $\mathcal{C}$ be the category of ω-cpo's (with bottom) and strict continuous functions, ! be the lifting, and R be any object. The category $R^{\mathcal{C}}$ has the same objects as $\mathcal{C}$, but its morphism from A to B is a (possibly non-strict) continuous function between the strict-function spaces $A \multimap R$ and $B \multimap R$.

3.4 Discussion: Towards Direct-Style Models

Ideally, we would like to find a direct axiomatization of the categories of linear continuations as cartesian closed categories with extra structure — as neatly demonstrated in the non-linear case by Selinger as control categories and its structural theorem with respect to categories of continuations [26]. The main difficulty in our linear case is that the linear classical disjunction is no longer premonoidal, and we do not know how to axiomatize them. So there seems no obvious way to adopt Selinger's work to define a notion of "linear control categories".

But there still are some hope: we can consider the category $R^{\mathcal{C}^{\mathrm{lin}}}$ defined by $R^{\mathcal{C}^{\mathrm{lin}}}(A,B) = \mathcal{C}(A \multimap R, B \multimap R)$, which can be regarded as a lluf subcategory of linear maps in $R^{\mathcal{C}}$ (provided the counit is epi). The disjunctions do form a symmetric premonoidal structure on $R^{\mathcal{C}^{\mathrm{lin}}}$. Moreover, the category $R^{\mathcal{C}^{\mathrm{lin}}}$ has finite products and the inclusion from $R^{\mathcal{C}^{\mathrm{lin}}}$ to $R^{\mathcal{C}}$ preserves them.

So it might be natural to formulate the structure directly as a cartesian closed category together with a lluf subcategory with finite products (preserved by the inclusion) and distributive symmetric premonoidal products (but not with codiagonals as required for control categories), satisfying certain coherence conditions — e.g. on the isomorphism $A \Rightarrow (B \vee C) \simeq (A \Rightarrow B) \vee C$.

4 The $\lambda\mu$-Calculus with Linear Controls

We formulate the calculus for expressing "linearly used continuations" as a constrained $\lambda\mu$-calculus where names (continuation variables) are used and bound just linearly. Here we make use of the syntax for the simply typed $\lambda\mu$-calculus with disjunctions [26], together with a typing system which represents this linearity constraint on names.

4.1 The Calculus

Types and Terms

$$\begin{array}{rcl} \sigma & ::= & b \mid \sigma \Rightarrow \sigma \mid \top \mid \sigma \wedge \sigma \mid \bot \mid \sigma \vee \sigma \\ M & ::= & x \mid \lambda x^\sigma.M \mid M\,M \mid * \mid \langle M, M\rangle \mid \pi_1\,M \mid \pi_2\,M \mid \\ & & [\alpha]M \mid \mu\alpha^\sigma.M \mid [\alpha,\alpha]M \mid \mu(\alpha^\sigma,\alpha^\sigma).M \end{array}$$

where b ranges over base types.

Typing

$$\frac{}{\Gamma \vdash x : \sigma \mid \emptyset}\; x : \sigma \in \Gamma$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau \mid \Delta}{\Gamma \vdash \lambda x^\sigma.M : \sigma \Rightarrow \tau \mid \Delta} \qquad \frac{\Gamma \vdash M : \sigma \Rightarrow \tau \mid \Delta \quad \Gamma \vdash N : \sigma \mid \emptyset}{\Gamma \vdash M\,N : \tau \mid \Delta}$$

$$\frac{}{\Gamma \vdash * : \top \mid \emptyset} \qquad \frac{\Gamma \vdash M : \sigma \mid \Delta_1 \quad \Gamma \vdash N : \tau \mid \Delta_2}{\Gamma \vdash \langle M, N\rangle : \sigma \wedge \tau \mid \Delta_1 \sharp \Delta_2}$$

$$\frac{\Gamma \vdash M : \sigma \wedge \tau \mid \emptyset}{\Gamma \vdash \pi_1\,M : \sigma \mid \emptyset} \qquad \frac{\Gamma \vdash M : \sigma \wedge \tau \mid \emptyset}{\Gamma \vdash \pi_2\,M : \tau \mid \emptyset}$$

$$\frac{\Gamma \vdash M : \sigma \mid \Delta}{\Gamma \vdash [\alpha]M : \bot \mid \{\alpha : \sigma\} \sharp \Delta} \qquad \frac{\Gamma \vdash M : \bot \mid \alpha : \sigma, \Delta}{\Gamma \vdash \mu\alpha^\sigma.M : \sigma \mid \Delta}$$

$$\frac{\Gamma \vdash M : \sigma \vee \tau \mid \Delta}{\Gamma \vdash [\alpha,\beta]M : \bot \mid \{\alpha : \sigma, \beta : \tau\} \sharp \Delta} \qquad \frac{\Gamma \vdash M : \bot \mid \alpha : \sigma, \beta : \tau, \Delta}{\Gamma \vdash \mu(\alpha^\sigma, \beta^\tau).M : \sigma \vee \tau \mid \Delta}$$

where $\Delta_1 \sharp \Delta_2$ represents one of possible merges of Δ_1 and Δ_2 as finite lists. We assume that, when we introduce $\Delta_1 \sharp \Delta_2$, there is no name occurring both in Δ_1 and in Δ_2. We write $\emptyset$ for the empty context. Note that names cannot be free in the argument of a function application.

Example 4. The reader is invited to verify that

$$\mu(\kappa^{\sigma \Rightarrow \bot}, \alpha^\sigma).[\kappa](\lambda x^\sigma.[\alpha]x) : (\sigma \Rightarrow \bot) \vee \sigma$$

typechecks, while

$$\lambda m^{(\sigma \Rightarrow \bot) \Rightarrow \bot}.\mu\alpha^\sigma.m\,(\lambda x^\sigma.[\alpha]x) : ((\sigma \Rightarrow \bot) \Rightarrow \bot) \Rightarrow \sigma$$

does not — the name α occurs in the argument of a function m which may duplicate or discard α.

Example 5. The disjunction $(-) \vee \tau$ fails to be functorial: given a term $M : \sigma \Rightarrow \sigma'$, one might expect to have

$$M \vee \tau \;=\; \lambda z^{\sigma \vee \tau}.\mu(\alpha'^{\sigma'}, \beta^\tau).[\alpha'](M\,(\mu\alpha^\sigma.[\alpha,\beta]z)) \;:\; \sigma \vee \tau \Rightarrow \sigma' \vee \tau$$

which is illegal because M may not use β linearly.

4.2 The Call-by-Name CPS Interpretation

The cartesian closed structure of the category of linear continuations motivates the following interpretation of the arrow type

$$[\![\sigma \Rightarrow \tau]\!] \;\simeq\; !([\![\sigma]\!] \multimap R) \otimes [\![\tau]\!]$$

which leads us to interpret a typing judgement as follows.

$$\begin{array}{l} [\![x_1 : \sigma_1, \ldots, x_m : \sigma_m \vdash M : \sigma \mid \alpha_1 : \tau_1, \ldots, \alpha_n : \tau_n]\!] : \\ \quad !([\![\sigma_1]\!] \multimap R) \otimes \ldots \otimes !([\![\sigma_m]\!] \multimap R) \otimes [\![\tau_1]\!] \otimes \ldots \otimes [\![\tau_n]\!] \longrightarrow [\![\sigma]\!] \multimap R \end{array}$$

Rather than describing the translation in terms of categorical combinators, below we shall give it as a CPS transform into DILL with sums. For types we have

$$\begin{array}{ccc} b^\circ = b & \top^\circ = 0 & \bot^\circ = I \\ (\sigma \Rightarrow \tau)^\circ = !(\sigma^\circ \multimap R) \otimes \tau^\circ & (\sigma \wedge \tau)^\circ = \sigma^\circ \oplus \tau^\circ & (\sigma \vee \tau)^\circ = \sigma^\circ \otimes \tau^\circ \end{array}$$

and for terms

$$\begin{array}{rcl} x^\circ &=& x \\ (\lambda x^\sigma.M^\tau)^\circ &=& \lambda(!x^{\sigma^\circ \multimap R} \otimes k^{\tau^\circ}).M^\circ\, k \\ (M^{\sigma\Rightarrow\tau}\, N^\sigma)^\circ &=& \lambda k^{\tau^\circ}.M^\circ\,(!N^\circ \otimes k) \\ *^\circ &=& \lambda k^0.\mathsf{abort}_R\, k \\ \langle M^\sigma, N^\tau\rangle^\circ &=& \lambda k^{\sigma^\circ \oplus \tau^\circ}.\mathsf{case}\; k \;\mathsf{of}\; \mathsf{inl}(x) \mapsto M^\circ x \parallel \mathsf{inr}(y) \mapsto N^\circ y \\ (\pi_1 M^{\sigma\wedge\tau})^\circ &=& \lambda k^{\sigma^\circ}.M^\circ\,(\mathsf{inl}\, k) \\ (\pi_2 M^{\sigma\wedge\tau})^\circ &=& \lambda k^{\tau^\circ}.M^\circ\,(\mathsf{inr}\, k) \\ ([\alpha]M)^\circ &=& \lambda *^I.M^\circ\,\alpha \\ (\mu\alpha^\sigma.M)^\circ &=& \lambda \alpha^{\sigma^\circ}.M^\circ * \\ ([\alpha,\beta]M)^\circ &=& \lambda *^I.M^\circ\,(\alpha\otimes\beta) \\ (\mu(\alpha^\sigma,\beta^\tau).M)^\circ &=& \lambda(\alpha^{\sigma^\circ} \otimes \beta^{\tau^\circ}).M^\circ * \end{array}$$

Also we use some "pattern matching binding", e.g. $\lambda(x^\sigma \otimes y^\tau).M$ as a shorthand for $\lambda z^{\sigma\otimes\tau}.\mathsf{let}\; x^\sigma \otimes y^\tau \;\mathsf{be}\; z \;\mathsf{in}\; M$, and $\lambda *^I .M$ for $\lambda z^I.\mathsf{let} * \mathsf{be}\; z \;\mathsf{in}\; M$. A typing judgement

$$x_1 : \sigma_1, \ldots, x_m : \sigma_m \vdash M : \sigma \mid \alpha_1 : \tau_1, \ldots, \alpha_n : \tau_n$$

is sent to

$$x_1 : \sigma_1^\circ \multimap R, \ldots, x_m : \sigma_m^\circ \multimap R \;;\; \alpha_1 : \tau_1^\circ, \ldots, \alpha_n : \tau_n^\circ \vdash M^\circ : \sigma^\circ \multimap R.$$

Note that, if we ignore all the linearity information, this is precisely the same as the call-by-name CPS transformation of Selinger [26].

Example 6. The well-typed term

$$\mu(\kappa^{\sigma\Rightarrow\bot}, \alpha^\sigma).[\kappa](\lambda x^\sigma.[\alpha]x) : (\sigma \Rightarrow \bot) \vee \sigma$$

is sent to

$$\lambda((!x^{\sigma^\circ \multimap R} \otimes *^I) \otimes \alpha^{\sigma^\circ}).x\,\alpha : ((!(\sigma^\circ \multimap R) \otimes I) \otimes \sigma^\circ) \multimap R$$

which essentially agrees with the transform of the identity function

$$(\lambda x^\sigma.x)^\circ = \lambda(!x^{\sigma^\circ \multimap R} \otimes k^{\sigma^\circ}).x\,k : (!(\sigma^\circ \multimap R) \otimes \sigma^\circ) \multimap R.$$

4.3 Axioms and Soundness

It is routine to see that this CPS transform validates the $\beta\eta$-equalities of the simply typed lambda calculus with products (this is a consequence of the cartesian closedness of the categories of linear continuations). In fact all axioms for the call-by-name $\lambda\mu$-calculus (as given by Selinger [26]) are valid, except the non-linear axiom $(\beta_\perp)$: $[\alpha^\perp]M = M$ which is replaced by its linear variant $\mu\alpha^\perp.C[[\alpha]M] = C[M]$ below.

Axioms

$$\begin{array}{lll}
(\beta_\Rightarrow) & (\lambda x.M)\,N & = M[N/x] \\
(\eta_\Rightarrow) & \lambda x.M\,x & = M[N/x] \quad (x \notin FV(M)) \\
(\beta_\wedge) & \pi_i\,\langle M_1, M_2\rangle & = M_i \\
(\eta_\wedge) & \langle \pi_1\,M, \pi_2\,M\rangle & = M \\
(\eta_\top) & * & = M \\
\\
(\beta_\mu) & [\alpha'](\mu\alpha.M) & = M[\alpha'/\alpha] \\
(\eta_\mu) & \mu\alpha.[\alpha]M & = M \\
(\beta_\vee) & [\alpha', \beta'](\mu(\alpha,\beta).M) & = M[\alpha'/\alpha, \beta'/\beta] \\
(\eta_\vee) & \mu(\alpha,\beta).[\alpha,\beta]M & = M \\
(\beta_\perp) & \mu\alpha^\perp.C[[\alpha]M] & = C[M] \\
(\zeta_\Rightarrow) & (\mu\alpha^{\sigma\Rightarrow\tau}.C[[\alpha]M])\,N & = \mu\beta^\tau.C[[\beta](M\,N)] \\
(\zeta_\wedge) & \pi_i\,(\mu\alpha^{\sigma_1\wedge\sigma_2}.C[[\alpha]M]) & = \mu\beta^{\sigma_i}.C[[\beta](\pi_i\,M)] \\
(\zeta_\vee) & [\alpha,\beta](\mu\delta.C[[\delta]M]) & = C[[\alpha,\beta]M]
\end{array}$$

Theorem 1. *The CPS translation is sound:* $\Gamma \vdash M = N : \sigma \mid \Delta$ *implies* $\Gamma^\circ \multimap R \,;\, \Delta^\circ \vdash M^\circ = N^\circ : \sigma^\circ \multimap R$.

Corollary 1. *The interpretation of the calculus into any symmetric monoidal closed category with linear exponential comonad and sums is sound:* $\Gamma \vdash M = N : \sigma \mid \Delta$ *implies* $[\![\Gamma \vdash M : \sigma \mid \Delta]\!] = [\![\Gamma \vdash N : \sigma \mid \Delta]\!]$.

4.4 Discussion: Completeness

We conjecture that the equational theory of the calculus given above is not just sound but also complete for the CPS semantics, hence also the models given by categories of linear continuations. (This should be the case, as the usual (non-linear) $\lambda\mu$-calculus is likely to be a conservative extension of our calculus and its term model gives rise to a category of continuations [16,26], hence a complete model.)

The main problem for showing the completeness, however, is that the types and terms are not sufficiently rich to give rise to a category of linear continuations as the term model; it is not very clear how to derive a base category and the response object R, as well as the linear exponential comonad !. This also suggests that the calculus may not be sufficient for explaining the nature of linear continuation-passing — there seem some room for further improvement.

4.5 Girard Translation as a CPS Transformation

In Example 2, we have noted that the standard construction of cartesian closed categories from models of linear logic can be regarded as an instance of our construction of categories of linear continuations. This means that Girard translation from intuitionistic logic into the (classical) linear logic can also be derived as an instance of our CPS transformation and extends to our $\lambda\mu$-calculus — in this reading, it really is a simply typed lambda calculus enriched with par's. The derived translation (obtained by taking the opposite of the term model of the linear lambda calculus and then letting R be $\bot$) is straightforwardly described as follows, using the linear lambda calculus DCLL [15] as the target. For types: $b^\circ = b$, $\top^\circ = \top$, $(\sigma \wedge \tau)^\circ = \sigma^\circ \,\&\, \tau^\circ$, $(\sigma \Rightarrow \tau)^\circ = \sigma^\circ \to \tau^\circ$, $\bot^\circ = \bot$ and $(\sigma \vee \tau)^\circ = \sigma^\circ \parr \tau^\circ = (\sigma^\circ \multimap \bot) \multimap (\tau^\circ \multimap \bot) \multimap \bot$. For terms, we have $x^\circ = x$, $(\lambda x.M)^\circ = \boldsymbol{\lambda} x.M^\circ$, $(M\,N)^\circ = M^\circ \,@\, N^\circ$, $*^\circ = \langle\rangle$, $\langle M, N\rangle^\circ = \langle M^\circ, N^\circ\rangle$, $(\pi_1\,M)^\circ = \mathsf{fst}\,M^\circ$, $(\pi_2\,M)^\circ = \mathsf{snd}\,M^\circ$, $([\alpha]M)^\circ = \alpha\,M^\circ$, $(\mu\alpha.M)^\circ = \mathsf{C}\,(\lambda\alpha.M^\circ)$, $([\alpha,\beta]M)^\circ = M^\circ\,\alpha\,\beta$ and $(\mu(\alpha,\beta).M)^\circ = \lambda\alpha.\lambda\beta.M^\circ$, where the combinator $\mathsf{C}_\sigma : ((\sigma \multimap \bot) \multimap \bot) \multimap \sigma$ expresses the isomorphism from $(\sigma \multimap \bot) \multimap \bot$ to σ. A judgement $\Gamma \vdash M : \sigma \mid \Delta$ is sent to $\Gamma^\circ\ ;\ \Delta^\circ \multimap \bot \vdash M^\circ : \sigma^\circ$. The soundness of this translation is just an instance of Corollary 1.

5 Concluding Remarks

In this paper we proposed a semantic approach for linearly used continuations in call-by-name setting, and developped a relevant semantic construction and also a syntactic calculus for such linear controls, together with its CPS transformation.

However, we must say that this work is still premature — at least not as successful as the case of the call-by-value setting for now — and there remain many important issues to be addressed. Among them, we already mentioned two major problems (which are related each other): (1) the lack of direct-style models, and (2) the completeness problem. This situation is really frustrating, compared with the call-by-value setting for which we have satisfactory answers for them [14]. The non-premonoidal disjunction in particular rises serious obstacles for a direct/complete axiomatization.

Another issue which we wish to understand is how the *Filinski duality* [11,26,27] between call-by-name and call-by-value languages with first-class continuations can be related to our approach on linear controls. To sketch the overall picture, below we shall list up the interpretations of the function type $A \Rightarrow B$ in the possible combinations of linearity and non-linearity:

	call-by-name	call-by-value
non-linear lang. with non-linear control	$(A \to R) \times B$	$(B \to R) \to (A \to R)$
non-linear lang. with linear control	$!(A \multimap R) \otimes B$	$(B \to R) \multimap (A \to R)$
linear lang. with non-linear control	$(A \to R) \otimes !B$	$(B \multimap R) \to (A \multimap R)$
linear lang. with linear control	$(A \multimap R) \otimes B$	$(B \multimap R) \multimap (A \multimap R)$

The top row was studied in detail by Selinger [26]. The bottom row (purely linear setting) is more or less trivial. We are most interested in the second row, but also in the third, as there seem to exist certain dualities between the call-by-name non-linear language with linear control and the call-by-value linear language with non-linear control

$$(!(A \multimap R) \otimes B) \multimap R \simeq (A \multimap R) \rightarrow (B \multimap R),$$

and also between the call-by-value non-linear language with linear control and the call-by-name linear language with non-linear control

$$((A \rightarrow R) \otimes !B) \rightarrow R \simeq (A \rightarrow R) \multimap (B \rightarrow R).$$

The practical interest on the third row may be rather limited, but we still hope that this duality-based view provides some good insights on the nature of linear controls in call-by-name and call-by-value settings, potentially with some tie-up with other computational features such as recursion and iteration [18].

For more practical side, we have not yet demonstrated the usefulness of this approach for reasoning about call-by-name programs with first-class (linear) controls. Perhaps this also reflects our lack of experience with call-by-name programming languages with first-class control primitives whose practical advantage is, we believe, yet to be understood.

Acknowledgements I thank Oliver Danvy for asking me how linear CPS for call-by-name can be semantically understood, and anonymous reviewers for helpful comments.

References

1. Ariola, Z.M. and Herbelin, H (2003) Minimal classical logic and control operators. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP2003),* Springer Lecture Notes in Comput. Sci. **2719**, pp. 871–885.
2. Barber, A. (1997) *Linear Type Theories, Semantics and Action Calculi.* PhD Thesis ECS-LFCS-97-371, Univ. Edinburgh.
3. Barber, A. and Plotkin, G. (1997) Dual intuitionistic linear logic. Manuscript. An earlier version available as Technical Report ECS-LFCS-96-347, Univ. Edinburgh.
4. Barr, M. (1979) **-Autonomous Categories.* Springer Lecture Notes in Math. **752**.
5. Barr, M. (1991) *-autonomous categories and linear logic. *Math. Struct. Comp. Sci.* **1**, 159–178.
6. Benton, N. (1995) A mixed linear non-linear logic: proofs, terms and models (extended abstract). In *8th International Workshop on Computer Science Logic (CSL'94), Selected Papers,* Springer Lecture Notes in Comput. Sci. **933**, pp. 121–135.
7. Berdine, J., O'Hearn, P.W., Reddy, U.S. and Thielecke, H. (2001) Linearly used continuations. In *Proc. 3rd ACM SIGPLAN Workshop on Continuations (CW'01),* Technical Report No. 545, Computer Science Dept., Indiana Univ., pp. 47–54.
8. Berdine, J., O'Hearn, P.W., Reddy, U.S. and Thielecke, H. (2002) Linear continuation-passing. *Higher-Order and Symbolic Computation* **15**(2/3), 181–203.
9. Bierman, G.M. (1995) What is a categorical model of intuitionistic linear logic? In *Proc. 2nd International Conference on Typed Lambda Calculi and Applications (TLCA '95),* Springer Lecture Notes in Comput. Sci. **902**, pp. 78–93.

10. Blute, R., Cockett, J.R.B. and Seely, R.A.G. (1996) ! and ? - storage as tensorial strength. *Mathematical Structures in Computer Science* **6**(4), 313–351.
11. Filinski, A. (1989) Declarative continuations: an investigation of duality in programming language semantics. In *Proc. Category Theory and Computer Science,* Springer Lecture Notes in Comput. Sci. **389**, pp. 224–249.
12. Filinski, A. (1992) Linear continuations. In *Proc. 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92),* pp. 27–38.
13. Girard, J.-Y. (1987) Linear logic. *Theoret. Comp. Sci.* **50**, 1–102.
14. Hasegawa, M. (2002) Linearly used effects: monadic and CPS transformations into the linear lambda calculus. In *Proc. 6th International Symposium on Functional and Logic Programming (FLOPS2002),* Springer Lecture Notes in Comput. Sci. **2441**, pp. 167–182.
15. Hasegawa, M. (2002) Classical linear logic of implications. In *Proc. llth Annual Conference of the European Association for Computer Science Logic (CSL'02),* Springer Lecture Notes in Comput. Sci. **2471**, pp. 458–472.
16. Hofmann, M. and Streicher, T. (1997) Continuation models are universal for $\lambda\mu$-calculus. In *Proc. 12th Annual IEEE Symposium on Logic in Computer Science (LICS'97),* pp. 387–397.
17. Hyland, M. and Schalk, A. (2003) Glueing and orthogonality for models of linear logic. *Theoret. Comp. Sci.* **294**(1/2), 183–231.
18. Kakutani, Y. (2002) Duality between call-by-name recursion and call-by-value iteration. In *Proc. 11th Annual Conference of the European Association for Computer Science Logic (CSL'02),* Springer Lecture Notes in Comput. Sci. **2471**, pp. 506–521.
19. Kameyama, Y. and Hasegawa, M. (2003) A sound and complete axiomatization of delimited continuations. In *Proc. 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003),* pp. 177–188.
20. Laird, J. (2003) A game semantics of linearly used continuations. In *Proc. 6th International Conference on Foundations of Software Science and Computation Structure (FoSSaCS2003),* Springer Lecture Notes in Comput. Sci. **2620**, pp. 313–327.
21. Lambek, J. and Scott, P.J. (1986) *Introduction to Higher Order Categorical Logic.* Cambridge Studies in Advanced Mathematics **7**, Cambridge University Press.
22. Moggi, E. (1991) Notions of computation and monads. *Inform. and Comput.* **93**(1), 55–92.
23. Parigot, M. (1992) $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proc. International Conference on Logic Programming and Automated Reasoning (LPAR'92),* Springer Lecture Notes in Comput. Sci. **624**, pp. 190–201.
24. Power, A.J. and Robinson, E.P. (1997) Premonoidal categories and notions of computation. *Math. Structures Comput. Sci.* **7**(5), 453–468.
25. Seely, R.A.G. (1989) Linear logic, $*$-autonomous categories and cofree coalgebras. In *Categories in Computer Science,* AMS Contemp. Math. **92**, pp. 371–389.
26. Selinger, P. (2001) Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Math. Structures Comput. Sci.* **11**(2), 207–260.
27. Wadler, P. (2003) Call-by-value is dual to call-by-name. In *Proc. 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003),* pp. 189–202.

A Dual Intuitionistic Linear Logic

Types and Terms

$$\begin{array}{rcl} \sigma & ::= & b \mid I \mid \sigma \otimes \sigma \mid \sigma \multimap \sigma \mid {!\sigma} \\ M & ::= & x \mid * \mid \mathsf{let}\ *\ \mathsf{be}\ M\ \mathsf{in}\ M \mid M \otimes M \mid \mathsf{let}\ x^\sigma \otimes x^\sigma\ \mathsf{be}\ M\ \mathsf{in}\ M \mid \\ & & \lambda x^\sigma.M \mid MM \mid {!M} \mid \mathsf{let}\ {!x^\sigma}\ \mathsf{be}\ M\ \mathsf{in}\ M \end{array}$$

Typing

$$\frac{}{\Gamma_1, x:\sigma, \Gamma_2\ ;\ \emptyset \vdash x:\sigma}(\text{Int-Ax}) \qquad \frac{}{\Gamma\ ;\ x:\sigma \vdash x:\sigma}(\text{Lin-Ax})$$

$$\frac{}{\Gamma; \emptyset \vdash * : I}(I\text{I}) \qquad \frac{\Gamma; \Delta_1 \vdash M : I \quad \Gamma; \Delta_2 \vdash N : \sigma}{\Gamma; \Delta_1 \sharp \Delta_2 \vdash \mathsf{let}\ *\ \mathsf{be}\ M\ \mathsf{in}\ N : \sigma}(I\text{E})$$

$$\frac{\Gamma; \Delta_1 \vdash M : \sigma_1 \quad \Gamma; \Delta_2 \vdash N : \sigma_2}{\Gamma; \Delta_1 \sharp \Delta_2 \vdash M \otimes N : \sigma_1 \otimes \sigma_2}(\otimes\text{I}) \qquad \frac{\Gamma; \Delta_1 \vdash M : \sigma_1 \otimes \sigma_2 \quad \Gamma; \Delta_2, x : \sigma_1, y : \sigma_2 \vdash N : \tau}{\Gamma; \Delta_1 \sharp \Delta_2 \vdash \mathsf{let}\ x^{\sigma_1} \otimes y^{\sigma_2}\ \mathsf{be}\ M\ \mathsf{in}\ N : \tau}(\otimes\text{E})$$

$$\frac{\Gamma; \Delta, x : \sigma_1 \vdash M : \sigma_2}{\Gamma; \Delta \vdash \lambda x^{\sigma_1}.M : \sigma_1 \multimap \sigma_2}(\multimap\text{I}) \qquad \frac{\Gamma; \Delta_1 \vdash M : \sigma_1 \multimap \sigma_2 \quad \Gamma; \Delta_2 \vdash N : \sigma_1}{\Gamma; \Delta_1 \sharp \Delta_2 \vdash M\,N : \sigma_2}(\multimap\text{E})$$

$$\frac{\Gamma; \emptyset \vdash M : \sigma}{\Gamma; \emptyset \vdash {!M} : {!\sigma}}(!\text{I}) \qquad \frac{\Gamma; \Delta_1 \vdash M : {!\sigma} \quad \Gamma, x : \sigma; \Delta_2 \vdash N : \tau}{\Gamma; \Delta_1 \sharp \Delta_2 \vdash \mathsf{let}\ {!x}\ \mathsf{be}\ M\ \mathsf{in}\ N : \tau}(!\text{E})$$

where $\Delta_1 \sharp \Delta_2$ represents one of possible merges of Δ_1 and Δ_2 as finite lists.

Axioms

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{let}\ *\ \mathsf{be}\ *\ \mathsf{in}\ M & = & M & \mathsf{let}\ *\ \mathsf{be}\ M\ \mathsf{in}\ * & = & M \\ \mathsf{let}\ x \otimes y\ \mathsf{be}\ M \otimes N\ \mathsf{in}\ L & = & L[M/x, N/y] & \mathsf{let}\ x \otimes y\ \mathsf{be}\ M\ \mathsf{in}\ x \otimes y & = & M \\ (\lambda x.M)\,N & = & M[N/x] & \lambda x.M\,x & = & M \\ \mathsf{let}\ {!x}\ \mathsf{be}\ {!M}\ \mathsf{in}\ N & = & N[M/x] & \mathsf{let}\ {!x}\ \mathsf{be}\ M\ \mathsf{in}\ {!x} & = & M \end{array}$$

$$\begin{array}{rcl} C[\mathsf{let}\ *\ \mathsf{be}\ M\ \mathsf{in}\ N] & = & \mathsf{let}\ *\ \mathsf{be}\ M\ \mathsf{in}\ C[N] \\ C[\mathsf{let}\ x \otimes y\ \mathsf{be}\ M\ \mathsf{in}\ N] & = & \mathsf{let}\ x \otimes y\ \mathsf{be}\ M\ \mathsf{in}\ C[N] \\ C[\mathsf{let}\ {!x}\ \mathsf{be}\ M\ \mathsf{in}\ N] & = & \mathsf{let}\ {!x}\ \mathsf{be}\ M\ \mathsf{in}\ C[N] \end{array}$$

where $C[-]$ is a linear context (no ! binds $[-]$).

Semantics A typing judgement

$$x_1 : \sigma_1, \ldots, x_m : \sigma_m\ ;\ y_1 : \tau_1, \ldots, y_n : \tau_n \vdash M : \sigma$$

is inductively interpreted as a morphism $[\![x_1 : \sigma_1, \ldots\ ;\ y_1 : \tau_1, \ldots \vdash M : \sigma]\!]$ from $![\![\sigma_1]\!] \otimes \ldots \otimes ![\![\sigma_m]\!] \otimes [\![\tau_1]\!] \otimes \ldots \otimes [\![\tau_n]\!]$ to $[\![\sigma]\!]$ in a symmetric monoidal closed category with a linear exponential comonad !.

Proposition 4 (categorical completeness). *The equational theory of DILL is sound and complete for categorical models given by symmetric monoidal closed categories with linear exponential comonads:* $\Gamma\ ;\ \Delta \vdash M = N : \sigma$ *is provable if and only if* $[\![\Gamma\ ;\ \Delta \vdash M : \sigma]\!] = [\![\Gamma\ ;\ \Delta \vdash N : \sigma]\!]$ *holds for every such models.*

B !-Strong Monads

Let $\mathcal{D}$ be a symmetric monoidal category with a linear exponential comonad !. A monad (T, η, μ) on $\mathcal{D}$ is *!-strong* (or: *strong with respect to* ! [10]) if it is equipped with a natural transformation called *!-strength* (*strength with respect to* !)

$$\theta_{A,X} : !A \otimes TX \longrightarrow T(!A \otimes X)$$

subject to the following axioms.

$$\begin{tikzcd}
I \otimes TX \arrow[r, "\text{iso.}"] \arrow[d, "m_I \otimes TX"'] & TX \arrow[r, "\text{iso.}"] & T(I \otimes X) \arrow[d, "T(m_I \otimes X)"] \\
!I \otimes TX \arrow[rr, "\theta_{I,X}"'] & & T(!I \otimes X)
\end{tikzcd}$$

$$\begin{tikzcd}
!A \otimes (!B \otimes TX) \arrow[r, "!A\otimes\theta_{B,X}"] \arrow[d, "\text{iso.}"'] & !A \otimes T(!B \otimes X) \arrow[r, "\theta_{A,!B\otimes X}"] & T(!A \otimes (!B \otimes X)) \arrow[d, "\text{iso.}"] \\
(!A \otimes !B) \otimes TX \arrow[d, "m_{A,B} \otimes TX"'] & & T((!A \otimes !B) \otimes X) \arrow[d, "T(m_{A,B} \otimes X)"] \\
!(A \otimes B) \otimes TX \arrow[rr, "\theta_{A\otimes B,X}"'] & & T(!(A \otimes B) \otimes X)
\end{tikzcd}$$

$$\begin{tikzcd}
& !A \otimes X \arrow[dl, "!A\otimes\eta_X"'] \arrow[dr, "\eta_{!A\otimes X}"] & \\
!A \otimes TX \arrow[rr, "\theta_{A,X}"'] & & T(!A \otimes X)
\end{tikzcd}$$

$$\begin{tikzcd}
!A \otimes T^2X \arrow[r, "\theta_{A,TX}"] \arrow[d, "!A\otimes\mu_X"'] & T(!A \otimes TX) \arrow[r, "T\theta_{A,X}"] & T^2(!A \otimes X) \arrow[d, "\mu_{!A\otimes X}"] \\
!A \otimes TX \arrow[rr, "\theta_{A,X}"'] & & T(!A \otimes X)
\end{tikzcd}$$

A Direct Proof of Strong Normalization for an Extended Herbelin's Calculus

Kentaro Kikuchi*

Department of Mathematics and Informatics, Chiba University,
1-33 Yayoi-cho, Inage-ku, Chiba 263-8522, Japan
kentaro@math.s.chiba-u.ac.jp

Abstract. Herbelin presented (at CSL'94) an explicit substitution calculus with a sequent calculus as a type system, in which reduction steps correspond to cut-elimination steps. The calculus, extended with some rules for substitution propagation, simulates β-reduction of ordinary λ-calculus. In this paper we present a proof of strong normalization for the typable terms of the calculus. The proof is a direct one in the sense that it does not depend on the result of strong normalization for the simply typed λ-calculus, unlike an earlier proof by Dyckhoff and Urban.

1 Introduction

In [12], Herbelin introduced a sequent calculus in which a unique cut-free proof is associated to each normal term of the simply typed λ-calculus. This is in contrast to the usual assignment (see, e.g. [18, p. 73]) which associates several cut-free proofs to the same normal terms. Herbelin developed a term calculus whose reduction steps correspond to cut-elimination steps in the sequent calculus. Some of the cut rules introduce explicit substitution operators [1], and cut-propagation steps of cut-elimination correspond to the propagation of explicit substitutions. Herbelin proved strong normalization for the typed terms of his calculus.

However, he also observed that the reduction rules in the calculus are not enough to simulate full β-reduction (e.g., it fails to simulate the leftmost reduction). Espírito Santo [11] and Dyckhoff and Urban [10] identified a set of terms in the calculus that correspond to terms in the untyped λ-calculus, and introduced additional reduction rules needed to allow substitutions to propagate properly. Thus it turned out that ordinary λ-calculus can be completely embedded in Herbelin's calculus with the additional rules, and in the typed case β-reduction can be analyzed through cut-elimination.

On the other hand, it is reported [8] that Herbelin's sequent calculus is particularly suited to proof search used in the theory of logic programming. This means that the extended Herbelin's calculus is a promising candidate for a proof-theoretic basis for integrating functional and logic programming languages.

* This work was supported by JSPS Research Fellowships for Young Scientists and by Grant-in-Aid for JSPS Fellows.

© Springer-Verlag Berlin Heidelberg 2004

Since the extended Herbelin's calculus behaves as an explicit substitution calculus for the λ-calculus, it is expected that various techniques from the field of explicit substitutions work as well for this calculus. Dyckhoff and Urban [10] indeed proved strong normalization for the typed terms of the calculus, using the method of [4] and the result of strong normalization for the simply typed λ-calculus. Note that as shown in [15], strong normalization for typed terms of an explicit substitution calculus is not a trivial property. In fact, a careful choice of reduction rules is required for proving strong normalization of cut-elimination that simulates β-reduction.

In this paper we prove strong normalization for the typable terms of the extended Herbelin's calculus directly without using the result for the simply typed λ-calculus. Our proof is an adaptation of the reducibility method [17] to explicit substitution calculus and to Herbelin-style calculus. For the explicit substitution calculus $\lambda\mathbf{x}$ [5], such adaptations have been considered in [6,7]. Compared with their proofs, ours makes use of a more general closure condition on reducibility sets; they are closed under x-conversion whenever the term is decent (Lemma 11). This is closely related to a lemma for an inductive proof of preservation of strong normalization (PSN) as developed in [2,5]. Our method is easily applicable to the case of $\lambda\mathbf{x}$, simplifying the proofs in [6,7].

The paper is organized as follows. In Section 2 we introduce the calculus and type system. In Section 3 we consider the subset of terms that correspond to ordinary λ-terms. In Section 4 we study a subcalculus that plays an important role in our proofs. In Section 5 we explain how to simulate β-reduction in the calculus. In Section 6 we prove the main lemma, from which we derive PSN, and in Section 7 we give a reducibility proof of strong normalization using the main lemma. Finally in Section 8 we conclude and give suggestions for further work.

To save space we omit some of the proofs, but a full version with all proofs is available at `http://www.math.s.chiba-u.ac.jp/~kentaro/index.html`.

2 $\overline{\lambda}\mathbf{x}$-calculus

Table 1 presents the syntax and typing rules of $\overline{\lambda}\mathbf{x}$-calculus, which is the same as the calculus *(AO + ES + B)* in [10] and varies a little from the calculi in [11], although we mainly follow notations in the latter. The syntax of $\overline{\lambda}\mathbf{x}$ has two kinds of expressions: terms and lists of terms, ranged over by u, v, t and by l, l', respectively. The set of terms is denoted by $\mathcal{T}_{\overline{\lambda}\mathbf{x}}$ and the set of lists of terms by $\mathcal{L}_{\overline{\lambda}\mathbf{x}}$. Elements of $\mathcal{T}_{\overline{\lambda}\mathbf{x}} \cup \mathcal{L}_{\overline{\lambda}\mathbf{x}}$ are called $\overline{\lambda}\mathbf{x}$-*terms* and ranged over by a, b, c. In $a\langle x := v\rangle$, $\langle x := v\rangle$ is called an *explicit substitution* or simply substitution and v is called the *body* of the substitution. The notions of free and bound variables are defined as usual, with an additional clause that the variable x in $a\langle x := v\rangle$ binds the free occurrences of x in a. We assume the following variable convention: names of bound variables are different from the names of free variables, and, moreover, different occurrences of the abstraction operator have different binding variables. The set of free variables of a $\overline{\lambda}\mathbf{x}$-term a is denoted by $FV(a)$. The symbol $\equiv$ denotes syntactical equality modulo α-conversion.

Table 1. $\overline{\lambda}$x-calculus

$$u, v, t ::= xl \mid \lambda x.t \mid tl \mid t\langle x := v\rangle$$
$$l, l' ::= [] \mid t :: l \mid ll' \mid l\langle x := v\rangle$$

$$\frac{}{\Gamma; A \vdash [] : A}\ Ax \qquad \frac{\Gamma, x : A; A \vdash l : B}{\Gamma, x : A; - \vdash xl : B}\ Der$$

$$\frac{\Gamma; - \vdash t : A \quad \Gamma; B \vdash l : C}{\Gamma; A \supset B \vdash t :: l : C}\ L\supset \qquad \frac{\Gamma, x : A; - \vdash t : B}{\Gamma; - \vdash \lambda x.t : A \supset B}\ R\supset$$

$$\frac{\Gamma; B \vdash l : A \quad \Gamma; A \vdash l' : C}{\Gamma; B \vdash ll' : C}\ Cut_1 \qquad \frac{\Gamma; - \vdash v : A \quad \Gamma, x : A; B \vdash l : C}{\Gamma; B \vdash l\langle x := v\rangle : C}\ Cut_2$$

$$\frac{\Gamma; - \vdash t : A \quad \Gamma; A \vdash l : B}{\Gamma; - \vdash tl : B}\ Cut_3 \qquad \frac{\Gamma; - \vdash v : A \quad \Gamma, x : A; - \vdash t : B}{\Gamma; - \vdash t\langle x := v\rangle : B}\ Cut_4$$

$(Beta)$	$(\lambda x.t)(u :: l) \to t\langle x := u\rangle l$	
$(1a)$	$[]l \to l$	
$(1b)$	$(u :: l)l' \to u :: (ll')$	
$(2a)$	$[]\langle x := v\rangle \to []$	
$(2b)$	$(u :: l)\langle x := v\rangle \to u\langle x := v\rangle :: l\langle x := v\rangle$	
$(3a)$	$(xl)l' \to x(ll')$	
$(3b)$	$(\lambda y.t)[] \to \lambda y.t$	
$(4a)$	$(yl)\langle x := v\rangle \to yl\langle x := v\rangle$	$(y \not\equiv x)$
$(4b)$	$(xl)\langle x := v\rangle \to vl\langle x := v\rangle$	
$(4c)$	$(\lambda y.t)\langle x := v\rangle \to \lambda y.t\langle x := v\rangle$	
$(5a)$	$(ll')l'' \to l(l'l'')$	
$(5b)$	$(ll')\langle x := v\rangle \to l\langle x := v\rangle l'\langle x := v\rangle$	
$(5c)$	$(tl)l' \to t(ll')$	
$(5d)$	$(tl)\langle x := v\rangle \to t\langle x := v\rangle l\langle x := v\rangle$	

A *typing context,* ranged over by Γ, is a finite set of pairs $\{x_1 : A_1, \ldots, x_n : A_n\}$ where the variables are pairwise distinct. $\Gamma, x : A$ denotes the union $\Gamma \cup \{x : A\}$ where x does not appear in Γ. There are two kinds of derivable sequents: $\Gamma; - \vdash t : B$ and $\Gamma; A \vdash l : B$, both of which have a distinguished position in

Fig. 1. Key-case

$$\dfrac{\dfrac{\Gamma, x : A; - \vdash t : B}{\Gamma; - \vdash \lambda x.t : A \supset B}\,R\supset \qquad \dfrac{\Gamma; - \vdash u : A \quad \Gamma; B \vdash l : C}{\Gamma; A \supset B \vdash u :: l : C}\,L\supset}{\Gamma; - \vdash (\lambda x.t)(u :: l) : C}\,Cut_3$$

$$\to \quad \dfrac{\dfrac{\Gamma; - \vdash u : A \quad \Gamma, x : A; - \vdash t : B}{\Gamma; - \vdash t\langle x := u\rangle : B}\,Cut_4 \qquad \Gamma; B \vdash l : C}{\Gamma; - \vdash t\langle x := u\rangle l : C}\,Cut_3$$

the LHS called *stoup*. The crucial restriction of this sequent calculus is that the rule $L\supset$ introduces $A \supset B$ in the stoup and B has to be in the stoup of the right subderivation's endsequent. In the cut-free case, the last rule of the right subderivation of an instance of $L\supset$ is again $L\supset$ and so on until *Ax* is reached. This yields an assignment of a unique cut-free proof to each normal term of the simply typed λ-calculus (cf. [18, Section 6.3]).

The notion of $\overline{\lambda}\mathbf{x}$-reduction is defined by the contextual closures of all reduction rules in Table 1. We use $\to_{\overline{\lambda}\mathbf{x}}$ for one-step reduction, $\overset{+}{\to}_{\overline{\lambda}\mathbf{x}}$ for its transitive closure, and $\overset{*}{\to}_{\overline{\lambda}\mathbf{x}}$ for its reflexive transitive closure. The set of $\overline{\lambda}\mathbf{x}$-terms that are strongly normalizing with respect to $\overline{\lambda}\mathbf{x}$-reduction is denoted by $\mathcal{SN}_{\overline{\lambda}\mathbf{x}}$. These kinds of notations are also used for the notions of other reductions introduced in this paper.

The subcalculus of $\overline{\lambda}\mathbf{x}$ without the *Beta*-rule is denoted by x. This subcalculus plays an important role in this paper and is studied in Section 4.

Herbelin's original $\overline{\lambda}$-calculus [12] is essentially the calculus without the last four reduction rules in Table 1. These rules are necessary for the simulation of full β-reduction of the λ-calculus.

The reduction rules of $\overline{\lambda}\mathbf{x}$-calculus also define cut-elimination procedures for typing derivations of $\overline{\lambda}\mathbf{x}$-terms, which ensures that the subject reduction property holds in this type system. In Figure 1 we display the key-case of the cut-elimination, which corresponds to the *Beta*-rule of $\overline{\lambda}\mathbf{x}$-calculus.

3 Pure Terms

Table 2 presents the syntax of *pure terms,* which are the subset of $\overline{\lambda}\mathbf{x}$-terms that correspond to terms of ordinary λ-calculus. The grammar of pure terms is close to the inductive characterization of the set of λ-terms found, e.g. in [14]. For the definition of β-reduction on pure terms, we need meta-substitution [_/_], which requires further meta-operations {_}_ and _@_ since a $\overline{\lambda}\mathbf{x}$-term obtained by substituting a pure term for a variable is not in general a pure term. Note the

Table 2. Pure terms

$$u, v, t ::= xl \mid \lambda x.t \mid (\lambda x.t)(u :: l)$$
$$l, l' ::= [] \mid t :: l$$

$$(\beta) \qquad (\lambda x.t)(u :: l) \;\rightarrow\; \{t[u/x]\}l$$

where

$$\begin{aligned}
[]@l &=_{def} l\\
(u :: l)@l' &=_{def} u :: (l@l')\\
[][v/x] &=_{def} []\\
(u :: l)[v/x] &=_{def} u[v/x] :: l[v/x]\\
\{xl\}l' &=_{def} x(l@l')\\
\{\lambda y.t\}[] &=_{def} \lambda y.t\\
\{\lambda y.t\}(u :: l) &=_{def} (\lambda y.t)(u :: l)\\
\{(\lambda y.t)(u :: l)\}l' &=_{def} (\lambda y.t)(u :: (l@l'))\\
(yl)[v/x] &=_{def} yl[v/x] \qquad (y \not\equiv x)\\
(xl)[v/x] &=_{def} \{v\}l[v/x]\\
(\lambda y.t)[v/x] &=_{def} \lambda y.t[v/x]\\
((\lambda y.t)(u :: l))[v/x] &=_{def} (\lambda y.t[v/x])(u[v/x] :: l[v/x])
\end{aligned}$$

similarity between the definition of these meta-operations and the reduction rules of $\overline{\lambda}\mathbf{x}$-calculus. $\overline{\lambda}\mathbf{x}$-calculus can be considered in some sense a calculus making these meta-operations explicit, while usual explicit substitution calculi make the usual substitution explicit.

Here we give some properties of these meta-operations.

Lemma 1. *For any pure term $t \in \mathcal{T}_{\overline{\lambda}\mathrm{x}}$, $\{t\}[] \equiv t$.*

Lemma 2. *For any pure terms $a \in \mathcal{T}_{\overline{\lambda}\mathrm{x}} \cup \mathcal{L}_{\overline{\lambda}\mathrm{x}}$ and $v \in \mathcal{T}_{\overline{\lambda}\mathrm{x}}$, if $x \notin FV(a)$ then $a[v/x] \equiv a$.*

Lemma 3 (Substitution lemma). *For any pure terms $a \in \mathcal{T}_{\overline{\lambda}\mathrm{x}} \cup \mathcal{L}_{\overline{\lambda}\mathrm{x}}$ and $u, v \in \mathcal{T}_{\overline{\lambda}\mathrm{x}}$, if $x \notin FV(u)$ and $x \not\equiv y$ then $a[v/x][u/y] \equiv a[u/y][v[u/y]/x]$.*

Lemma 4. *For any pure terms $a, a' \in \mathcal{T}_{\overline{\lambda}\mathrm{x}} \cup \mathcal{L}_{\overline{\lambda}\mathrm{x}}$ and $v \in \mathcal{T}_{\overline{\lambda}\mathrm{x}}$, if $a \rightarrow_\beta a'$ then $a[v/x] \rightarrow_\beta a'[v/x]$.*

Table 3. Translations Ψ and Θ

$\Psi(x) =_{def} x[]$	$\Theta(xl) =_{def} \Theta'(x, l)$
$\Psi(MN) =_{def} \{\Psi(M)\}(\Psi(N) :: [])$	$\Theta(\lambda x.t) =_{def} \lambda x.\Theta(t)$
$\Psi(\lambda x.M) =_{def} \lambda x.\Psi(M)$	$\Theta((\lambda x.t)(u :: l)) =_{def} \Theta'(\lambda x.\Theta(t), u :: l)$
	$\Theta'(M, []) =_{def} M$
	$\Theta'(M, u :: l) =_{def} \Theta'(M\Theta(u), l)$

Translations between pure terms and ordinary λ-terms are given in [11] and [10] through a grammar of λ-terms that is different from the usual one. Here we define translations between λ-terms with the usual grammar and pure terms as shown in Table 3.

Proposition 1. *$\Theta \circ \Psi = id$ and $\Psi \circ \Theta = id$.*

Theorem 1.

1. *For any λ-terms M, M', if $M \rightarrow_\beta M'$ then $\Psi(M) \rightarrow_\beta \Psi(M')$.*
2. *For any pure terms $t, t' \in \mathcal{T}_{\overline{\lambda}\mathtt{x}}$, if $t \rightarrow_\beta t'$ then $\Theta(t) \rightarrow_\beta \Theta(t')$.*

It is also possible to show that the translations preserve the types of terms, defining translations on typing derivations. Later we see that β-reduction on pure terms can be simulated by $\overline{\lambda}\mathtt{x}$-reduction, thus showing how to simulate normalization in natural deduction by cut-elimination in Herbelin's sequent calculus.

4 Properties of the Subcalculus $\mathtt{x}$

In this section we study properties of the subcalculus $\mathtt{x}$ which is obtained from $\overline{\lambda}\mathtt{x}$-calculus by deleting the *Beta*-rule. In the typed case it corresponds to cut-elimination steps except the key-case. We show that the subcalculus is strongly normalizing and confluent and that its normal forms are pure terms.

Proposition 2. *The subcalculus $\mathtt{x}$ is strongly normalizing.*

Proof. The proof is by interpretation, following Appendix A of [9]. We define a function $h : \mathcal{T}_{\overline{\lambda}\mathtt{x}} \cup \mathcal{L}_{\overline{\lambda}\mathtt{x}} \longrightarrow \mathbb{N}$ as follows:

$$\begin{aligned}
h(xl) &=_{def} h(l) + 1 \\
h(\lambda x.t) &=_{def} h(t) + 1 \\
h(tl) &=_{def} 2 \times h(t) + h(l) + 1 \\
h(t\langle z := v\rangle) &=_{def} h(t) \times (3 \times h(v) + 1)
\end{aligned}$$

$$\begin{aligned}
h([]) &=_{def} 1\\
h(u :: l) &=_{def} h(u) + h(l) + 1\\
h(ll') &=_{def} 2 \times h(l) + h(l') + 1\\
h(l\langle z := v\rangle) &=_{def} h(l) \times (3 \times h(v) + 1)
\end{aligned}$$

and observe that if $a \rightarrow_{\mathrm{x}} b$ then $h(a) > h(b)$. □

Proposition 3. *The subcalculus* x *is confluent.*

Proof. By Newman's Lemma, it suffices to check the local confluence. The proof follows Appendix B of [9]. □

As a result, we can define the unique x-normal form of each $\overline{\lambda}\mathbf{x}$-term.

Definition 1. *Let $a \in \mathcal{T}_{\overline{\lambda}\mathrm{x}} \cup \mathcal{L}_{\overline{\lambda}\mathrm{x}}$. The unique* x*-normal form of a is denoted by* $\mathrm{x}(a)$.

Proposition 4. *Let $a \in \mathcal{T}_{\overline{\lambda}\mathrm{x}} \cup \mathcal{L}_{\overline{\lambda}\mathrm{x}}$. a is a pure term iff a is in* x*-normal form.*

Proof. The only if part is by induction on the structure of pure terms. We prove the if part by induction on the structure of a. Suppose that a is in x-normal form. Then by the induction hypothesis, all subterms of a are pure. Here if a is not pure then a is one of the forms $tl(\not\equiv (\lambda x.t)(u :: l))$, $t\langle x := v\rangle$, ll' and $l\langle x := v\rangle$ where t, l, v, l' are pure. In any case we see that a is an x-redex, which is a contradiction. □

The next proposition shows that the subcalculus x correctly simulates the meta-operations on pure terms.

Proposition 5. *Let t, v, l, l' be pure terms with $t, v \in \mathcal{T}_{\overline{\lambda}\mathrm{x}}$ and $l, l' \in \mathcal{L}_{\overline{\lambda}\mathrm{x}}$. Then*

1. $ll' \overset{*}{\rightarrow}_{\mathrm{x}} l@l'$,
2. $l\langle y := v\rangle \overset{*}{\rightarrow}_{\mathrm{x}} l[v/y]$,
3. $tl \overset{*}{\rightarrow}_{\mathrm{x}} \{t\}l$,
4. $t\langle y := v\rangle \overset{*}{\rightarrow}_{\mathrm{x}} t[v/y]$.

Proof. The first part is by induction on the structure of l. The third part is by a case analysis according to the form of tl. The remaining two parts are proved by simultaneous induction on the structure of l or t. □

From the above proposition we have the following lemma which allows us to reduce inference on x-normal form to inference for meta-operations on pure terms.

Lemma 5. *Let $t, v \in \mathcal{T}_{\overline{\lambda}\mathrm{x}}$ and $l, l' \in \mathcal{L}_{\overline{\lambda}\mathrm{x}}$. Then*

1. $\mathrm{x}(ll') \equiv \mathrm{x}(l)@\mathrm{x}(l')$,
2. $\mathrm{x}(l\langle y := v\rangle) \equiv \mathrm{x}(l)[\mathrm{x}(v)/y]$,
3. $\mathrm{x}(tl) \equiv \{\mathrm{x}(t)\}\mathrm{x}(l)$,
4. $\mathrm{x}(t\langle y := v\rangle) \equiv \mathrm{x}(t)[\mathrm{x}(v)/y]$.

Proof. We only consider the fourth part. Since $\mathrm{x}(t)$ and $\mathrm{x}(v)$ are pure terms, we have $\mathrm{x}(t)\langle y := \mathrm{x}(v)\rangle \overset{*}{\rightarrow}_{\mathrm{x}} \mathrm{x}(t)[\mathrm{x}(v)/y]$ by Proposition 5 (4). Hence, $\mathrm{x}(t\langle y := v\rangle) \equiv \mathrm{x}(\mathrm{x}(t)\langle y := \mathrm{x}(v)\rangle) \equiv \mathrm{x}(\mathrm{x}(t)[\mathrm{x}(v)/y]) \equiv \mathrm{x}(t)[\mathrm{x}(v)/y]$. □

5 Simulation of β-reduction

Now we are in a position to show that $\overline{\lambda}\mathtt{x}$-reduction simulates β-reduction.

Theorem 2. *For any pure terms $a, b \in \mathcal{T}_{\overline{\lambda}\mathtt{x}} \cup \mathcal{L}_{\overline{\lambda}\mathtt{x}}$, if $a \rightarrow_{\beta} b$ then $a \xrightarrow{+}_{\overline{\lambda}\mathtt{x}} b$.*

Proof. By induction on the structure of a. We treat the case $a \equiv (\lambda x.t)(u :: l)$, $b \equiv \{t[u/x]\}l$. Then use $\rightarrow_{Beta}$ to create $t\langle x := u\rangle l$, and use Proposition 5 (4) and (3) to reach $\{t[u/x]\}l$. □

Since the translations in Section 3 preserve the types of terms, the proof of the above theorem indicates how to simulate normalization in natural deduction by cut-elimination in Herbelin's sequent calculus. Specifically, a redex in natural deduction is translated into the key-case corresponding to a *Beta*-redex $(\lambda x.t)(u :: l)$. Then transformation is performed as in Figure 1 to create the proof corresponding to $t\langle x := u\rangle l$, followed by cut-reduction steps to reach the proof corresponding to $\{t[u/x]\}l$. The latter cut-reduction steps are in fact strongly normalizing and confluent, since they correspond to reduction steps of the subcalculus x.

The strictness in Theorem 2 has a nice consequence.

Corollary 1. *Let $a \in \mathcal{T}_{\overline{\lambda}\mathtt{x}} \cup \mathcal{L}_{\overline{\lambda}\mathtt{x}}$. If $a \in \mathcal{SN}_{\overline{\lambda}\mathtt{x}}$ then $\mathtt{x}(a) \in \mathcal{SN}_{\beta}$.*

Proof. Suppose $\mathtt{x}(a) \notin \mathcal{SN}_{\beta}$. Using Theorem 2 we get an infinite $\overline{\lambda}\mathtt{x}$-reduction sequence starting with $\mathtt{x}(a)$. Since $a \xrightarrow{*}_{\overline{\lambda}\mathtt{x}} \mathtt{x}(a)$ we have $a \notin \mathcal{SN}_{\overline{\lambda}\mathtt{x}}$. □

6 Main Lemma

In this section we give an inductive proof of PSN for $\overline{\lambda}\mathtt{x}$-calculus with respect to β-reduction on pure terms. Although PSN itself was already proved in [10] in a different way, our main lemma is also useful for the reducibility proof of strong normalization in the next section. We follow the method of [2,3,5] for $\lambda\mathtt{x}$-calculus, in which the key notions are void reduction and decent terms.

Definition 2. *A substitution $\langle x := v\rangle$ is said to be* void in $a\langle x := v\rangle$ *if $x \notin FV(\mathtt{x}(a))$.* Void reduction *is $\overline{\lambda}\mathtt{x}$-reduction inside the body of a void substitution (more precisely, it is the contextual closure of the reduction: $a\langle z := v\rangle \rightarrow_{\overline{\lambda}\mathtt{x}} a\langle z := v'\rangle$ where $v \rightarrow_{\overline{\lambda}\mathtt{x}} v'$ and $z \notin FV(\mathtt{x}(a))$).*

As in the case of $\lambda\mathtt{x}$, we have the following lemmas.

Lemma 6 (Projection). *Let $a, b \in \mathcal{T}_{\overline{\lambda}\mathtt{x}} \cup \mathcal{L}_{\overline{\lambda}\mathtt{x}}$.*

1. *If $a \rightarrow_{\overline{\lambda}\mathtt{x}} b$, then $\mathtt{x}(a) \xrightarrow{*}_{\beta} \mathtt{x}(b)$.*
2. *If $a \rightarrow_{Beta} b$ is not a void reduction, then $\mathtt{x}(a) \xrightarrow{+}_{\beta} \mathtt{x}(b)$.*

Lemma 7. *If $a_0, a_1, \ldots \in \mathcal{T}_{\overline{\lambda}\mathtt{x}} \cup \mathcal{L}_{\overline{\lambda}\mathtt{x}}$ such that $\mathtt{x}(a_0) \in \mathcal{SN}_\beta$ and $a_0 \rightarrow_{\overline{\lambda}\mathtt{x}} a_1 \rightarrow_{\overline{\lambda}\mathtt{x}} \ldots$ is an infinite $\overline{\lambda}\mathtt{x}$-reduction sequence, there is a $k \in \mathbb{N}$ such that for all $i \geqslant k$, $a_i \rightarrow_{\overline{\lambda}\mathtt{x}} a_{i+1}$ is void.*

Proof. Since $\rightarrow_\mathtt{x}$ is strongly normalizing, we may assume that the infinite $\overline{\lambda}\mathtt{x}$-reduction sequence has the form $a_0 \overset{*}{\rightarrow}_\mathtt{x} a_1 \rightarrow_{Beta} a_2 \overset{*}{\rightarrow}_\mathtt{x} a_3 \ldots$. Now, by Lemma 6 (1), we have $\mathtt{x}(a_0) \overset{*}{\rightarrow}_\beta \mathtt{x}(a_2) \overset{*}{\rightarrow}_\beta \mathtt{x}(a_4) \overset{*}{\rightarrow}_\beta \mathtt{x}(a_6) \ldots$, where by Lemma 6 (2) we have $\mathtt{x}(a_{2n}) \overset{+}{\rightarrow}_\beta \mathtt{x}(a_{2n+2})$ if $a_{2n+1} \rightarrow_{Beta} a_{2n+2}$ is not void.

Now, since $\mathtt{x}(a_0) \in \mathcal{SN}_\beta$, there is a $j \in \mathbb{N}$ such that for all $i \geqslant j$, $a_{2i+1} \rightarrow_{Beta} a_{2i+2}$ is void. In what follows we prove that from some point onwards not only the $\rightarrow_{Beta}$ reductions are void but also the $\rightarrow_\mathtt{x}$ reductions. This is done by defining an interpretation h on $\mathcal{T}_{\overline{\lambda}\mathtt{x}} \cup \mathcal{L}_{\overline{\lambda}\mathtt{x}}$:

$$
\begin{array}{ll}
h(xl) & =_{def} h(l) + 1 \\
h(\lambda x.t) & =_{def} h(t) + 1 \\
h(tl) & =_{def} 2 \times h(t) + h(l) + 1 \\
h(t\langle z := v\rangle) & =_{def} \begin{cases} h(t) \times (3 \times h(v) + 1) & \text{if } z \in FV(\mathtt{x}(t)) \\ h(t) \times 4 & \text{if } z \notin FV(\mathtt{x}(t)) \end{cases}
\end{array}
$$

$$
\begin{array}{ll}
h([]) & =_{def} 1 \\
h(u :: l) & =_{def} h(u) + h(l) + 1 \\
h(ll') & =_{def} 2 \times h(l) + h(l') + 1 \\
h(l\langle z := v\rangle) & =_{def} \begin{cases} h(l) \times (3 \times h(v) + 1) & \text{if } z \in FV(\mathtt{x}(l)) \\ h(l) \times 4 & \text{if } z \notin FV(\mathtt{x}(l)) \end{cases}
\end{array}
$$

One may then verify that:

- if $a \rightarrow_{\overline{\lambda}\mathtt{x}} b$ is void, then $h(a) = h(b)$, and
- if $a \rightarrow_\mathtt{x} b$ is not void, then $h(a) > h(b)$.

Thus there must be a $k > j$ such that for all $i \geqslant k$ we have that not only $a_{2i+1} \rightarrow_{Beta} a_{2i+2}$ is void but also $a_{2i} \overset{*}{\rightarrow}_\mathtt{x} a_{2i+1}$. □

Definition 3 (Decent terms). *Let $a \in \mathcal{T}_{\overline{\lambda}\mathtt{x}} \cup \mathcal{L}_{\overline{\lambda}\mathtt{x}}$. a is said to be* decent *if for every substitution $\langle z := v\rangle$ occurring in a, $v \in \mathcal{SN}_{\overline{\lambda}\mathtt{x}}$.*

The next proposition follows easily from the fact that void reduction takes place inside the body of a (void) substitution.

Proposition 6. *For decent terms, void reduction is strongly normalizing.*

Our aim is now to prove the converse of Corollary 1 when we restrict the $\overline{\lambda}\mathtt{x}$-terms in question to decent terms. Before we proceed to the proof we need one more lemma.

Definition 4. *Let a be a pure term with $a \in \mathcal{SN}_\beta$.* $\mathrm{maxred}_\beta(a)$ *is defined as the maximal length of all β-reduction sequences starting from a.*

Lemma 8. *Let $a, b \in \mathcal{T}_{\overline{\lambda}\mathrm{x}} \cup \mathcal{L}_{\overline{\lambda}\mathrm{x}}$ with $\mathrm{x}(a) \in \mathcal{SN}_\beta$. If b is a subterm of a and is not inside the body of a substitution in a, then* $\mathrm{maxred}_\beta(\mathrm{x}(b)) \leqslant \mathrm{maxred}_\beta(\mathrm{x}(a))$.

Proof. By induction on the structure of a. We treat the case $a \equiv t\langle z := v\rangle$. If b is a strict subterm of a and is not inside the body of a substitution in a, then b is a subterm of t. Hence

$$\begin{aligned}\mathrm{maxred}_\beta(\mathrm{x}(b)) &\leqslant \mathrm{maxred}_\beta(\mathrm{x}(t)) && \text{(by the induction hypothesis)}\\ &\leqslant \mathrm{maxred}_\beta(\mathrm{x}(t)[\mathrm{x}(v)/z]) && \text{(by Lemma 4)}\\ &= \mathrm{maxred}_\beta(\mathrm{x}(t\langle z := v\rangle)) && \text{(by Lemma 5 (4))}\end{aligned}$$

□

Lemma 9 (Main lemma). *If a is decent and $\mathrm{x}(a) \in \mathcal{SN}_\beta$, then $a \in \mathcal{SN}_{\overline{\lambda}\mathrm{x}}$.*

Proof. By induction on $\mathrm{maxred}_\beta(\mathrm{x}(a))$. Suppose that a is decent and that if b is decent and $\mathrm{maxred}_\beta(\mathrm{x}(b)) < \mathrm{maxred}_\beta(\mathrm{x}(a))$ then $b \in \mathcal{SN}_{\overline{\lambda}\mathrm{x}}$. We first show that if $a \to_{\overline{\lambda}\mathrm{x}} a'$ then a' is decent. If the reduction takes place inside the body of a substitution in a then clearly a' is decent. In what follows we show that for all subterms b of a, if the reduction $b \to_{\overline{\lambda}\mathrm{x}} b'$ takes place outside the body of a substitution in a then b' is decent. This is proved by induction on the structure of b. We treat some cases.

- $b \equiv tl$, $l \to_{\overline{\lambda}\mathrm{x}} l'$ and $b' \equiv tl'$. Then l' is decent by the induction hypothesis. Therefore b' is decent.
- $b \equiv tl\langle z := v\rangle$ and $b' \equiv t\langle z := v\rangle l\langle z := v\rangle$. Then all bodies of substitutions in b' are also bodies of substitutions in b. Hence b' is decent.
- $b \equiv (\lambda z.t)(u :: l)$ and $b' \equiv t\langle z := u\rangle l$. In this case we crucially need the first induction hypothesis. All bodies of substitutions in t or l are also bodies of substitutions in b; we need to show that the new body of a substitution, u, is in $\mathcal{SN}_{\overline{\lambda}\mathrm{x}}$ too. Now since u is a subterm of b, u is decent, and $\mathrm{maxred}_\beta(\mathrm{x}(u)) < \mathrm{maxred}_\beta((\lambda z.\mathrm{x}(t))(\mathrm{x}(u) :: \mathrm{x}(l))) = \mathrm{maxred}_\beta(\mathrm{x}(b)) \leqslant \mathrm{maxred}_\beta(\mathrm{x}(a))$ by Lemma 8. Therefore, by the first induction hypothesis, $u \in \mathcal{SN}_{\overline{\lambda}\mathrm{x}}$.

Hence if $a \to_{\overline{\lambda}\mathrm{x}} a'$ then a' is decent. Moreover, if $a \to_{\overline{\lambda}\mathrm{x}} a'$ then by Lemma 6 (1), $\mathrm{x}(a) \overset{*}{\to}_\beta \mathrm{x}(a')$ and so $\mathrm{maxred}_\beta(\mathrm{x}(a')) \leqslant \mathrm{maxred}_\beta(\mathrm{x}(a))$, where if $\mathrm{maxred}_\beta(\mathrm{x}(a')) < \mathrm{maxred}_\beta(\mathrm{x}(a))$ then $a' \in \mathcal{SN}_{\overline{\lambda}\mathrm{x}}$ by the first induction hypothesis, otherwise $\mathrm{maxred}_\beta(\mathrm{x}(a')) = \mathrm{maxred}_\beta(\mathrm{x}(a))$ and we can apply the above argument to a' to show that if $a' \to_{\overline{\lambda}\mathrm{x}} a''$ then a'' is decent. Thus we see that for any a' such that $a \overset{*}{\to}_{\overline{\lambda}\mathrm{x}} a'$, a' is decent.

Now suppose that a has an infinite $\overline{\lambda}\mathrm{x}$-reduction path. By Lemma 7 there is a term a' on this path such that from a' on all reductions are void. But we just proved that a' is decent, so we have a contradiction with Proposition 6. Therefore, $a \in \mathcal{SN}_{\overline{\lambda}\mathrm{x}}$. □

Corollary 2 (PSN). *For any pure term* $a \in \mathcal{T}_{\overline{\lambda}\mathbf{x}} \cup \mathcal{L}_{\overline{\lambda}\mathbf{x}}$, $a \in \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$ *iff* $a \in \mathcal{SN}_{\beta}$.

Proof. The only if part is by Corollary 1. Since pure terms are decent, we have the if part by Lemma 9. □

7 Strong Normalization

In this section we prove that all typable $\overline{\lambda}\mathbf{x}$-terms are strongly normalizing. For this we use the reducibility method adapted to explicit substitution calculus and to Herbelin-style calculus. Here we consider reducibility sets only over $\mathcal{T}_{\overline{\lambda}\mathbf{x}}$ and not over $\mathcal{L}_{\overline{\lambda}\mathbf{x}}$, which is sufficient to prove our main result.

Definition 5. *For each type A, the set* $\mathcal{R}^A$ *is defined inductively as follows:*

$$\begin{aligned} \mathcal{R}^{\varphi} &=_{def} \mathcal{SN}_{\overline{\lambda}\mathbf{x}} \cap \mathcal{T}_{\overline{\lambda}\mathbf{x}} \\ \mathcal{R}^{B\supset C} &=_{def} \{t \in \mathcal{T}_{\overline{\lambda}\mathbf{x}} \mid \forall v \in \mathcal{R}^B[t(v :: []) \in \mathcal{R}^C]\} \end{aligned}$$

where φ *is a type variable.*

In the following, we abbreviate a $\overline{\lambda}\mathbf{x}$-term $t_1 :: (t_2 :: \cdots (t_n :: [])\ldots)$ to $t_1 :: t_2 :: \cdots :: t_n :: []$, and $(\ldots((tl_1)l_2)\ldots)l_n$ to $tl_1l_2\ldots l_n$.

Lemma 10. For *every type A and every variable* x,

1. $\mathcal{R}^A \subseteq \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$.
2. *If* $x(u_1 :: \cdots :: u_n :: []) \in \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$, *then* $(x[])(u_1 :: [])\ldots(u_n :: []) \in \mathcal{R}^A$.

Proof. By simultaneous induction on the structure of A.

i) A is a type variable φ.
   1. By the definition of $\mathcal{R}^{\varphi}$.
   2. Let $x(u_1 :: \cdots :: u_n :: []) \in \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$. Then $(x[])(u_1 :: [])\ldots(u_n :: [])$ is decent, and $\mathbf{x}(x(u_1 :: \cdots :: u_n :: [])) \in \mathcal{SN}_{\beta}$ by Corollary 1. Since $\mathbf{x}((x[])(u_1 :: [])\ldots(u_n :: [])) \equiv \mathbf{x}(x(u_1 :: \cdots :: u_n :: []))$, we have $(x[])(u_1 :: [])\ldots(u_n :: []) \in \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$ by Lemma 9. Hence $(x[])(u_1 :: [])\ldots(u_n :: []) \in \mathcal{R}^{\varphi}$.

ii) A is of the form $B \supset C$.
   1. Let $t \in \mathcal{R}^{B\supset C}$. By the induction hypothesis for the second item, $x[] \in \mathcal{R}^B$ and so $t(x[] :: []) \in \mathcal{R}^C$. By the induction hypothesis for the first item, $t(x[] :: []) \in \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$. Hence $t \in \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$.
   2. Let $x(u_1 :: \cdots :: u_n :: []) \in \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$ and let $v \in \mathcal{R}^B$. By the induction hypothesis for the first item, $v \in \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$ and so $x(u_1 :: \cdots :: u_n :: v :: []) \in \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$. By the induction hypothesis for the second item, $(x[])(u_1 :: [])\ldots(u_n :: [])(v :: []) \in \mathcal{R}^C$. Hence $(x[])(u_1 :: [])\ldots(u_n :: []) \in \mathcal{R}^{B\supset C}$ if $x(u_1 :: \cdots :: u_n :: []) \in \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$. □

Henceforth we use the result of Lemma 10 (1) without reference.

The next two lemmas are essential to our reducibility proof in which the reducibility sets need to be closed under certain expansion with respect to $\overline{\lambda}\mathbf{x}$-reduction.

Lemma 11. *Let* $\mathtt{x}(s) \equiv \mathtt{x}(t)$ *and* $t \in \mathcal{R}^A$. *If* s *is decent (in particular, if every substitution body in* s *is a subterm of* t*), then* $s \in \mathcal{R}^A$.

Proof. By induction on the structure of A.

i) A is a type variable φ. Let $\mathtt{x}(s) \equiv \mathtt{x}(t)$ and $t \in \mathcal{R}^\varphi (\subseteq \mathcal{SN}_{\overline{\lambda}\mathtt{x}})$. Then $\mathtt{x}(s) \equiv \mathtt{x}(t) \in \mathcal{SN}_\beta$ by Corollary 1. If s is decent, then $s \in \mathcal{SN}_{\overline{\lambda}\mathtt{x}}$ by Lemma 9, and hence $s \in \mathcal{R}^\varphi$.
ii) A is of the form $B \supset C$. We show that for any $v \in \mathcal{R}^B$, $s(v :: []) \in \mathcal{R}^C$. Suppose $v \in \mathcal{R}^B (\subseteq \mathcal{SN}_{\overline{\lambda}\mathtt{x}})$. Since $t \in \mathcal{R}^{B\supset C}$ by assumption, we have $t(v :: []) \in \mathcal{R}^C$. By Lemma 5 (3), $\mathtt{x}(s(v :: [])) \equiv \{\mathtt{x}(s)\}\mathtt{x}(v :: []) \equiv \{\mathtt{x}(t)\}\mathtt{x}(v :: []) \equiv \mathtt{x}(t(v :: []))$. Since s and v are decent, so is $s(v :: [])$. Hence by the induction hypothesis, we have $s(v :: []) \in \mathcal{R}^C$. □

Lemma 12. *If* $(t\langle x := v\rangle)(u_1 :: []) \dots (u_n :: []) \in \mathcal{R}^A$, *then* $(\lambda x.t)(v :: [])(u_1 :: []) \dots (u_n :: []) \in \mathcal{R}^A$.

Proof. By induction on the structure of A.

i) A is a type variable φ. By Lemma 11, it suffices to show that if $(t\langle x := v\rangle)(u_1 :: \cdots :: u_n :: []) \in \mathcal{R}^\varphi$ then $(\lambda x.t)(v :: u_1 :: \cdots :: u_n :: []) \in \mathcal{R}^\varphi$. Suppose $(t\langle x := v\rangle)(u_1 :: \cdots :: u_n :: []) \in \mathcal{R}^\varphi (\subseteq \mathcal{SN}_{\overline{\lambda}\mathtt{x}})$. Then $t, v, u_1, \dots, u_n \in \mathcal{SN}_{\overline{\lambda}\mathtt{x}}$. Here we can see that if $(\lambda x.t)(v :: u_1 :: \cdots :: u_n :: [])$ has an infinite reduction sequence then so does $(t\langle x := v\rangle)(u_1 :: \cdots :: u_n :: [])$ contradicting the hypothesis. Hence $(\lambda x.t)(v :: u_1 :: \cdots :: u_n :: []) \in \mathcal{SN}_{\overline{\lambda}\mathtt{x}} \cap \mathcal{T}_{\overline{\lambda}\mathtt{x}} = \mathcal{R}^\varphi$.
ii) A is of the form $B \supset C$. Easily seen by the induction hypothesis for C. □

Now we prove the reducibility lemma using the results we obtained so far.

Lemma 13.

1. *Let* $x_1 : A_1, \dots, x_n : A_n; - \vdash t : B$ *and let* $u_i \in \mathcal{R}^{A_i}$ *for each* $1 \leqslant i \leqslant n$ *where* $x_j \notin FV(u_i)$ *for all* $1 \leqslant j \leqslant n$. *Then* $t\langle x_1 := u_1\rangle \dots \langle x_n := u_n\rangle \in \mathcal{R}^B$.
2. *Let* $x_1 : A_1, \dots, x_n : A_n; B \vdash l : C$ *and let* $u_i \in \mathcal{R}^{A_i}$ *for each* $1 \leqslant i \leqslant n$ *where* $x_j \notin FV(u_i)$ *for all* $1 \leqslant j \leqslant n$. *Then for any* $t \in \mathcal{R}^B$, $t(l\langle x_1 := u_1\rangle \dots \langle x_n := u_n\rangle) \in \mathcal{R}^C$.

Proof. Both items are proved simultaneously by induction on the structure of derivations. We treat some cases. Let Γ denote $x_1 : A_1, \dots, x_n : A_n$.

i)
$$\frac{}{\Gamma; A \vdash [] : A}\,Ax$$

We show that for any $t \in \mathcal{R}^A$, $t([]\langle x_1 := u_1\rangle \dots \langle x_n := u_n\rangle) \in \mathcal{R}^A$. Suppose $t \in \mathcal{R}^A (\subseteq \mathcal{SN}_{\overline{\lambda}\mathtt{x}})$. By assumption, we have $u_i \in \mathcal{R}^{A_i} (\subseteq \mathcal{SN}_{\overline{\lambda}\mathtt{x}})$ for each $1 \leqslant i \leqslant n$. Hence $t([]\langle x_1 := u_1\rangle \dots \langle x_n := u_n\rangle)$ is decent. Now we have $\mathtt{x}(t([]\langle x_1 := u_1\rangle \dots \langle x_n := u_n\rangle)) \equiv \mathtt{x}(t[]) \equiv \{\mathtt{x}(t)\}[] \equiv \mathtt{x}(t)$ using Lemma 5 (3) and Lemma 1. Hence $t([]\langle x_1 := u_1\rangle \dots \langle x_n := u_n\rangle) \in \mathcal{R}^A$ by Lemma 11.

ii)

$$\frac{\Gamma, x : A; A \vdash l : B}{\Gamma, x : A; - \vdash xl : B}\ Der$$

We show that

$$(xl)\langle x_1 := u_1\rangle \ldots \langle x_i := u_i\rangle\langle x := u\rangle\langle x_{i+1} := u_{i+1}\rangle \ldots \langle x_n := u_n\rangle \in \mathcal{R}^B$$

where $u \in \mathcal{R}^A$ and $x_j \notin FV(u)$ for all $1 \leqslant j \leqslant n$. By the induction hypothesis, $u(l\langle x_1 := u_1\rangle \ldots \langle x_i := u_i\rangle\langle x := u\rangle\langle x_{i+1} := u_{i+1}\rangle \ldots \langle x_n := u_n\rangle) \in \mathcal{R}^B$. Now,

$$\begin{aligned}
&(xl)\langle x_1 := u_1\rangle \ldots \langle x_i := u_i\rangle\langle x := u\rangle\langle x_{i+1} := u_{i+1}\rangle \ldots \langle x_n := u_n\rangle \\
\overset{*}{\rightarrow}_{\mathrm{x}}&(xl\langle x_1 := u_1\rangle \ldots \langle x_i := u_i\rangle)\langle x := u\rangle\langle x_{i+1} := u_{i+1}\rangle \ldots \langle x_n := u_n\rangle \\
\rightarrow_{\mathrm{x}}&(ul\langle x_1 := u_1\rangle \ldots \langle x_i := u_i\rangle\langle x := u\rangle)\langle x_{i+1} := u_{i+1}\rangle \ldots \langle x_n := u_n\rangle \\
\overset{*}{\rightarrow}_{\mathrm{x}}&u\langle x_{i+1} := u_{i+1}\rangle \ldots \langle x_n := u_n\rangle l\langle x_1 := u_1\rangle \ldots \langle x := u\rangle \ldots \langle x_n := u_n\rangle
\end{aligned}$$

and hence

$$\begin{aligned}
&\mathrm{x}((xl)\langle x_1 := u_1\rangle \ldots \langle x_i := u_i\rangle\langle x := u\rangle\langle x_{i+1} := u_{i+1}\rangle \ldots \langle x_n := u_n\rangle) \\
\equiv\ &\mathrm{x}(u\langle x_{i+1} := u_{i+1}\rangle \ldots \langle x_n := u_n\rangle l\langle x_1 := u_1\rangle \ldots \langle x := u\rangle \ldots \langle x_n := u_n\rangle) \\
\equiv\ &\{\mathrm{x}(u)[\mathrm{x}(u_{i+1})/x_{i+1}]\ldots[\mathrm{x}(u_n)/x_n]\}\mathrm{x}(l)[\mathrm{x}(u_1)/x_1]\ldots[\mathrm{x}(u)/x]\ldots[\mathrm{x}(u_n)/x_n] \\
&\hfill \text{(by Lemma 5)} \\
\equiv\ &\{\mathrm{x}(u)\}\mathrm{x}(l)[\mathrm{x}(u_1)/x_1]\ldots[\mathrm{x}(u)/x]\ldots[\mathrm{x}(u_n)/x_n] \qquad \text{(by Lemma 2)} \\
\equiv\ &\mathrm{x}(u(l\langle x_1 := u_1\rangle \ldots \langle x := u\rangle \ldots \langle x_n := u_n\rangle)). \qquad \text{(by Lemma 5)}
\end{aligned}$$

Therefore, by Lemma 11, we have

$$(xl)\langle x_1 := u_1\rangle \ldots \langle x_i := u_i\rangle\langle x := u\rangle\langle x_{i+1} := u_{i+1}\rangle \ldots \langle x_n := u_n\rangle \in \mathcal{R}^B.$$

iii)

$$\frac{\Gamma; - \vdash v : A \quad \Gamma; B \vdash l : C}{\Gamma; A \supset B \vdash v :: l : C}\ L\supset$$

We show that for any $t \in \mathcal{R}^{A\supset B}$, $t((v :: l)\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle) \in \mathcal{R}^C$. Suppose $t \in \mathcal{R}^{A\supset B}$. By the induction hypothesis for the left premise, $v\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle \in \mathcal{R}^A$, and so $t(v\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle :: []) \in \mathcal{R}^B$. Then by the induction hypothesis for the right premise,

$$(t(v\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle :: []))(l\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle) \in \mathcal{R}^C.$$

Since $\mathrm{x}(t((v :: l)\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle)) \equiv \mathrm{x}((t(v\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle :: []))(l\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle))$, we have $t((v :: l)\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle) \in \mathcal{R}^C$ by applying Lemma 11.

iv)

$$\frac{\Gamma, x : A; - \vdash t : B}{\Gamma; - \vdash \lambda x.t : A \supset B}\, R\supset$$

We show that $(\lambda x.t)\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle \in \mathcal{R}^{A\supset B}$. Suppose $v \in \mathcal{R}^A$. Then by the induction hypothesis, $t\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle\langle x := v\rangle \in \mathcal{R}^B$. By Lemma 12, $(\lambda x.t\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle)(v :: [\,]) \in \mathcal{R}^B$. Hence $\lambda x.t\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle \in \mathcal{R}^{A\supset B}$, and by applying Lemma 11 we have $(\lambda x.t)\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle \in \mathcal{R}^{A\supset B}$.

v)

$$\frac{\Gamma; - \vdash v : A \quad \Gamma, x : A; B \vdash l : C}{\Gamma; B \vdash l\langle x := v\rangle : C}\, Cut_2$$

We show that for any $t \in \mathcal{R}^B$, $t(l\langle x := v\rangle\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle) \in \mathcal{R}^C$. Suppose $t \in \mathcal{R}^B$. By the induction hypothesis for the left premise, $v\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle \in \mathcal{R}^A$. Then by the induction hypothesis for the right premise, $t(l\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle\langle x := v\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle\rangle) \in \mathcal{R}^C$. Using Lemma 5 and the substitution lemma (Lemma 3) we have $\mathtt{x}(t(l\langle x := v\rangle\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle)) \equiv \mathtt{x}(t(l\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle\langle x := v\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle\rangle))$, and hence $t(l\langle x := v\rangle\langle x_1 := u_1\rangle \ldots \langle x_n := u_n\rangle) \in \mathcal{R}^C$ by Lemma 11. □

Theorem 3.

1. *Let* $\Gamma; - \vdash t : B$ *for some* Γ *and* B. *Then* $t \in \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$.
2. *Let* $\Gamma; B \vdash l : C$ *for some* Γ, B *and* C. *Then* $l \in \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$.

Proof. 1. Suppose Γ is $\{x_1 : A_1, \ldots, x_n : A_n\}$. By Lemma 13 (1), $t\langle x_1 := y_1[\,]\rangle \ldots \langle x_n := y_n[\,]\rangle \in \mathcal{R}^B (\subseteq \mathcal{SN}_{\overline{\lambda}\mathbf{x}})$ where each y_i is fresh and $y_i[\,] \in \mathcal{R}^{A_i}$ by Lemma 10 (2). Hence $t \in \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$.

2. Similarly, $(z[\,])(l\langle x_1 := y_1[\,]\rangle \ldots \langle x_n := y_n[\,]\rangle) \in \mathcal{R}^C (\subseteq \mathcal{SN}_{\overline{\lambda}\mathbf{x}})$ where z is also fresh. Hence $l \in \mathcal{SN}_{\overline{\lambda}\mathbf{x}}$. □

The above theorem shows that the cut-elimination procedure defined by the reduction rules of $\overline{\lambda}$x-calculus is strongly normalizing. Also, by Corollary 1, we have strong normalization for typable pure terms with respect to β-reduction, and thus strong normalization for typable terms in the usual λ-calculus as well.

8 Conclusion

In this paper we presented a direct proof of strong normalization for the typable terms of the extended Herbelin's calculus. The main lemma was useful for both the inductive proof of PSN with respect to β-reduction on pure terms and the reducibility proof of strong normalization for typable terms. Our reducibility method seems to be helpful in investigating other reduction properties and semantical aspects of the calculus.

In the literature [13,16] there are reducibility methods for other calculi with explicit substitutions. The relationship between them and ours will be investigated in future work.

Since the extended Herbelin's calculus clarifies how to simulate β-reduction by cut-elimination, it can be viewed as a basis for understanding computational meanings of various cut-elimination procedures. It would also be interesting to use the calculus for studies of integrating proof search and proof normalization.

Acknowledgements. I am grateful to Takafumi Sakurai for useful discussions and encouragement. I also thank Roy Dyckhoff, Stéphane Lengrand and the anonymous referees for valuable comments.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming,* 1:375–416, 1991.
2. R. Bloo. Preservation of strong normalisation for explicit substitution. Computing Science Report 95-08, Eindhoven University of Technology, 1995.
3. R. Bloo. *Preservation of Termination for Explicit Substitution.* PhD thesis, Eindhoven University of Technology, 1997.
4. R. Bloo and H. Geuvers. Explicit substitution: on the edge of strong normalization. *Theoretical Computer Science,* 211:375–395, 1999.
5. R. Bloo and K. H. Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *Proceedings of CSN'95 (Computing Science in the Netherlands),* pages 62–72, 1995.
6. E. Bonelli. Perpetuality in a named lambda calculus with explicit substitutions. *Mathematical Structures in Computer Science,* 11:47–90, 2001.
7. D. Dougherty and P. Lescanne. Reductions, intersection types, and explicit substitutions. *Mathematical Structures in Computer Science,* 13:55–85, 2003.
8. R. Dyckhoff and L. Pinto. Proof search in constructive logic. In S. B. Cooper and J. K. Truss, editors, *Proceedings of the Logic Colloquium 1997,* London Mathematical Society Lecture Note Series 258, pages 53–65. Cambridge University Press, 1997.
9. R. Dyckhoff and C. Urban. Strong normalisation of Herbelin's explicit substitution calculus with substitution propagation. In *Proceedings of WESTAPP'01,* pages 26–45, 2001.
10. R. Dyckhoff and C. Urban. Strong normalization of Herbelin's explicit substitution calculus with substitution propagation. *Journal of Logic and Computation,* 13:689–706, 2003.
11. J. Espírito Santo. Revisiting the correspondence between cut elimination and normalisation. In *Proceedings of ICALP'00,* Lecture Notes in Computer Science 1853, pages 600–611. Springer-Verlag, 2000.
12. H. Herbelin. A λ-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Proceedings of CSL'94,* Lecture Notes in Computer Science 933, pages 61–75. Springer-Verlag, 1995.
13. H. Herbelin. Explicit substitutions and reducibility. *Journal of Logic and Computation,* 11:429–449, 2001.

14. F. Joachimski and R. Matthes. Short proofs of normalization for the simply-typed λ-calculus, permutative conversions and Gödel's T. *Archive for Mathematical Logic,* 42:59–87, 2003.
15. P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate. In *Proceedings of TLCA '95,* Lecture Notes in Computer Science 902, pages 328–334. Springer-Verlag, 1995.
16. E. Ritter. Characterising explicit substitutions which preserve termination. In *Proceedings of TLCA'99,* Lecture Notes in Computer Science 1581, pages 325–339. Springer-Verlag, 1999.
17. W. W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic,* 32:198–212, 1967.
18. A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Cambridge Tracts in Theoretical Computer Science 43. Cambridge University Press, 2nd edition, 2000.

Normalization by Evaluation for $\lambda^{\to 2}$

Thorsten Altenkirch[1*] and Tarmo Uustalu[2**]

[1] School of Computer Science and IT, University of Nottingham
Nottingham NG8 1BB, United Kingdom
`txa@cs.nott.ac.uk`
[2] Institute of Cybernetics, Tallinn Technical University
Akadeemia tee 21, EE-12618 Tallinn, Estonia
`tarmo@cs.ioc.ee`

Abstract. We show that the set-theoretic semantics of $\lambda^{\to 2}$—simply typed lambda calculus with a boolean type but no type variables—is complete by inverting evaluation using decision trees. This leads to an implementation of normalization by evaluation which is witnessed by the source of part of this paper being a literate Haskell script. We show the correctness of our implementation using logical relations.

1 Introduction

Which is the simplest typed λ-calculus without uninterpreted base types or type variables? We suggest that the answer should be $\lambda^{\to 2}$: simply typed lambda calculus extended by a type of booleans `Bool` with `True`, `False : Bool` and `If` $t\,u_0\,u_1 : \sigma$, given t : `Bool` and $u_0, u_1 : \sigma$. The equational theory is given by the usual $\beta\eta$-equations of $\lambda^{\to}$ and the following equations concerning `Bool`:

$$\begin{aligned}
\mathtt{If}\,\mathtt{True}\,u_0\,u_1 &=_{\beta\eta} u_0\\
\mathtt{If}\,\mathtt{False}\,u_0\,u_1 &=_{\beta\eta} u_1\\
\mathtt{If}\,t\,\mathtt{True}\,\mathtt{False} &=_{\beta\eta} t\\
v\,(\mathtt{If}\,t\,u_0\,u_1) &=_{\beta\eta} \mathtt{If}\,t\,(v\,u_0)\,(v\,u_1)
\end{aligned}$$

The equations are motivated by the categorical interpretation of `Bool` as a boolean object, i.e., an object Bool such that $\mathrm{Hom}(\Gamma \times \mathrm{Bool}, \sigma) \simeq \mathrm{Hom}(\Gamma, \sigma) \times \mathrm{Hom}(\Gamma, \sigma)$ (naturally in Γ and σ). The calculus can thus be interpreted in any cartesian closed category with Bool (using the cartesian structure to interpret contexts).

The equational theory introduces some interesting equalities. E.g., consider

$$\begin{aligned}
\mathtt{once} &= \lambda f^{\mathtt{Bool}\to\mathtt{Bool}} \lambda x^{\mathtt{Bool}} f\,x\\
\mathtt{thrice} &= \lambda f^{\mathtt{Bool}\to\mathtt{Bool}} \lambda x^{\mathtt{Bool}} f\,(f\,(f\,x))
\end{aligned}$$

[*] partially supported by EU Framework 5 thematic networks TYPES and APPSEM II and the Estonian IT Foundation

[**] partially supported by the Estonian Science Foundation under grant No. 5567

© Springer-Verlag Berlin Heidelberg 2004

We observe that $\texttt{once} =_{\beta\eta} \texttt{thrice}$. To see this, we note that, given $f : \texttt{Bool} \to \texttt{Bool}$, we have

$$\begin{aligned}
f\,(f\,(f\,\texttt{True})) &=_{\beta\eta} \texttt{If}\,(f\,\texttt{True})\,(f\,(f\,\texttt{True}))\,(f\,(f\,\texttt{False}))\\
&=_{\beta\eta} \texttt{If}\,(f\,\texttt{True})\,\texttt{True}\,(f\,(f\,\texttt{False}))\\
&=_{\beta\eta} \texttt{If}\,(f\,\texttt{True})\,\texttt{True}\,(\texttt{If}\,(f\,\texttt{False})\,(f\,\texttt{True})\,(f\,\texttt{False}))\\
&=_{\beta\eta} \texttt{If}\,(f\,\texttt{True})\,\texttt{True}\,(\texttt{If}\,(f\,\texttt{False})\,\texttt{False}\,\texttt{False})\\
&=_{\beta\eta} \texttt{If}\,(f\,\texttt{True})\,\texttt{True}\,\texttt{False}\\
&=_{\beta\eta} f\,\texttt{True}
\end{aligned}$$

Symmetrically, we can show that $f\,(f\,(f\,\texttt{False})) =_{\beta\eta} f\,\texttt{False}$, and hence

$$\begin{aligned}
&\texttt{thrice}\\
&= \lambda f^{\texttt{Bool}\to\texttt{Bool}}\lambda x^{\texttt{Bool}} f\,(f\,(f\,x))\\
&=_{\beta\eta} \lambda f^{\texttt{Bool}\to\texttt{Bool}}\lambda x^{\texttt{Bool}}\,\texttt{If}\,x\,(f\,(f\,(f\,\texttt{True})))\,(f\,(f\,(f\,\texttt{False})))\\
&=_{\beta\eta} \lambda f^{\texttt{Bool}\to\texttt{Bool}}\lambda x^{\texttt{Bool}}\,\texttt{If}\,x\,(f\,\texttt{True})\,(f\,\texttt{False})\\
&=_{\beta\eta} \lambda f^{\texttt{Bool}\to\texttt{Bool}}\lambda x^{\texttt{Bool}} f\,x\\
&= \texttt{once}
\end{aligned}$$

It is easy to see that once and thrice are equal in the standard semantics where Bool is interpreted by a two-element set Bool = {true, false} and function types are set-theoretic function spaces. We observe that there are only four elements in $\text{Bool} \to \text{Bool} = \{x \mapsto \text{true}, x \mapsto x, x \mapsto \neg x, x \mapsto \text{false}\}$ and that for all the four $f \in \text{Bool} \to \text{Bool}$ we have $f^3 = f$.

May we use set-theoretic reasoning to prove equalities up to $\beta\eta$-convertibility? The answer is yes for $\lambda^{\to 2}$, because for $\lambda^{\to 2}$ we can *invert* set-theoretic *evaluation* of typed closed terms. That is: we can define a function $\mathsf{quote}^\sigma \in [\![\sigma]\!]_{\mathsf{set}} \to \mathsf{Tm}\ \sigma$ such that $t =_{\beta\eta} \mathsf{quote}^\sigma\, [\![t]\!]_{\mathsf{set}}$, for any $t \in \mathsf{Tm}\ \sigma$. Consequently, we get that, for any $t, t' \in \mathsf{Tm}\ \sigma$, $t =_{\beta\eta} t' \iff [\![t]\!]_{\mathsf{set}} = [\![t']\!]_{\mathsf{set}}$.

The existence of quote also implies that $=_{\beta\eta}$ is maximally consistent, i.e., identifying any two non-$\beta\eta$-convertible closed terms would lead to an inconsistent theory. This provides another justification for the specific choice of $=_{\beta\eta}$.

We do not analyze the normal forms, i.e. the codomain of nf and quote, here. However, the construction presented here, which is based on decision trees, leads to simple normal forms and we conjecture that this is the same set as the set of normal forms presented in [1,5] in the case Bool = 1 + 1.

Haskell as a Poor Man's Type Theory

Our construction is entirely constructive, so it can be carried out, e.g. in Martin-Löf's Type Theory, and we obtain an implementation of normalization $\mathsf{nf}^\sigma\, t = \mathsf{quote}^\sigma\, [\![t]\!]_{\mathsf{set}}$. We shall here use the functional language Haskell as a poor man's Type Theory and obtain a Haskell program to normalize terms.

Haskell hasn't got dependent types (in particular inductive families), hence the Haskell types we are using are only approximations of their type-theoretic correspondents. E.g., in Type Theory, we can introduce a type $\mathsf{Tm}_\Gamma\ \sigma$ for the terms of type σ in context Γ, but in Haskell all such types get approximated by a type Tm that contains all untyped terms. Similarly, in Type Theory, we can have a type $[\![\sigma]\!]_{\mathsf{set}}$ for the set-theoretic denotation of each object-language type σ, so that, e.g., $[\![\sigma :\!\rightarrow \tau]\!]_{\mathsf{set}} = [\![\sigma]\!]_{\mathsf{set}} \rightarrow [\![\tau]\!]_{\mathsf{set}}$, but in the Haskell implementation we have to contend ourselves a mixed-variant recursive type El with a constructor SLam $\in$ Ty $\rightarrow$ (El $\rightarrow$ El) $\rightarrow$ El.

We believe that this informal use of Type Theory is an effective way to arrive at functional programs which are correct by construction. However, we hope that in the future we can go further and bridge the gap between informal type theoretic reasoning and the actual implementation by using a dependently typed programming language such as the Epigram system, which is currently being developed by Conor McBride [12].

Related Work

Inverting evaluation to achieve normalization by evaluation (NBE, aka. reduction-free normalization) was pioneered in [6] for simply typed lambda calculus with type variables and a non-standard semantics; a categorical account in terms of presheaves was given in [2]; this was extended to System F in [3,4]; see [9] for a recent survey on NBE. The completeness of the set-theoretic model in the presence of coproducts has been shown in [8] and our case arises as a special case when there are no type variables. Normalization procedures for typed λ-calculus with coproducts can be found in [10,11] using rewriting techniques and [1,5] using NBE and sheaf theory. Both approaches allow type variables but do not handle the empty type. Here we present a much simpler construction for closed types using the simplest possible semantics of first-order simply typed λ-calculi—the set-theoretic one—and also provide a concrete implementation of quote and nf whose correctness we show in detail.

2 Implementation of $\lambda^{\rightarrow 2}$

The source of Sections 2 and 3 of this paper is a literate Haskell script implementing normalization for $\lambda^{\rightarrow 2}$ and is available from

http://www.cs.nott.ac.uk/~txa/publ/Nbe2.lhs

We start by introducing types Ty $\in \star$, variables Var $\in \star$, typing contexts Con $\in \star$ and untyped terms Tm $\in \star$ of the object language by the following Haskell datatype definitions:

```
data Ty = Bool | Ty :-> Ty
        deriving (Show, Eq)

type Var = String

type Con = [ (Var, Ty) ]
```

```
data Tm = Var Var
        | TTrue | TFalse | If Tm Tm Tm
        | Lam Ty String Tm | App Tm Tm
        deriving (Show, Eq)
```

We view these recursive definitions as inductive definitions, i.e., we do not consider infinite terms. All the functions we define are total wrt. their precise type-theoretic types.

Implementing typed terms $\mathsf{Tm} \in \mathsf{Con} \to \mathsf{Ty} \to \star$ would take inductive families, which we cannot use in Haskell. But we can implement type inference $\mathsf{infer} \in \mathsf{Con} \to \mathtt{Tm} \to \mathsf{Maybe}\ \mathsf{Ty}$ (where $\mathsf{Maybe}\ X = 1 + X$ as usual):

```
infer :: Con -> Tm -> Maybe Ty
infer gamma (Var x) =
    do sigma <- lookup x gamma
       Just sigma
infer gamma TTrue  = Just Bool
infer gamma TFalse = Just Bool
infer gamma (If t u0 u1) =
    do Bool <- infer gamma t
       sigma0 <- infer gamma u0
       sigma1 <- infer gamma u1
       if sigma0 == sigma1 then Just sigma0 else Nothing
infer gamma (Lam sigma x t) =
    do tau <- infer ((x, sigma) : gamma) t
       Just (sigma :-> tau)
infer gamma (App t u) =
    do (sigma :-> tau) <- infer gamma t
       sigma' <- infer gamma u
       if sigma == sigma' then Just tau else Nothing
```

This implementation is correct in the sense that $t \in \mathsf{Tm}_\Gamma\ \sigma$ iff $\mathsf{infer}_\Gamma\ t = \mathsf{Just}\ \sigma$.

Evaluation of types $[\![-]\!] \in \mathsf{Ty} \to \star$ is again an inductive family, which we cannot implement in Haskell, and the workaround is to have all $[\![\sigma]\!]$ coalesced into one metalanguage type *el* (of untyped elements) much the same way as all $\mathsf{Tm}_\Gamma\ \sigma$ appear coalesced in $\mathtt{Tm}$. We use a type class Sem to state what we require of such a coalesced type *el*:

```
class Sem el  where
    true  :: el
    false :: el
    xif :: el -> el -> el -> el
    lam :: Ty -> (el -> el) -> el
    app :: el -> el -> el
```

Evaluation of types $[\![-]\!] \in \mathsf{Ty} \to \star$ naturally induces evaluation of contexts $[\![-]\!] \in \mathsf{Con} \to \star$ (taking a context Γ to the type of environments for Γ), defined by

$$\frac{}{[] \in [\![[]]\!]} \qquad \frac{\rho \in [\![\Gamma]\!] \quad d \in [\![\sigma]\!]}{(x,d) : \rho \in [\![(x,\sigma) : \Gamma]\!]}$$

In the Haskell code we approximate evaluation of contexts by a type $\mathbf{Env}_{el}$ (of untyped environments):

```
type Env el = [ (Var, el) ]
```

Given $t \in \mathsf{Tm}_\Gamma\, \sigma$ we define the evaluation of terms $[\![t]\!] \in [\![\Gamma]\!] \to [\![\sigma]\!]$. In Haskell this is implemented as eval:

```
eval :: Sem el => Env el -> Tm -> el
eval rho (Var x) = d
    where (Just d) = lookup x rho
eval rho TTrue = true
eval rho TFalse =  false
eval rho (If t u0 u1) =
    xif (eval rho t) (eval rho u0) (eval rho u1)
eval rho (Lam sigma x t) =
    lam sigma (\ d -> eval ((x, d) : rho) t)
eval rho (App t u) = app (eval rho t) (eval rho u)
```

The standard set-theoretic semantics is given by

$$\begin{aligned}[\![\mathtt{Bool}]\!]_{\mathsf{set}} &= \mathtt{Bool}\\ [\![\sigma :\!\to \tau]\!]_{\mathsf{set}} &= [\![\sigma]\!]_{\mathsf{set}} \to [\![\tau]\!]_{\mathsf{set}}\end{aligned}$$

This can be represented in Haskell as an instance of Sem:

```
data El = STrue | SFalse | SLam Ty (El -> El)

instance Sem El where
    true  = STrue
    false = SFalse
    xif STrue  d _ = d
    xif SFalse _ d = d
    lam = SLam
    app (SLam _ f) d = f d
```

Since sets form a cartesian closed category with a boolean object, the set-theoretic semantics validates all $\beta\eta$-equalities. This is to say that $[\![-]\!]_{\mathsf{set}}$ is equationally sound:

Proposition 2.1 (Soundness). *If $\rho \in [\![\Gamma]\!]$ and $t =_{\beta\eta} t' \in \mathsf{Tm}_\Gamma\ \sigma$, then $[\![t]\!]_{\mathsf{set}}\ \rho = [\![t']\!]_{\mathsf{set}}\ \rho$.*

Since all the sets we consider are finite, semantic equality can be implemented in Haskell, by making use of the function $\mathsf{enum} \in (\sigma \in \mathsf{Ty}) \to \mathsf{Tree}\ [\![\sigma]\!]$, which we will provide later:

```
instance Eq El where
    STrue  == STrue  = True
    SFalse == SFalse = True
    (SLam sigma f) == (SLam _ f') =
        and [f d == f' d | d <- flatten (enum sigma)]
    _ == _ = False
```

Using on the same function we can also print elements of `El`:

```
instance Show El where
    show STrue  = "STrue"
    show SFalse = "SFalse"
    show (SLam sigma f) =
        "SLam " ++ (show sigma) ++ " " ++
        (show [ (d, f d) | d <- flatten (enum sigma) ])
```

The equational theory of the calculus itself gives rise to another semantics—the free semantics, or typed terms up to $\beta\eta$-convertibility. This can be approximated by the following Haskell code, where a redundancy-avoiding version `if'` of `If` is used which produces a shorter but $\beta\eta$-equivalent term:

```
if' :: Tm -> Tm -> Tm -> Tm
if' t TTrue TFalse = t
if' t u0 u1 = if u0 == u1 then u0 else If t u0 u1

instance Sem Tm where
    true  = TTrue
    false = TFalse
    xif = if'
    lam sigma f = Lam sigma "x" (f (Var "x"))
    app = App
```

We also observe that the use of a fixed variable is justified by the fact that our algorithm uses at most one bound variable at the time. A correct dependently typed version of the free semantics requires the use of presheaves to ensure that the argument to Lam is stable under renaming. We refrain from presenting the details here. It is well known that this semantics is equationally sound.

3 Implementation of quote

We now proceed to implementing $\mathsf{quote} \in (\sigma \in \mathsf{Ty}) \to [\![\sigma]\!]_{\mathsf{set}} \to \mathsf{Tm}\ \sigma$.

To define $\mathsf{quote}^{\sigma\to\tau}$ we use enum^{σ}, which generates a decision tree whose leaves are all the elements of $[\![\sigma]\!]$, and $\mathsf{questions}^{\sigma}$, which generates a list of questions, i.e., elements of $[\![\sigma]\!] \to [\![\mathsf{Bool}]\!]$, based on answers to whom an element of $[\![\sigma]\!]$ can be looked up in the tree enum^{σ}. (Since our decision trees are perfectly balanced and we use the same list of questions along each branch of a tree, we can separate this list of questions from the tree.)

Decision trees $\mathsf{Tree} \in \mathsf{Ty} \to \star$ are provided by

```
data Tree a = Val a | Choice (Tree a) (Tree a) deriving (Show, Eq)
```

We will exploit the fact that Tree is a monad

```
instance Monad Tree where
    return = Val
    (Val a) >>= h = h a
    (Choice l r) >>= h = Choice (l >>= h) (r >>= h)
```

(`return` and `>>=` are Haskell for the unit resp. the bind or Kleisli extension operation of a monad) and hence a functor

```
instance Functor Tree where
    fmap h ds = ds >>= return . h
```

(`fmap` is Haskell for the action of a functor on morphisms).

It is convenient to use the function flatten which calculates the list of leaves of a given tree:

```
flatten :: Tree a -> [ a ]
flatten (Val a) = [ a ]
flatten (Choice l r) = (flatten l) ++ (flatten r)
```

We implement enum^σ and $\mathsf{questions}^\sigma$ by mutual induction on $\sigma \in \mathsf{Ty}$. The precise typings of the functions are $\mathsf{enum} \in (\sigma \in \mathsf{Ty}) \to \mathsf{Tree}\ [\![\sigma]\!]$ and $\mathsf{questions} \in (\sigma \in \mathsf{Ty}) \to [[\![\sigma]\!] \to [\![\mathtt{Bool}]\!]]$. As usual, Haskell cannot express those subtleties due to its lack of dependent types, but we can declare

```
enum :: Sem el => Ty -> Tree el
questions :: Sem el => Ty -> [ el -> el ]
```

The base case is straightforward: A boolean is true or false and to know which one it is it suffices to know it.

```
enum Bool = Choice (Val true) (Val false)

questions Bool = [ \ b -> b ]
```

The implementation of $\mathsf{enum}^{\sigma:\to\tau}$ and $\mathsf{questions}^{\sigma:\to\tau}$ proceeds from the idea that a function is determined by its graph: to know a function it suffices to know its value on all possible argument values. The main idea in the implementation of $\mathsf{enum}^{\sigma:\to\tau}$ is therefore to start with enum^τ and to duplicate the tree for each question in $\mathsf{questions}^\sigma$ using the bind of Tree:

```
enum (sigma :-> tau) =
    fmap (lam sigma) (mkEnum (questions sigma) (enum tau))

mkEnum :: Sem el => [ el -> el ] -> Tree el -> Tree (el -> el)
mkEnum [] es = fmap (\ e -> \ d -> e) es
mkEnum (q : qs) es = (mkEnum qs es) >>= \ f1 ->
                     (mkEnum qs es) >>= \ f2 ->
                     return (\ d -> xif (q d) (f1 d) (f2 d))
```

$\mathsf{questions}^{\sigma :\rightarrow \tau}$ produces the appropriate questions by enumerating σ and using questions from τ:

```
questions (sigma :-> tau) =
    [ \ f -> q (app f d) | d <- flatten (enum sigma),
                           q <- questions tau ]
```

As an example, the enumeration and questions for `Bool` $:\rightarrow$ `Bool` return:

```
Choice
   (Choice
       (Val (lam Bool (\ d -> xif d true  true))
       (Val (lam Bool (\ d -> xif d true  false)))
   (Choice
       (Val (lam Bool (\ d -> xif d false true ))
       (Val (lam Bool (\ d -> xif d false false)))
```

resp.

```
(\ f -> app f true :
   (\ f -> app f false :
       []))
```

We can look up an element in the decision tree for a type by answering all the questions, this is realized by the function find below. To define the domain of find precisely we define a relation between lists of answers and decisions trees $\diamond \subseteq [A] \times \mathsf{Tree}\ B$ inductively:

$$\frac{t \in B}{[] \diamond \mathsf{Val}\ t} \qquad \frac{a \in A \quad as \diamond l \quad as \diamond r}{a : as \diamond \mathsf{Choice}\ l\ r}$$

Now given $as \in [[\![\mathtt{Bool}]\!]]$, $ts \in \mathsf{Tree}\,[\![\sigma]\!]$ such that $as \diamond ts$ we obtain find $as\ ts \in [\![\sigma]\!]$, implemented in Haskell:

```
find :: Sem el => [ el ] -> Tree el -> el
find [] (Val t) = t
find (a : as) (Choice l r) = xif a (find as l) (find as r)
```

We are now ready to implement $\mathsf{quote}^{\sigma} \in [\![\sigma]\!]_{\mathsf{set}} \rightarrow \mathsf{Tm}\ \sigma$, with Haskell typing

```
quote :: Ty -> El -> Tm
```

by induction on $\sigma \in \mathsf{Ty}$. As usual, the base case is easy

```
quote Bool STrue  = TTrue
quote Bool SFalse = TFalse
```

$\mathsf{quote}^{\sigma :\rightarrow \tau}$ is more interesting: Our strategy is to map $\mathsf{quote}^{\tau} \circ f$ to the set-theoretic enum^{τ} and to then build a tree of `If` expressions by using the syntactic $\mathsf{questions}^{\sigma}$ in conjunction with the syntactic find:

```
quote (sigma :-> tau) (SLam _ f) =
    lam sigma (\ t -> find [ q t | q <- questions sigma ]
                           (fmap (quote tau . f) (enum sigma)))
```

(Notice that in Haskell it is inferred automatically which semantics is meant where.)

As already discussed in the introduction, we implement normalization $\mathsf{nf} \in (\sigma \in \mathsf{Ty}) \to \mathsf{Tm}\ \sigma \to \mathsf{Tm}\ \sigma$ by

```
nf :: Ty -> Tm -> Tm
nf sigma t = quote sigma (eval [] t)
```

Since we can infer types, we can implement $\mathsf{nf}' \in \mathtt{Tm} \to \mathsf{Maybe}\ (\Sigma_{\sigma \in \mathsf{Ty}} \mathsf{Tm}\ \sigma)$:

```
nf' :: Tm -> Maybe (Ty, Tm)
nf' t = do sigma <- infer [] t
           Just (sigma, nf sigma t)
```

We test our implementation with the example from the introduction:

```
b2b = Bool :-> Bool
once   = Lam b2b "f" (Lam Bool "x" (App (Var "f") (Var "x")))
twice  = Lam b2b "f" (Lam Bool "x" (App (Var "f")
                                   (App (Var "f") (Var "x"))))
thrice = Lam b2b "f"
           (Lam Bool "x" (App (Var "f")
                              (App (Var "f")
                                   (App (Var "f") (Var "x")))))
```

and convince ourselves that $(\mathsf{nf}'\ \mathtt{once} == \mathsf{nf}'\ \mathtt{thrice}) = \mathsf{true}$ but $(\mathsf{nf}'\ \mathtt{once} == \mathsf{nf}'\ \mathtt{twice}) = \mathsf{false}$. Since semantic equality is decidable we do not actually have to construct the normal forms to decide convertibility.

Since testing can only reveal the presence of errors we shall use the rest of this paper to prove that quote and hence nf behave correctly.

4 Correctness of quote

The main tool in our proof will be a notion of logical relations, a standard tool for the characterization of definable elements in models of typed lambda calculi since the pioneering work of Plotkin [13].

Let us agree to abbreviate $\mathsf{Tm}_{[]}\ \sigma$ by $\mathsf{Tm}\ \sigma$ and $[\![t]\!]_{\mathsf{set}}[]$ by $[\![t]\!]_{\mathsf{set}}$.

Definition 4.1 (Logical Relations). *We define a family of relations* $\mathsf{R}^\sigma \subseteq \mathsf{Tm}\ \sigma \times [\![\sigma]\!]_{\mathsf{set}}$ *by induction on* $\sigma \in \mathsf{Ty}$ *as follows:*

- $t\mathsf{R}^{\mathsf{Bool}}b$ *iff* $t =_{\beta\eta} \mathtt{True}$ *and* $b = \mathrm{true}$ *or* $t =_{\beta\eta} \mathtt{False}$ *and* $b = \mathrm{false}$*;*
- $t\mathsf{R}^{\sigma :\to \tau} f$ *iff, for all* u, d, $u\mathsf{R}^\sigma d$ *implies* $\mathsf{App}\ t\ u \mathsf{R}^\tau f\ d$.

Note that R is not indexed by contexts, logical relations only relate closed terms.

We extend logical relations to contexts: We write $\mathsf{Tm}\ \Gamma = [\![\Gamma]\!]_{\mathsf{syn}}$ for the type of closed substitutions. Now for $\Gamma \in \mathsf{Con}$ we define $\mathsf{R}^{\Gamma} \subseteq \mathsf{Tm}\ \Gamma \times [\![\Gamma]\!]_{\mathsf{set}}$ by:

$$\frac{}{[]\mathsf{R}^{[]}[]} \qquad \frac{\rho \mathsf{R}^{\Gamma} \rho' \quad t\mathsf{R}^{\sigma} d}{(x,t) : \rho\ \mathsf{R}^{(x,\sigma):\Gamma}\ (x,d) : \rho'}$$

Logical relations are invariant under $\beta\eta$-equality.

Lemma 4.2. *If $t\mathsf{R}^{\sigma}d$ and $t =_{\beta\eta} t'$, then $t'\mathsf{R}^{\sigma}d$.*

Logical relations obey the following Fundamental Theorem, a kind of soundness theorem for logical relations.

Lemma 4.3 (Fundamental Theorem of Logical Relations). *If $\theta\mathsf{R}^{\Gamma}\rho$ and $t \in \mathsf{Tm}_{\Gamma}\ \sigma$, then $[t]\ \theta\mathsf{R}^{\sigma}[\![t]\!]_{\mathsf{set}}\ \rho$. In particular, if $t \in \mathsf{Tm}\ \sigma$, then $t\mathsf{R}^{\sigma}[\![t]\!]_{\mathsf{set}}$.*

The main result required to see that quote is correct is the following lemma:

Lemma 4.4 (Main Lemma). *If $t\mathsf{R}^{\sigma}d$, then $t =_{\beta\eta} \mathsf{quote}^{\sigma}\ d$.*

The proof of this lemma is the subject of the next section.

By correctness of quote we mean that it inverts set-theoretic evaluation of typed closed terms.

Theorem 4.5 (Main Theorem). *If $t \in \mathsf{Tm}\ \sigma$, then $t =_{\beta\eta} \mathsf{quote}^{\sigma}\ [\![t]\!]_{\mathsf{set}}$.*

Proof. Immediate from the Fundamental Theorem and the Main Lemma. □

The (constructive) existence and correctness of quote has a number of straightforward important consequences.

Corollary 4.6 (Completeness). *If $t, t' \in \mathsf{Tm}\ \sigma$, then $[\![t]\!]_{\mathsf{set}} = [\![t']\!]_{\mathsf{set}}$ implies $t =_{\beta\eta} t'$.*

Proof. Immediate from the Main Theorem. □

From soundness (Proposition 2.1) and completeness together we get that $=_{\beta\eta}$ is decidable: checking whether $t =_{\beta\eta} t'$ reduces to checking whether $[\![t]\!]_{\mathsf{set}} = [\![t']\!]_{\mathsf{set}}$, which is decidable as $[\![-]\!]_{\mathsf{set}}$ is computable and equality in finite sets is decidable.

Corollary 4.7. *If $t, t' \in \mathsf{Tm}\ \sigma$, then $t =_{\beta\eta} t'$ iff $\mathsf{quote}^{\sigma}\ [\![t]\!]_{\mathsf{set}} = \mathsf{quote}^{\sigma}\ [\![t']\!]_{\mathsf{set}}$.*

Proof. Immediate from soundness (Proposition 2.1) and the Main Theorem. □

This corollary shows that $\mathsf{nf}^{\sigma} = \mathsf{quote}^{\sigma} \circ [\![-]\!]_{\mathsf{set}} : \mathsf{Tm}\ \sigma \to \mathsf{Tm}\ \sigma$ indeed makes sense as normalization function: apart from just delivering, for any given typed closed term, some $\beta\eta$-equal term, it is actually guaranteed to deliver the same term for t, t', if t, t' are $\beta\eta$-equal (morally, this is Church-Rosser for reduction-free normalization).

Note that although we only stated completeness and normalization for typed closed terms above, these trivially extend to all typed terms as open terms can always be closed up by lambda-abstractions and this preserves $\beta\eta$-equality.

Corollary 4.8. *If $t, t' \in \mathtt{Tm}\ \sigma$ and $[C]\ [(\mathtt{x}, t)] =_{\beta\eta} [C]\ [(\mathtt{x}, t')]$ for every $C \in \mathtt{Tm}_{[(\mathtt{x},\sigma)]}\ \mathtt{Bool}$, then $t =_{\beta\eta} t'$. Or, contrapositively, and more concretely, if $t, t' \in \mathtt{Tm}\ (\sigma_1 :\!\rightarrow \ldots :\!\rightarrow \sigma_n :\!\rightarrow \mathtt{Bool})$ and $t \neq_{\beta\eta} t'$, then there exist $u_1 \in \mathtt{Tm}\ \sigma_1$, $\ldots u_n \in \mathtt{Tm}\ \sigma_n$ such that*

$$\mathsf{nf}^{\mathsf{Bool}}\ (\mathtt{App}\ (\ldots(\mathtt{App}\ t\ u_1)\ \ldots)\ u_n) \neq \mathsf{nf}^{\mathsf{Bool}}\ (\mathtt{App}\ (\ldots\ (\mathtt{App}\ t'\ u_1)\ \ldots)\ u_n)$$

Proof. This corollary does not follow from the statement of the Main Theorem, but it follows from its proof. □

Corollary 4.9 (Maximal consistency). *If $t, t' \in \mathtt{Tm}\ \sigma$ and $t \neq_{\beta\eta} t'$, then from the equation $t = t'$ as an additional axiom one would derive* `True = False`.

Proof. Immediate from the previous corollary. □

5 Proof of the Main Lemma

We now present the proof of the main lemma which was postponed in the previous section. To keep the proof readable, we write $\mathsf{enum}_{\mathsf{set}}$, $\mathsf{questions}_{\mathsf{set}}$, $\mathsf{find}_{\mathsf{set}}$ to emphasize the uses of the set-theoretic semantics instances of enum, questions, find, while the free semantics instances will be written as $\mathsf{enum}_{\mathsf{syn}}$, $\mathsf{questions}_{\mathsf{syn}}$, $\mathsf{find}_{\mathsf{syn}}$. We use the fact that any functor F such as Tree has an effect on relations $R \subseteq A \times B$ denoted by $FR \subseteq FA \times FB$, which can be defined as:

$$\frac{p \in F\,\{(a, b) \in A \times B \mid a\ R\ b\}}{\mathsf{fmap\,fst}\,p\ FR\ \mathsf{fmap\,snd}\,p}$$

We first give the core of the proof and prove the lemmas this takes afterwards.

Proof (of the Main Lemma). By induction on σ.

- Case `Bool`: Assume $t\mathsf{R}^{\mathsf{Bool}}b$. Then either $t =_{\beta\eta}$ `True` and $b = \mathrm{true}$, in which case we have

  $$t =_{\beta\eta} \mathtt{True} = \mathsf{quote}^{\mathsf{Bool}}\ \mathrm{true} = \mathsf{quote}^{\mathsf{Bool}}\ b$$

  or $t =_{\beta\eta}$ `False` and $b = \mathrm{false}$, in which case we have

  $$t =_{\beta\eta} \mathtt{False} = \mathsf{quote}^{\mathsf{Bool}}\ \mathrm{false} = \mathsf{quote}^{\mathsf{Bool}}\ b$$

- Case $\sigma :\!\rightarrow \tau$: Assume $t\mathsf{R}^{\sigma :\rightarrow \tau}f$, for all u, d, $u\mathsf{R}^{\sigma}d$ implies $\mathtt{App}\ t\ u\mathsf{R}^{\tau}f\ d$. We have

  $$\begin{aligned} t &=_{\beta\eta} \mathtt{Lam}^{\sigma}\ \mathtt{x}\ (\mathtt{App}\ t\ (\mathtt{Var}\ \mathtt{x})) \\ &=_{\beta\eta} (\text{by Lemma 5.2 below}) \\ &\quad \mathtt{Lam}^{\sigma}\ \mathtt{x}\ (\mathtt{App}\ t\ (\mathsf{find}_{\mathsf{syn}}\ [q\ (\mathtt{Var}\ \mathtt{x}) \mid q \leftarrow \mathsf{questions}^{\sigma}_{\mathsf{syn}}]\ \mathsf{enum}^{\sigma}_{\mathsf{syn}})) \\ &= (\text{by Lemma 5.3 below}) \\ &\quad \mathtt{Lam}^{\sigma}\ \mathtt{x}\ (\mathsf{find}_{\mathsf{syn}}\ [q\ (\mathtt{Var}\ \mathtt{x}) \mid q \leftarrow \mathsf{questions}^{\sigma}_{\mathsf{syn}}] \\ &\qquad\qquad (\mathsf{fmap}\ (\mathtt{App}\ t)\ \mathsf{enum}^{\sigma}_{\mathsf{syn}})) \end{aligned}$$

$$\begin{aligned}
&=_{\beta\eta} \text{(by Sublemma)}\\
&\quad \mathtt{Lam}^{\sigma}\ \mathtt{x}\ (\mathsf{find}_{\mathsf{syn}}\ [q\ (\mathtt{Var}\ \mathtt{x}) \mid q \leftarrow \mathsf{questions}^{\sigma}_{\mathsf{syn}}]\\
&\qquad\qquad (\mathsf{fmap}\ (\mathsf{quote}^{\tau} \circ f)\ \mathsf{enum}^{\sigma}_{\mathsf{set}}))\\
&= \mathsf{quote}^{\sigma :\to \tau}\ f
\end{aligned}$$

The Sublemma is:

$$\mathsf{fmap}\ (\mathsf{App}\ t)\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}\ (\mathsf{Tree}\ =_{\beta\eta})\ \mathsf{fmap}\ (\mathsf{quote}^{\tau} \circ f)\ \mathsf{enum}^{\sigma}_{\mathsf{set}}$$

For proof, we notice that, by Lemma 5.1 (1) below for σ,

$$\mathsf{enum}^{\sigma}_{\mathsf{syn}}\ (\mathsf{Tree}\ \mathsf{R}^{\sigma})\ \mathsf{enum}^{\sigma}_{\mathsf{set}}$$

Hence, by assumption and the fact that fmap commutes with the effect on relations

$$\mathsf{fmap}\ (\mathsf{App}\ t)\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}\ (\mathsf{Tree}\ \mathsf{R}^{\tau})\ \mathsf{fmap}\ f\ \mathsf{enum}^{\sigma}_{\mathsf{set}}$$

Hence, by IH of the Lemma for τ,

$$\mathsf{fmap}\ (\mathsf{App}\ t)\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}\ (\mathsf{Tree}\ =_{\beta\eta})\ \mathsf{fmap}\ (\mathsf{quote}^{\tau} \circ f)\ \mathsf{enum}^{\sigma}_{\mathsf{set}}$$

□

The proof above used two lemmas. One is essentially free, but the other is technical.

Lemma 5.1 ("Free" Lemma).

1. $\mathsf{enum}^{\sigma}_{\mathsf{syn}}\ (\mathsf{Tree}\ \mathsf{R}^{\sigma})\ \mathsf{enum}^{\sigma}_{\mathsf{set}}$.
2. $\mathsf{questions}^{\sigma}_{\mathsf{syn}}\ [\mathsf{R}^{\sigma} \to \mathsf{R}^{\mathtt{Bool}}]\ \mathsf{questions}^{\sigma}_{\mathsf{set}}$.

Proof. The proof is simultaneous for (1) and (2) by induction on σ.

- Case `Bool`: Trivial.
- Case $\sigma :\to \tau$: Proof of (1) uses IH (2) for σ and IH (1) for τ; proof of (2) uses IH (1) for σ and IH (2) for τ.

□

Lemma 5.2 (Technical Lemma). *For* $t \in \mathsf{Tm}_{\Gamma}\ \sigma$*:*

$$t =_{\beta\eta} \mathsf{find}_{\mathsf{syn}}\ [q\ t \mid q \leftarrow \mathsf{questions}^{\sigma}_{\mathsf{syn}}]\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}$$

Proof. By induction on σ.

- Case `Bool`:

$$\begin{aligned}
t &= \mathtt{if'}\ t\ \mathtt{True}\ \mathtt{False}\\
&= \mathtt{if'}\ t\ (\mathsf{find}_{\mathsf{syn}}\ []\ (\mathsf{Val}\ \mathtt{True}))\ (\mathsf{find}_{\mathsf{syn}}\ []\ (\mathsf{Val}\ \mathtt{False}))\\
&= \mathsf{find}_{\mathsf{syn}}\ [t]\ (\mathsf{Choice}\ (\mathsf{Val}\ \mathtt{True})\ (\mathsf{Val}\ \mathtt{False}))\\
&= \mathsf{find}_{\mathsf{syn}}\ [q\ t \mid q \leftarrow \mathsf{questions}^{\mathtt{Bool}}_{\mathsf{syn}}]\ \mathsf{enum}^{\mathtt{Bool}}_{\mathsf{syn}}
\end{aligned}$$

– Case $\sigma :\!\rightarrow \tau$:

$$\begin{aligned}
t &=_{\beta\eta} \mathtt{Lam}^{\sigma}\ z\ (\mathtt{App}\ t\ (\mathtt{Var}\ z)) \qquad (z \text{ fresh wrt } \Gamma)\\
&=_{\beta\eta} (\text{by IH for } \sigma)\\
&\quad \mathtt{Lam}^{\sigma}\ z\ (\mathtt{App}\ t\ (\mathsf{find}_{\mathsf{syn}}\ [q\ (\mathtt{Var}\ z) \mid q \leftarrow \mathsf{questions}^{\sigma}_{\mathsf{syn}}]\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}))\\
&=_{\beta\eta} (\text{by Sublemma below})\\
&\quad \mathtt{Lam}^{\sigma}\ z\\
&\qquad (\mathsf{find}_{\mathsf{syn}}\ [q\ (\mathtt{App}\ t\ u) \mid u \leftarrow \mathsf{flatten}\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}, q \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ (\mathtt{Var}\ z))\ (\mathsf{mkenum}_{\mathsf{syn}}\ \mathsf{questions}^{\sigma}_{\mathsf{syn}}\ \mathsf{enum}^{\tau}_{\mathsf{syn}})))\\
&= (\text{by Lemma 5.3})\\
&\quad \mathsf{find}_{\mathsf{syn}}\ [q\ (\mathtt{App}\ t\ u) \mid u \leftarrow \mathsf{flatten}\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}, q \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ \mathtt{Lam}^{\sigma}\ z\ (\mathsf{fmap}\ (\lambda g\ g\ (\mathtt{Var}\ z))\\
&\qquad\qquad (\mathsf{mkenum}_{\mathsf{syn}}\ \mathsf{questions}^{\sigma}_{\mathsf{syn}}\ \mathsf{enum}^{\tau}_{\mathsf{syn}})))\\
&= \mathsf{find}_{\mathsf{syn}}\ [q\ (\mathtt{App}\ t\ u) \mid u \leftarrow \mathsf{flatten}\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}, q \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\quad (\mathsf{fmap}\ (\lambda g\ \mathtt{Lam}^{\sigma}\ z\ (g\ (\mathtt{Var}\ z)))\ (\mathsf{mkenum}_{\mathsf{syn}}\ \mathsf{questions}^{\sigma}_{\mathsf{syn}}\ \mathsf{enum}^{\tau}_{\mathsf{syn}}))\\
&=_{\alpha} \mathsf{find}_{\mathsf{syn}}\ [q\ (\mathtt{App}\ t\ u) \mid u \leftarrow \mathsf{flatten}\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}, q \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\quad (\mathsf{fmap}\ (\lambda g\ \mathtt{Lam}^{\sigma}\ \mathtt{x}\ (g\ (\mathtt{Var}\ \mathtt{x})))\ (\mathsf{mkenum}_{\mathsf{syn}}\ \mathsf{questions}^{\sigma}_{\mathsf{syn}}\ \mathsf{enum}^{\tau}_{\mathsf{syn}}))\\
&= \mathsf{find}_{\mathsf{syn}}\ [q\ t \mid q \leftarrow \mathsf{questions}^{\sigma:\rightarrow\tau}_{\mathsf{syn}}]\ \mathsf{enum}^{\sigma:\rightarrow\tau}_{\mathsf{syn}}
\end{aligned}$$

The α-conversion step is justified by the easily verifiable fact that the leaves of $\mathsf{mkenum}_{\mathsf{syn}}\ \mathsf{questions}^{\sigma}_{\mathsf{syn}}\ \mathsf{enum}^{\tau}_{\mathsf{syn}}$ are closed "terms with a hole" (which has the implication that $\mathsf{mkenum}_{\mathsf{syn}}\ \mathsf{questions}^{\sigma}_{\mathsf{syn}}\ \mathsf{enum}^{\tau}_{\mathsf{syn}} \in \mathsf{Tree}\ (\mathsf{Tm}_{\Delta}\ \sigma \rightarrow \mathsf{Tm}_{\Delta}\ \tau)$ is natural in Δ).
The sublemma is: If $u \in \mathsf{Tm}_{\Delta}\ \sigma$, $qs \in [\mathsf{Tm}_{\Delta}\ \sigma \rightarrow \mathsf{Tm}_{\Delta}\ \mathtt{Bool}]$, $us \in \mathsf{Tree}\ (\mathsf{Tm}_{\Delta}\ \sigma)$ and $qs \diamond us$, then

$$\begin{aligned}
&\mathtt{App}\ t\ (\mathsf{find}_{\mathsf{syn}}\ [q\ u \mid q \leftarrow qs]\ us)\\
&=_{\beta\eta} \mathsf{find}_{\mathsf{syn}}\ [q\ (\mathtt{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ us, q \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\tau}_{\mathsf{syn}}))
\end{aligned}$$

The proof is by induction on $qs \diamond us$.

- Case $[] \diamond \mathsf{Val}\ u^{\star}$:

$$\begin{aligned}
&\mathtt{App}\ t\ (\mathsf{find}_{\mathsf{syn}}\ [q\ u \mid q \leftarrow []]\ (\mathsf{Val} u^{\star}))\\
&= \mathtt{App}\ t\ u^{\star}\\
&=_{\beta\eta} (\text{by IH of the Lemma for } \tau)\\
&\quad \mathsf{find}_{\mathsf{syn}}\ [q\ (\mathtt{App}\ t\ u^{\star}) \mid q \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\ \mathsf{enum}^{\tau}_{\mathsf{syn}}\\
&= \mathsf{find}_{\mathsf{syn}}\ [q\ (\mathtt{App}\ t\ u^{\star}) \mid q \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{fmap}\ (\lambda v \lambda u'\ v)\ \mathsf{enum}^{\tau}_{\mathsf{syn}}))\\
&= \mathsf{find}_{\mathsf{syn}}\ [q\ (\mathtt{App}\ t\ u') \mid u' \leftarrow [u^{\star}], q \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ []\ \mathsf{enum}^{\tau}_{\mathsf{syn}}))
\end{aligned}$$

- Case $q : qs \diamond \mathsf{Choice}\ l\ r$:

$$\begin{aligned}
&\mathsf{App}\ t\ (\mathsf{find}_{\mathsf{syn}}\ [q'\ u \mid q' \leftarrow q : qs]\ (\mathsf{Choice}\ l\ r))\\
=\ &\mathsf{App}\ t\ (\mathtt{if}'\ (q\ u)\ (\mathsf{find}_{\mathsf{syn}}\ [q'\ u \mid q' \leftarrow qs]\ l)\\
&\qquad\qquad\qquad (\mathsf{find}_{\mathsf{syn}}\ [q'\ u \mid q' \leftarrow qs]\ r))\\
=_{\beta\eta}\ &\mathtt{if}'\ (q\ u)\ (\mathsf{App}\ t\ (\mathsf{find}_{\mathsf{syn}}\ [q'\ u \mid q' \leftarrow qs]\ l))\\
&\qquad\qquad (\mathsf{App}\ t\ (\mathsf{find}_{\mathsf{syn}}\ [q'\ u \mid q' \leftarrow qs]\ r))\\
=_{\beta\eta}\ &(\text{by IH of the Sublemma for } qs \diamond l,\ qs \diamond r)\\
&\mathtt{if}'\ (q\ u)\\
&\quad (\mathsf{find}_{\mathsf{syn}}\ [q'\ (\mathsf{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ l, q' \leftarrow \mathsf{questions}^{\mathcal{T}}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\mathcal{T}}_{\mathsf{syn}})))\\
&\quad (\mathsf{find}_{\mathsf{syn}}\ [q'\ (\mathsf{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ r, q' \leftarrow \mathsf{questions}^{\mathcal{T}}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\mathcal{T}}_{\mathsf{syn}})))\\
=\ &\mathsf{find}_{\mathsf{syn}}\ []\ (\mathsf{Val}\ (\mathtt{If}'\ (q\ u)\\
&\quad (\mathsf{find}_{\mathsf{syn}}\ [q'\ (\mathsf{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ l, q' \leftarrow \mathsf{questions}^{\mathcal{T}}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\mathcal{T}}_{\mathsf{syn}})))\\
&\quad (\mathsf{find}_{\mathsf{syn}}\ [q'\ (\mathsf{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ r, q' \leftarrow \mathsf{questions}^{\mathcal{T}}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\mathcal{T}}_{\mathsf{syn}})))))\\
=_{\beta\eta}\ &(\text{by twice Lemma 5.4})\\
&\mathsf{find}_{\mathsf{syn}}\ ([q'\ (\mathsf{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ l, q' \leftarrow \mathsf{questions}^{\mathcal{T}}_{\mathsf{syn}}]\\
&\qquad +\!\!+\ ([q'\ (\mathsf{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ r, q' \leftarrow \mathsf{questions}^{\mathcal{T}}_{\mathsf{syn}}]\\
&\qquad +\!\!+\ []))\\
&\quad ((\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\mathcal{T}}_{\mathsf{syn}})) \gg\!= \lambda v_0\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\mathcal{T}}_{\mathsf{syn}})) \gg\!= \lambda v_1\\
&\qquad \mathsf{Val}\ (\mathtt{If}'\ (q\ u)\ v_0\ v_1))\\
=\ &\mathsf{find}_{\mathsf{syn}}\ ([q'\ (\mathsf{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ l, q' \leftarrow \mathsf{questions}^{\mathcal{T}}_{\mathsf{syn}}]\\
&\qquad +\!\!+\ [q'\ (\mathsf{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ r, q' \leftarrow \mathsf{questions}^{\mathcal{T}}_{\mathsf{syn}}])\\
&\quad (\mathsf{fmap}\ (\lambda g\ g\ u)\\
&\qquad ((\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\mathcal{T}}_{\mathsf{syn}}) \gg\!= \lambda g_0\\
&\qquad\quad (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\mathcal{T}}_{\mathsf{syn}} \gg\!= \lambda g_1\\
&\qquad\quad \mathsf{Val}\ (\lambda u'\ \mathtt{If}'\ (q\ u')\ (g_0\ u')\ (g_1\ u')))))\\
=\ &\mathsf{find}_{\mathsf{syn}}\ [q'\ (\mathsf{App}\ t\ u') \mid t \leftarrow (\mathsf{flatten}\ l) +\!\!+ (\mathsf{flatten}\ r),\\
&\qquad\qquad\qquad\qquad q' \leftarrow \mathsf{questions}^{\mathcal{T}}_{\mathsf{syn}}]\\
&\quad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ (q : qs)\ \mathsf{enum}^{\mathcal{T}}_{\mathsf{syn}}))\\
=\ &\mathsf{find}_{\mathsf{syn}}\ [q'\ (\mathsf{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ (\mathsf{Choice}\ l\ r),\\
&\qquad\qquad\qquad\qquad q' \leftarrow \mathsf{questions}^{\mathcal{T}}_{\mathsf{syn}}]\\
&\quad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ (q : qs)\ \mathsf{enum}^{\mathcal{T}}_{\mathsf{syn}}))
\end{aligned}$$

□

We have used two lemmas, which are easy to prove:

Lemma 5.3. *If* $as \in [A]$, $us \in \mathsf{Tree}\ B$ *and* $as \diamond us$, *then*

$$\mathsf{find_{syn}}\ as\ (\mathsf{fmap}\ f\ us) =_{\beta\eta} f\ (\mathsf{find_{syn}}\ as\ us)$$

Proof. Simple induction on $as \diamond us$.

Lemma 5.4. *If* $as, bs \in [\mathsf{Tm}_\Gamma\ \mathtt{Bool}]$, $ts \in \mathsf{Tree}\ (\mathsf{Tm}_\Gamma \sigma)$, $h \in \mathsf{Tm}_\Gamma\ \sigma \to \mathsf{Tree}\ (\mathsf{Tm}_\Gamma\ \tau)$, $as \diamond ts$ *and, for all* $u \in \mathsf{Tm}_\Gamma\ \sigma$, $bs \diamond h\ u$, *then*

$$\mathsf{find_{syn}}\ (as \mathbin{+\!\!+} bs)\ (ts \gg\!= h) =_{\beta\eta} \mathsf{find_{syn}}\ bs\ (h\ (\mathsf{find_{syn}}\ as\ ts))$$

Proof. By induction on $as \diamond ts$.

– Case $[] \diamond \mathsf{Val}\ t$:

$$\begin{aligned}
&\mathsf{find_{syn}}\ ([] \mathbin{+\!\!+} bs)\ ((\mathsf{Val}\ t) \gg\!= h)\\
&= \mathsf{find_{syn}}\ bs\ ((\mathsf{Val}\ t) \gg\!= h)\\
&= \mathsf{find_{syn}}\ bs\ (h\ t)\\
&= \mathsf{find_{syn}}\ bs\ (h\ (\mathsf{find_{syn}}\ []\ (\mathsf{Val}\ t)))
\end{aligned}$$

– Case $a : as \diamond \mathsf{Choice}\ l\ r$:

$$\begin{aligned}
&\mathsf{find_{syn}}\ ((a : as) \mathbin{+\!\!+} bs)\ ((\mathsf{Choice}\ l\ r) \gg\!= h)\\
&= \mathsf{find_{syn}}\ (a : (as \mathbin{+\!\!+} bs))\ ((\mathsf{Choice}\ l\ r) \gg\!= h)\\
&= \mathtt{If'}\ a\ (\mathsf{find_{syn}}\ (as \mathbin{+\!\!+} bs)\ (l \gg\!= h))\ (\mathsf{find_{syn}}\ (as \mathbin{+\!\!+} bs)\ (r \gg\!= h))\\
&= (\text{by IH for } as \diamond l,\ as \diamond r)\\
&\quad \mathtt{If'}\ a\ (\mathsf{find_{syn}}\ bs\ (h\ (\mathsf{find_{syn}}\ as\ l)))\ (\mathsf{find_{syn}}\ bs\ (h\ (\mathsf{find_{syn}}\ as\ r)))\\
&=_{\beta\eta} \mathsf{find_{syn}}\ bs\ (h\ (\mathtt{If'}\ a\ (\mathsf{find_{syn}}\ as\ l)\ (\mathsf{find_{syn}}\ as\ r)))\\
&= \mathsf{find_{syn}}\ bs\ (h\ (\mathsf{find_{syn}}\ (a : as)\ (\mathsf{Choice}\ l\ r)))
\end{aligned}$$

□

6 Discussion and Further Work

Instead of decision trees we could have used a direct encoding of the graph of a function, we call this the truth-table semantics. However, this approach leads not only too much longer normal forms but also the semantic equality is less efficient. On the other hand it is possible to go further and use Binary Decision Diagrams (BDDs) [7] instead of decision trees. We plan to explore this in further work and also give a detailed analysis of the normal forms returned by our algorithm.

We have argued that $\lambda^{\to 2}$ is the simplest λ-calculus with closed types, however we are confident that the technique described here works also for closed types in $\lambda^{0+1\times\to}$ (the finitary λ-calculus). We leave this extension for a journal version of this work.

One can go even further and implement finitary Type Theory, i.e. $\lambda^{012\Sigma\Pi}$ (note that $A + B = \Sigma x \in 2.\ \mathtt{If}\ x\ A\ B$). This could provide an interesting base for a type-theoretic hardware description and verification language.

The approach presented here works only for calculi without type variables. It remains open to see whether this approach can be merged with the the standard techniques for NBE for systems with type variables, leading to an alternative proof of completeness and maybe even finite completeness for the calculi discussed above.

References

1. T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *Proc. of 16th Annual IEEE Symposium on Logic in Computer Science,* pages 303–310. IEEE CS Press, 2001.
2. T. Altenkirch, M. Hofmann, and T. Streicher. Categorical reconstruction of a reduction free normalization proof. In D. Pitt, D. E. Rydeheard, and P. Johnstone, editors, *Category Theory and Computer Science,* number 953 in LNCS, pages 182–199, Springer-Verlag, 1995.
3. T. Altenkirch, M. Hofmann, and T. Streicher. Reduction-free normalisation for a polymorphic system. In *Proc. of 11th Annual IEEE Symposium on Logic in Computer Science,* pages 98–106. IEEE CS Press, 1996.
4. T. Altenkirch, M. Hofmann, and T. Streicher. Reduction-free normalisation for system *F.* Unpublished, available on WWW at `http://www.cs.nott.ac.uk/~txa/publ/f97.pdf`, 1997.
5. V. Balat. *Une étude des sommes fortes : isomorphismes et formes normales.* PhD thesis, Université Denis Diderot, 2002.
6. U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proc. of 6th Annual IEEE Symposium on Logic in Computer Science,* pages 202–211. IEEE CS Press, 1991.
7. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers,* 35(8):677–691, 1986.
8. D. J. Dougherty and R. Subrahmanyam. Equality between functionals in the presence of coproducts. *Information and Computation,* 157(1–2):52–83, 2000.
9. P. Dybjer and A. Filinski. Normalization and partial evaluation. In G. Barthe et al., editors, *Applied Semantics,* number 2395 in LNCS, pages 137–192. Springer-Verlag, 2002.
10. N. Ghani. *Adjoint Rewriting.* PhD thesis, LFCS, Univ. of Edinburgh, 1995.
11. N. Ghani. $\beta\eta$-equality for coproducts. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Typed Lambda Calculus and Applications,* number 902 in LNCS, pages 171–185. Springer-Verlag, 1995.
12. C. McBride and J. McKinna. The view from the left. To appear in the Journal of Functional Programming, Special Issue: Dependent Type Theory Meets Programming Practice, 2004.
13. G. D. Plotkin. Lambda-definability and logical relations. Memorandum SAI-RM-4, Lab. for Artif. Intell., Univ. of Edinburgh, Oct. 1973.

Basic Pattern Matching Calculi: a Fresh View on Matching Failure

Wolfram Kahl

Department of Computing and Software, McMaster University
http://www.cas.mcmaster.ca/~kahl/

Abstract. We propose pattern matching calculi as a refinement of λ-calculus that integrates mechanisms appropriate for fine-grained modelling of non-strict pattern matching.
Compared with the functional rewriting strategy usually employed to define the operational semantics of pattern matching in non-strict functional programming languages like Haskell or Clean, our pattern matching calculi achieve the same effects using simpler and more local rules.
The main device is to embed into expressions the separate syntactic category of matchings; the resulting language naturally encompasses pattern guards and Boolean guards as special cases.
By allowing a confluent reduction system and a normalising strategy, these pattern matching calculi provide a new basis for operational semantics of non-strict programming languages and also for implementations.

1 Introduction

The operational semantics of functional programming languages is usually explained via special kinds of λ-calculi and term rewriting systems (TRSs). One way to look at the relation between these two approaches is to consider λ-calculi as *internalisations* of term rewriting systems: λ-abstraction internalises *applicative* TRSs (where each function is defined in a single "equation" the left-hand side of which is an application of the function symbol to only variables), and fixedpoint combinators internalise recursive function definitions.

In addition to these two features, modern functional programming languages support function definitions based on *pattern matching*. A *pattern* is an expression built only from variables and *constructors* — in the context of applicative TRSs, constructors are function symbols that never occur as head of a rule. In the functional programming context, constructors are introduced by datatype definitions, for example the list constructors "`[]`" (empty list) and "`_ : _`" ("cons", non-empty list construction from head and tail).

In the term rewriting view, a function is *defined by pattern matching* if it is defined by a group of rules, each having as left-hand side an application of the defined function symbol to patterns. Definitions using pattern matching are "processed sequentially"; for an example assume the following definition:

© Springer-Verlag Berlin Heidelberg 2004

```
isEmptyList (x : xs) = False
isEmptyList ys       = True
```

The second line is not an equation valid for arbitrary ys, but a rule that is only considered if the left-hand side of the first rule gives rise to a mismatch — here, the only value of ys for which this is the case is the empty list [].

Function definitions with patterns as arguments on the left-hand sides are typical of modern functional programming languages. In non-strict languages like Haskell, Clean, or Miranda, the operational semantics of pattern matching is quite complex; usually it is formulated as the *functional rewriting strategy,* which is a rather involved priority rewriting strategy [19, sect. 4.7.1].

In the case of, for example, Haskell, the operational semantics of pattern matching is defined via the special instance of pattern matching in case expressions. For isEmptyList, the above definition is considered as shorthand for:

```
isEmptyList zs = case zs of (x : xs) -> False
                            ys       -> True
```

Case expressions can be seen as an internalisation of pattern matching that is not quite analogous to the internalisation of function abstraction in λ-calculus; the important difference is that, in comparison with λ-abstractions, case expressions contain not only the abstracted pattern matchings, but also an additional application to an argument. To further complicate matters, Boolean *guards* and, more recently, *pattern guards* interfere with the "straightforward" pattern matching.

In this paper we present a new calculus that cleanly internalises pattern matching by drawing a clearer distinction between the aspects involved. For that purpose, we essentially liberate the case expression from its rigidly built-in application, generalising the special syntactic category of case alternatives into the new syntactic category of *matchings* that incorporates all aspects of pattern matching, as opposed to the (preserved) syntactical category of *expressions* that now is mostly concerned with pattern construction and function application.

This allows straightforward internalisation of pattern matching definitions without having to introduce new variables like zs for the case variant:

$$\texttt{isEmptyList} = \{\!|\, (x : xs) \mapsto\!\!\!> \mathsf{False} \,|\, ys \mapsto\!\!\!> \mathsf{True} \,|\!\}$$

In addition, using the pattern matching calculus as basis of functional programming has advantages both for expressivity and reasoning about programs.

With respect to reasoning, the full internalisation of pattern matching eliminates the problem of all priority systems that what is written down as an unconditional equation only applies to certain patterns "left over" from higher-priority equations defining the same function. The usual justification for allowing this non-orthogonality is that otherwise the number of equations would explode. Our matching language allows direct transliteration of such prioritised definitions without additional cost, and even includes the means to factor out more commonalities than is possible in priority rewriting systems. The syntactical features necessary to achieve this turn out to be sufficient to include both Boolean guards and pattern guards as special cases. This gives the language a boost in expressivity over systems directly based on term rewriting, and at the same time keeps the system simple and uniform.

A noteworthy result is that by providing two variants of a simple rule concerned with results of matching failure, we obtain two interesting systems, both confluent and equipped with the same normalising strategy:

- The first mirrors exactly the definition of pattern matching in, e.g., Haskell, which corresponds to the functional rewrite strategy modified by treating matching against non-covered alternatives as a run-time error. It is well known that the functional strategy, considered as a term rewriting strategy, is not normalising, so there are certain terms that, translated into our first system, have a normal form that corresponds to such run-time errors.
- The second system is a refinement of the first in that it preserves all terminating reductions not ending in a run-time errors, and also has such "successful" reductions for some terms that reduce to run-time errors in the first system.

Similar mechanisms have been proposed in the literature, see Sect. 8; we feel that the setting of the pattern matching calculus helps to clarify the issues involved and provides an attractive environment for describing and analysing such alternative treatments of matching failure.

After presenting the abstract syntax, we show how the pattern matching calculus encodes λ-calculus and Haskell pattern matching including Boolean and pattern guards. Sect. 4 presents the reduction rules, which are applied to selected examples in Sect. 5. In Sect. 6 we summarise the mechanised confluence proof, and Sect. 7 is devoted to the normalising reduction strategy. Sect. 8 discusses related work. Details omitted here for reasons of space can be found in [10].

2 Abstract Syntax

The pattern matching calculus, from now on usually abbreviated PMC, has two major syntactic categories, namely *expressions* and *matchings*. These are defined by mutual recursion. When considering the analogy to functional programs, only expressions of the pattern matching calculus correspond to expressions of functional programming languages. Matchings can be seen as a generalisation of groups of case alternatives. Operationally, matchings can be "waiting for argument supply", or they can be *saturated;* saturated matchings can *succeed* and then *return* an expression, or they can *fail. Patterns* form a separate syntactic category that will be used to construct pattern matchings.

We now present the abstract syntax of the pattern matching calculus with some intuitive explanation of the intended meaning of the constructs.

As base sets, we use Var as the set of *variables,* and Constr as the set of *constructors.* For the purpose of our examples, numbers are assumed to be elements of Constr and are used only in zero-ary constructions (which are written without parentheses). Constructors will, as usual, be used to build both patterns and expressions. Indeed, one might consider Pat as a subset of Expr.

The following summarises the abstract syntax of PMC:

Pat	::= Var	variable
	| Constr(Pat, ..., Pat)	constructor pattern
Expr	::= Var	variable
	| Constr(Expr, ..., Expr)	constructor application
	| Expr Expr	function application
	| $\{\!\mid$ Match $\mid\!\}$	matching abstraction
	| $\oslash$	empty expression
Match	::= $\rceil$Expr$\lceil$	expression matching
	| ↯	failure
	| Pat $\mapsto$ Match	pattern matching
	| Expr $\triangleright$ Match	argument supply
	| Match **|** Match	alternative

Patterns are built from variables and constructor applications. All variables occurring in a pattern are *free* in that pattern; for every pattern p : Pat, we denote its set of free variables by $\mathsf{FV}(p)$. In the following, we silently restrict *all* patterns to be *linear,* i.e., not to contain more than one occurrence of any variable.

Expressions are the syntactic category that embodies the term construction aspects; besides variables, constructor application and function application, we also have the following special kinds of expressions:

– Every matching m gives rise to the *matching abstraction* (matching expression) $\{\!\mid m \mid\!\}$, which might be read "match m".
  If the matching m is *unsaturated,* i.e., "waiting for arguments", then $\{\!\mid m \mid\!\}$ abstracts m into a function.
  If m is a saturated matching, then it can either succeed or fail; if it succeeds, then $\{\!\mid m \mid\!\}$ reduces to the value "returned" by m; otherwise, $\{\!\mid m \mid\!\}$ is considered ill-defined.
– we call $\oslash$ the *empty expression;* it results from matching failures — according to the above, it could also be called the "ill-defined expression".
  We use the somewhat uncommitted name "empty expression" since we will consider two interpretations of $\oslash$:
  • It can be a "manifestly undefined" expression equivalent to non-termination — following the common view that divergence is semantically equivalent to run-time errors.
  • It can be a special "error" value propagating matching failure ↯, considered as an "exception" through the syntactic category of expressions.

None of the expression constructors binds any variables; we overload the FV(_) notation and denote for an expression e : Expr its set of free variables by $\mathsf{FV}(e)$.

For the purposes of pattern matching, constructor applications of the same constructor, but with different arities, are considered incompatible.

Matchings are the syntactic category embodying the pattern analysis aspects:

- For an expression e : Expr, the *expression matching* $\rceil e \lceil$ always succeeds and returns e, so we propose to read it *"return e"*.
- $\lightning$ is the matching that always fails.
- The *pattern matching* $p \mapsto m$ waits for supply of one argument more than m; this pattern matching can be understood as succeeding on instances of the (linear) pattern p : Pat and then continuing to behave as the resulting instance of the matching m : Match. It roughly corresponds to a single case alternative in languages with case expressions.
- *argument supply* $a \rhd m$ is the matching-level incarnation of function application, with the argument on the left, and the matching it is supplied to on the right. It saturates the first argument m is waiting for.
  The inclusion of argument supply into the calculus is an important source of flexibility in the design of the reduction system.
- the *alternative* $m_1 \| m_2$ will in this paper be understood sequentially: it behaves like m_1 until this fails, and only then it behaves like m_2.

Pattern matching $p \mapsto m$ binds all variables occurring in p, so $\mathsf{FV}(p \mapsto m) = \mathsf{FV}(m) - \mathsf{FV}(p)$, letting $\mathsf{FV}(m)$ denote the set of free variables of a matching m. Pattern matching is the only variable binder in this calculus — taking this into account, the definitions of free variables, bound variables, and substitution are as usual. Note that there are no matching variables; variables can only occur as patterns or as expressions.

We will omit the parentheses in matchings of the shape $a \rhd (p \mapsto m)$ since there is only one way to parse $a \rhd p \mapsto m$ in PMC.

3 Examples

Even though we have not yet introduced PMC reduction, the explanations of the syntax of PMC in the previous section should allow the reader to understand the examples presented in this section. We first show the natural embedding of the untyped λ-calculus into PMC and then continue to give translations for Haskell function definitions first using pattern matching only, then together with Boolean guards and finally together with pattern guards,

It is easy to see that the pattern matching calculus includes the *λ-calculus*. Variables and function application are translated directly, and λ-abstraction is a matching abstraction over a pattern matching that has a single-variable pattern and a result matching that immediately returns the body:

$$\lambda\ v\ .\ e := \{\!|\, v \mapsto \rceil e \lceil \,|\!\}$$

In Sect. 5 we shall see that this embedding also preserves reducibility.

As an example for the translation of *Haskell programs* into PMC, we show one that also serves as an example for non-normalisation of the functional rewriting strategy; with this program and the additional definition `bot = bot`, the functional strategy loops (detected by some implementations) on evaluation of the expression `f bot (3:[])`, although "obviously" it "could" reduce to `2`:

```
f (x:xs) []     = 1
f ys     (v:vs) = 2
```

For translation into PMC, we have to decide how we treat bot. We could translate it directly into an application of a fixedpoint combinator to the identity function; if we call the resulting expression $\bot$, then $\bot$ gives rise to cyclic reductions. In this case, we obtain for f bot (3:[]) the following expression:

$$\{\!|\,((x : xs) \Mapsto [\,] \Mapsto \rceil 1 \lceil) \,\pmb{|}\, (ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil)\,|\!\}\ \bot\ (3 : [\,])$$

A different possibility is to recognise that the above "definition" of bot has as goal to produce an undefined expression; if the empty expression $\oslash$ is understood as undefined, then we could use that.

We will investigate reduction of both possibilities below, in Sect. 5.

In several functional programming languages, *Boolean guards* may be added after the pattern part of a definition equation; the failure of such a guard has the same effect as pattern matching failure: if more definition equations are present, the next one is tried. For example:

```
g (x:xs) | x > 5 = 2
g ys             = 3
```

Translation into a case-expression turns such a guard into a match to the Boolean constructor True and a default branch that redirects mismatches to the next line of the definition. In PMC, we do not need to make the mismatch case explicit, but can directly translate from the Haskell formulation. The above function g therefore corresponds to the following PMC expression:

$$\{\!|\,((x : xs) \Mapsto (x > 5) \rhd \mathsf{True} \Mapsto \rceil 2 \lceil) \,\pmb{|}\, (ys \Mapsto \rceil 3 \lceil)\,|\!\}$$

A generalisation of Boolean guards are *pattern guards* [6]; these incorporate not only the decision aspect of Boolean guards, but also the variable binding aspect of pattern matching. In PMC, both can be represented as *saturated patterns,* i.e., as pattern matchings that already have an argument supplied to them. For a pattern guard example, we use Peyton Jones' clunky:

```
clunky env v1 v2 |  Just r1 <- lookup env v1
                  , Just r2 <- lookup env v2  = r1 + r2
                 | otherwise                  = v1 + v2
```

We attempt analogous layout for the PMC expression corresponding to the functionclunky (with appropriate conventions, we could omit more parentheses):

$$\begin{array}{ll}\{\!|\, env \Mapsto v_1 \Mapsto v_2 \Mapsto & ((lookup\ env\ v_1 \rhd Just(r_1) \Mapsto \\ & \quad lookup\ env\ v_2 \rhd Just(r_2) \Mapsto \rceil r_1 + r_2 \lceil) \\ & \pmb{|}\, \rceil v_1 + v_2 \lceil \qquad\qquad\qquad\qquad)\,|\!\}\end{array}$$

Irrefutable patterns, in Haskell indicated by the prefix "~", match lazily, i.e., matching is delayed until one of the component variables is needed. There are no special provisions for irrefutable patterns in PMC; they have to be translated in essentially the same way as in Haskell. For example, with *body* possibly containing occurrences of x and xs, the definition:

```
q ~(x:xs) = body
```

expands into the following shape according to the Haskell report:

```
q = \ v -> (\x -> \ xs -> body )(case v of (x:xs) -> x  )
                                (case v of (x:xs) -> xs )
```

In PMC, we can turn the two function applications into saturated patterns:

$$\begin{aligned} q = \lBrace v \Mapsto\ & \lBrace v \rhd (x : xs) \Mapsto \rceil x \lceil\ \rBrace \rhd x\ \Mapsto \\ & \lBrace v \rhd (x : xs) \Mapsto \rceil xs \lceil \rBrace \rhd xs \Mapsto \mathit{body} \rBrace \end{aligned}$$

4 Standard Reduction Rules

The intuitive explanations in Sect. 2 only provide guidance to one particular way of providing a semantics to PMC expressions and matchings. In this section, we provide a set of rules that implement the usual pattern matching semantics of non-strict languages by allowing corresponding reduction of PMC expressions as they arise from translating functional programs. In particular, we do not include extensionality rules.

Formally, we define two *redex reduction* relations: $\underset{\mathsf{E}}{\longrightarrow} : \mathsf{Expr} \leftrightarrow \mathsf{Expr}$ for expressions, and $\underset{\mathsf{M}}{\longrightarrow} : \mathsf{Match} \leftrightarrow \mathsf{Match}$ for matchings. These are the smallest relations including the rules listed in the sections 4.1 to 4.3. In 4.4 we shortly discuss the characteristics of the resulting rewriting system.

We use the following conventions for metavariables: v is a variable; a, a_1, a_2, …, b, e, e_1, e_2, …, f are expressions; k, n are natural numbers; c, d are constructors; m, m_1, m_2, … are matchings; p, p_1, p_2, …, q are patterns.

4.1 Failure and Returning

Failure is the (left) unit for $\mid$; this enables discarding of failed alternatives and transfer of control to the next alternative:

$$\lightning \mid m \quad \underset{\mathsf{M}}{\longrightarrow} \quad m \qquad (\lightning\mid)$$

A matching abstraction where all alternatives fail represents an ill-defined case — this motivates the introduction of the empty expression into our language:

$$\lBrace \lightning \rBrace \quad \underset{\mathsf{E}}{\longrightarrow} \quad \varnothing \qquad (\lBrace \lightning \rBrace)$$

Empty expressions are produced only by this rule; the rules $(\varnothing @)$ and $(\varnothing \rhd c)$ below only propagate them.

Expression matchings are left-zeros for $\mathbin{\boldsymbol{|}}$:

$$\rceil e \lceil \mathbin{\boldsymbol{|}} m \quad \xrightarrow[\mathsf{M}]{} \quad \rceil e \lceil \qquad\qquad (\rceil\lceil\mathbin{\boldsymbol{|}})$$

Matching abstractions built from expression matchings are equivalent to the contained expression:

$$\{\!|\, \rceil e \lceil \,|\!\} \quad \xrightarrow[\mathsf{E}]{} \quad e \qquad\qquad (\{\!|\rceil\lceil|\!\})$$

4.2 Application and Argument Supply

Application of a matching abstraction reduces to argument supply inside the abstraction:

$$\{\!|\, m \,|\!\}\ a \quad \xrightarrow[\mathsf{E}]{} \quad \{\!|\, a \rhd m \,|\!\} \qquad\qquad (\{\!|\ |\!\}@)$$

Argument supply to an expression matching reduces to function application inside the expression matching:

$$a \rhd \rceil e \lceil \quad \xrightarrow[\mathsf{M}]{} \quad \rceil e\ a \lceil \qquad\qquad (\rhd\rceil\lceil)$$

No matter which of our two interpretations of the empty expression we choose, it absorbs arguments when used as function in an application:

$$\oslash\ e \quad \xrightarrow[\mathsf{E}]{} \quad \oslash \qquad\qquad (\oslash @)$$

Analogously, failure absorbs argument supply:

$$e \rhd \lightning \quad \xrightarrow[\mathsf{M}]{} \quad \lightning \qquad\qquad (\rhd\lightning)$$

Argument supply distributes into alternatives:

$$e \rhd (m_1 \mathbin{\boldsymbol{|}} m_2) \quad \xrightarrow[\mathsf{M}]{} \quad (e \rhd m_1) \mathbin{\boldsymbol{|}} (e \rhd m_2) \qquad\qquad (\rhd\mathbin{\boldsymbol{|}})$$

4.3 Pattern Matching

Everything matches a variable pattern; this matching gives rise to substitution:

$$a \rhd v \Mapsto m \quad \xrightarrow[\mathsf{M}]{} \quad m[v \backslash a] \qquad\qquad (\rhd v)$$

Matching constructors match, and the proviso can always be ensured via α-conversion (for this rule to make sense, linearity of patterns is important):

$$c(e_1, \ldots, e_n) \rhd c(p_1, \ldots, p_n) \Mapsto m \quad \xrightarrow[\mathsf{M}]{} \quad e_1 \rhd p_1 \Mapsto \cdots e_n \rhd p_n \Mapsto m$$

$$\text{if } \mathsf{FV}(c(e_1, \ldots, e_n)) \cap \mathsf{FV}(c(p_1, \ldots, p_n)) = \{\} \qquad\qquad (c \rhd c)$$

Matching of different constructors fails:

$$d(e_1, \ldots, e_k) \rhd c(p_1, \ldots, p_n) \Mapsto m \quad \xrightarrow[\mathsf{M}]{} \quad \lightning \qquad \text{if } c \neq d \text{ or } k \neq n \quad (d \rhd c)$$

For the case where an empty expression is matched against a constructor pattern, we consider two different right-hand sides:

- The calculus $\mathsf{PMC}_{\varnothing}$ interprets the empty expression as equivalent to non-termination, so constructor pattern matchings are strict in the supplied argument:

$$\varnothing \rhd c(p_1, \ldots, p_n) \Mapsto m \quad \xrightarrow[\mathsf{M}]{} \quad \rceil \varnothing \lceil \qquad\qquad (\varnothing \rhd c \rightarrow \varnothing)$$

- The calculus $\mathsf{PMC}_{\lightning}$ interprets the empty expression as propagating the exception of matching failure, and "resurrects" that failure when matching against a constructor:

$$\varnothing \rhd c(p_1, \ldots, p_n) \Mapsto m \quad \xrightarrow[\mathsf{M}]{} \quad \lightning \qquad\qquad (\varnothing \rhd c \rightarrow \lightning)$$

For statements that hold in both $\mathsf{PMC}_{\varnothing}$ and $\mathsf{PMC}_{\lightning}$, we let the rule name $(\varnothing \rhd c)$ stand for the rule $(\varnothing \rhd c \rightarrow \varnothing)$ in $\mathsf{PMC}_{\varnothing}$ and for $(\varnothing \rhd c \rightarrow \lightning)$ in $\mathsf{PMC}_{\lightning}$.

4.4 Rewriting System Aspects

For a reduction rule (R), the one-step *redex* reduction relation defined by that rule is written $\xrightarrow[(R)]{}$; this will either have only expressions, or only matchings in its domain and range. Furthermore, we let $\xrightarrow[(R)]{\circ}$ be the one-step reduction relation closed under expression and matching construction.

Each of the rewriting systems $\mathsf{PMC}_{\varnothing}$ and $\mathsf{PMC}_{\lightning}$ formed by the reduction rules introduced in sections 4.1 to 4.3 consists of nine first-order term rewriting rules, two rule-schemata $(\varnothing \rhd c)$ and $(d \rhd c)$ — parameterised by the constructors and the arities — that involve the binding constructor $\Mapsto$, but not any bound variables, the second-order rule $(\rhd v)$ involving substitution, and the second-order rule schema $(c \rhd c)$ for pattern matching that re-binds variables.

The substituting rule $(\rhd v)$ has almost the same syntactical characteristics as β-reduction, and can be directly reformulated as a CRS rule. (CRS stands for *combinatory reduction system* [11,17].)

The pattern matching rule schema $(c \rhd c)$ involves binders binding multiple variables, but its individual rules still could be reformulated as CRS rules.

The whole system is neither orthogonal nor does it have any other properties like weak orthogonality for which the literature provides confluence proofs; we describe a confluence proof in Sect. 6.

5 Reduction Examples

For the translation of λ-calculus into PMC it is easy to see that every β-reduction can be emulated by a three-step reduction sequence in PMC:

$$\begin{aligned}(\lambda\ v\ .\ e)\ a = \{\!|\, v \Mapsto \rceil e \lceil \,|\!\}\ a &\xrightarrow[(\{\!|\ |\!\}@)]{} \{\!|\, a \rhd v \Mapsto \rceil e \lceil \,|\!\} \xrightarrow[(\rhd v)]{\circ} \{\!|\, \rceil e \lceil [v \backslash a] \,|\!\} \\ = \{\!|\, \rceil e[v \backslash a] \lceil \,|\!\} &\xrightarrow[(\{\!|\rceil\lceil|\!\})]{} e[v \backslash a]\end{aligned}$$

By induction over λ-terms and PMC-reductions starting from translations of λ-terms one can show that such reductions can never lead to PMC expressions containing constructors, failure, $\oslash$, or alternatives, and can only use the four rules $(\{\!|\ |\!\}@)$, $(\rhd v)$, $(\{\!| \rceil\lceil |\!\})$, and $(\rhd\rceil\lceil)$. Of these, the first three make up the translation of β-reduction, and the last can only be applied to "undo" the effect of a "premature" application of $(\{\!|\ |\!\}@)$. In addition, for each PMC reduction sequence starting from the translation of a λ-term t, we can construct a corresponding β-reduction sequence starting from t showing that the only real difference between arbitrary *PMC* reduction sequences of translations of λ-terms and β-reduction sequences is that the *PMC* reduction sequence may contain steps corresponding to "unfinished" β-steps, and "premature" $(\{\!|\ |\!\}@)$ steps.

Therefore, no significant divergence is possible, and confluence of the standard PMC reduction rules, to be shown below, implies that this embedding of the untyped λ-calculus is faithful.

For the PMC translation of the Haskell expression `f bot (3:[])`, the normalising strategy we present below produces the following reduction sequence:

$$
\begin{aligned}
& \{\!|\, ((x : xs) \Mapsto [\,] \Mapsto \rceil 1 \lceil) \,\|\, (ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil) \,|\!\}\ \bot\ (3 : [\,]) \\
& \xrightarrow[(\{\!|\ |\!\}@)]{\circ}\ \{\!|\, \bot \rhd (((x : xs) \Mapsto [\,] \Mapsto \rceil 1 \lceil) \,\|\, (ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil)) \,|\!\}\ (3 : [\,]) \\
& \xrightarrow[(\{\!|\ |\!\}@)]{}\ \{\!|\, (3 : [\,]) \rhd \bot \rhd (((x : xs) \Mapsto [\,] \Mapsto \rceil 1 \lceil) \,\|\, (ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil)) \,|\!\} \\
& \xrightarrow[(\rhd \|)]{\circ}\ \{\!|\, (3 : [\,]) \rhd ((\bot \rhd (x : xs) \Mapsto [\,] \Mapsto \rceil 1 \lceil) \,\|\, (\bot \rhd ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil)) \,|\!\}
\end{aligned}
$$

From here, reduction would loop on the vain attempt to evaluate the first occurrence of $\bot$. If we replace $\bot$ with the empty expression $\oslash$. then we obtain different behaviour according to which interpretation we choose for $\oslash$:

In PMC$_{\oslash}$, the empty expression propagates:

$$
\begin{aligned}
& \{\!|\, (3 : [\,]) \rhd ((\oslash \rhd (x : xs) \Mapsto [\,] \Mapsto \rceil 1 \lceil) \,\|\, (\oslash \rhd ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil)) \,|\!\} \\
& \xrightarrow[(\oslash \rhd c \to \oslash)]{\circ}\ \{\!|\, (3 : [\,]) \rhd (\rceil \oslash \lceil \,\|\, (\oslash \rhd ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil)) \,|\!\} \\
& \xrightarrow[(\rceil\lceil \|)]{\circ}\ \{\!|\, (3 : [\,]) \rhd \rceil \oslash \lceil \,|\!\}\ \xrightarrow[(\rhd \rceil\lceil)]{\circ}\ \{\!|\, \rceil \oslash\ (3 : [\,]) \lceil \,|\!\}\ \xrightarrow[(\{\!| \rceil\lceil |\!\})]{\circ}\ \oslash\ (3 : [\,])\ \xrightarrow[(\oslash @)]{}\ \oslash
\end{aligned}
$$

In PMC$_{\oslash}$, the empty expression $\oslash$ is like a runtime error: it terminates reduction in an "abnormal" way, by propagating through all constructs like an uncaught exception. In PMC$_{\lightning}$, however, this exception can be caught: matching the empty expression against list construction produces a failure, and the other alternative succeeds:

$$
\begin{aligned}
& \{\!|\, (3 : [\,]) \rhd ((\oslash \rhd (x : xs) \Mapsto [\,] \Mapsto \rceil 1 \lceil) \,\|\, (\oslash \rhd ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil)) \,|\!\} \\
& \xrightarrow[(\oslash \rhd c \to \lightning)]{\circ}\ \{\!|\, (3 : [\,]) \rhd (\lightning \,\|\, (\oslash \rhd ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil)) \,|\!\} \\
& \xrightarrow[(\lightning \|)]{\circ}\ \{\!|\, (3 : [\,]) \rhd \oslash \rhd ys \Mapsto (v : vs) \Mapsto \rceil 2 \lceil \,|\!\} \\
& \xrightarrow[(\rhd v)]{\circ}\ \{\!|\, (3 : [\,]) \rhd (v : vs) \Mapsto \rceil 2 \lceil \,|\!\}\ \xrightarrow[(c \rhd c)]{\circ}\ \{\!|\, 3 \rhd v \Mapsto [\,] \rhd vs \Mapsto \rceil 2 \lceil \,|\!\} \\
& \xrightarrow[(\rhd v)]{\circ}\ \{\!|\, [\,] \rhd vs \Mapsto \rceil 2 \lceil \,|\!\}\ \xrightarrow[(\rhd v)]{\circ}\ \{\!|\, \rceil 2 \lceil \,|\!\}\ \xrightarrow[(\{\!| \rceil\lceil |\!\})]{\circ}\ 2
\end{aligned}
$$

It is not hard to see that reduction in PMC cannot arrive at the result 2 in the example with $\perp$, even if the second alternative can, for example after the third step of the original sequence, be reduced to $\rceil 2 \lceil$: If a pattern matching $p \mapsto\!\!\!> m$ has been supplied with an argument a, then in the resulting $a \rhd p \mapsto\!\!\!> m$, the matching m can be considered as *guarded* by the *pattern guard* "$a \rhd p$" (in abuse of syntax). An alternative can only be committed to if *all* its pattern guards *succeed*; discarding — ultimately via $(\lightning\,\|)$ — an alternative with only non-variable patterns and arguments for all patterns only works if its *first* pattern guard can be determined as mismatching. In $\mathsf{PMC}_{\oslash}$, this can only be via $(d \rhd c)$; while in $\mathsf{PMC}_{\lightning}$, it could also be via $(\oslash \rhd c \rightarrow \lightning)$.

6 Confluence and Formalisation

Just among the first-order rules, four critical pairs arise: where the matching delimiters $\|$ and $\{\!|\ |\!\}$ on the one hand are eliminated by failure $\lightning$ or expression matchings $\rceil e \lceil$, and on the other hand are traversed by argument supply. None of these critical pairs is resolved by single steps of simple parallel reduction. It is easy to see that a shortcut rule, such as $\{\!|\, a \rhd \rceil e \lceil \,|\!\} \rightarrow e\ a$, immediately gives rise to a new critical pair that would need to be resolved by a longer shortcut rule, in this case $\{\!|\, b \rhd a \rhd \rceil e \lceil \,|\!\} \rightarrow e\ a\ b$.

A more systematic approach than introducing an infinite number of such shortcut rules is to adopt Aczel's approach [1] to parallel reduction that also reduces redexes created "upwards" by parallel reduction steps. Confluence of PMC reduction can then be shown by establishing the diamond property for the parallel reduction relations.

Using a formalisation in Isabelle-2003/Isar/HOL [15], I have performed a machine-checked proof of this confluence result.[1] Since both de Bruijn indexing and translation into higher-order abstract syntax would have required considerable technical effort and would have resulted in proving properties less obviously related to the pattern matching calculus as presented here, I have chosen as basis for the formalisation the Isabelle-1999/HOL theory set used by Vestergaard and Brotherston in their confluence proof for λ-calculus [22]. This formalisation is based on *first-order abstract syntax* and makes all the issues involved in variable renaming explicit. Therefore, the formalisation includes the rules as given in Sect. 4 with the same side-conditions; only the formalisation of the substituting variable match rule $(\rhd v)$ has an additional side-condition ensuring permissible substitution in analogy with the treatment of the β-rule in [22].

Vestergaard and Brotherston employed parallel reduction in the style of the Tait/Martin-Löf proof method, and used Takahashi's proof of the diamond property via complete developments. For PMC, we had to replace this by the Aczel-style extended parallel reduction relations, and a direct case analysis for the diamond property of these relations.

Due to the fact that we are employing two mutually recursive syntactic categories (in Isabelle, the argument list of constructors actually counts as a third

[1] The proof is available at URL: http://www.cas.mcmaster.ca/~kahl/PMC/

category in that mutual recursion), and due to the number of constructors of the pattern matching calculus (twelve including the list constructors, as opposed to three in the λ-calculus), the number of constituent positions in these constructors (twelve — including those of list construction — versus three), and the number of reduction rules (thirteen versus one), there is considerable combinatorial blow-up in the length of both the formalisation and the necessary proofs.

7 Normalisation

Since PMC is intended to serve as operational semantics for *lazy* functional programming with pattern matching, we give a reduction strategy that reduces expressions and matchings to *strong head normal form* (SHNF), see, e.g., [19, Sect. 4.3] for an accessible definition. With the set of rules defined in Sect. 4, the following facts about SHNFs are easily seen:

- Variables, constructor applications, the empty expression $\oslash$, failure $\lightning$, expression matchings $\rceil e \lceil$, and pattern matchings $p \Mapsto m$ are already in SHNF.
- All rules that have an application $f\ a$ at their top level have a metavariable for a, and none of these rules has a metavariable for f, so $f\ a$ is in SHNF if f is in SHNF and $f\ a$ is not a redex.
- A matching abstraction $\{\!|\, m \,|\!\}$ is in SHNF if m is in SHNF, unless $\{\!|\, m \,|\!\}$ is a redex for one of the rules $(\{\!|\lightning|\!\})$ or $(\{\!|\rceil\lceil|\!\})$.
- Since all alternative rules have metavariables for m_2, an alternative $m_1 \,\|\, m_2$ is in SHNF if m_1 is in SHNF, unless $m_1 \,\|\, m_2$ itself is a redex.
- No rule for argument supply $a \rhd m$ has a metavariable for m, and all rules for argument supply $a \rhd m$ that have non-metavariable a have m of shape $c(p_1, \ldots, p_n) \Mapsto m'$. Therefore, if $a \rhd m$ is not a redex, it is in SHNF if m is in SHNF and, whenever m is of the shape $c(p_1, \ldots, p_n) \Mapsto m'$, a is in SHNF, too.

Due to the homogenous nature of its rule set, PMC therefore has a deterministic strategy for reduction of applications, matching abstractions, alternatives, and argument supply to SHNF:

- If an application $f\ a$ is a redex, reduce it; otherwise if f is not in SHNF, proceed into f.
- For a matching abstraction $\{\!|\, m \,|\!\}$, if m is not in SHNF, proceed into m, otherwise reduce $\{\!|\, m \,|\!\}$ if it is a redex.
- For an alternative $m_1 \,\|\, m_2$, if m_1 is not in SHNF, proceed into m_1, otherwise reduce $m_1 \,\|\, m_2$ if it is a redex.
- If an argument supply $a \rhd m$ is a redex, reduce it (this is essential for the case where m is of shape $m_1 \,\|\, m_2$, which is not necessarily in SHNF, and $(\rhd\|)$ has to be applied). Otherwise, if m is not in SHNF, proceed into m. If m is of the shape $c(p_1, \ldots, p_n) \Mapsto m'$, and a is not in SHNF, proceed into a.

Matching abstractions and alternatives are redexes only if the selected constituent is in SHNF — this simplified the formulation of the strategy for these cases.

This strategy induces a deterministic normalising strategy in the usual way.

8 Related Work

In Peyton Jones' book [18], the chapter 4 by Peyton Jones and Wadler introduces a "new built-in value FAIL, which is returned when a pattern-match fails" (p. 61). In addition, their "enriched λ-calculus" also contains an alternative constructor, for which FAIL is the identity, thus corresponding to our failure $\lightning$. However, FAIL only occurs in contexts where there is a right-most ERROR alternative (errors ERROR are distinguished from non-termination $\bot$), so there is no opportunity to discover that, in our terms, $\{\!|\,\mathsf{FAIL}\,|\!\} = \mathsf{ERROR}$. Also, errors always propagate; since ERROR corresponds to our empty expression $\oslash$, their error behaviour corresponds to our rule $(\oslash \rhd c \to \oslash)$. Wadler's chapter 5, one of the standard references for compilation of pattern matching, contains a section about optimisation of expressions containing alternative and FAIL, arguing along lines that would be simplified by our separation into matchings and expressions.

Similarly, Tullsen includes a primitive "failure combinator" that never succeeds among his "First Class Patterns" combinators extending Haskell [21]. He uses a purely semantic approach, with functions into a Maybe type or, more generally, into a MonadPlus as "patterns". In this way, he effectively embeds our two-sorted calculus into the single sort of Haskell expressions, with a fixed interpretation. However, since expressions are non-patterns, Tullsen's approach treats them as standard Haskell expressions and, therefore, does not have the option to consider "resurrecting failures" as in our rule $(\oslash \rhd c \to \lightning)$. Harrison *et al.* follow a similar approach for modelling Haskell's evaluation-on-demand in detail [9]; they consider "case branches `p -> e`" as separate syntactical units — such a case branch is a PMC matching $p \Mapsto e$ — and interpret them as functions into a Maybe type; the interpretation of case expressions translates failure into bottom, like in Tullsen's approach.

Erwig and Peyton Jones, together with their proposal of pattern guards, in [6] also proposed to use a `Fail` exception for allowing pattern matching failure as result of conventional Haskell expressions, and explicitly mention the possibility to catch this exception in the same *or in another* case expression. This is the only place in the literature where we encountered an approach somewhat corresponding to our rule $(\oslash \rhd c \to \lightning)$; we feel that our formalisation in the shape of $\mathsf{PMC}_{\lightning}$ can contribute significantly to the further exploration of this option.

Van Oostrom defined an untyped λ-calculus with patterns in [16], abstracting over (restricted) λ-terms. This calculus does not include mismatch rules and therefore requires complicated encoding of typical multi-pattern definitions.

Typed pattern calculi with less relation to lazy functional programming are investigated by Delia Kesner and others in, e.g., [3,8]. Patterns in these calculi can be understood as restricted to those cases that are the result of certain kinds of pattern compilation, and therefore need not include any concern for incomplete alternatives or failure propagation.

As explained in the introduction, pattern matching can be seen as an internalisation of term rewriting; PMC represents an internalisation of the functional rewriting strategy described for example in [19]. Internalisation of general, nondeterministic term rewriting has been studied by H. Cirstea, C. Kirchner and others as the rewriting calculus, also called ρ-**calculus** [4,5], and, most recently, in typed variants as "pure pattern type systems" [2]. The ρ-**calculus** is parameterised by a theory modulo which matching is performed; this can be used to deal with *views* [23]. Since the ρ-**calculus** allows arbitrary expressions as patterns, confluence holds only under restriction to *call-by-value* strategies. The ρ-**calculus** has unordered sets of alternatives that can also be empty; in [7] a distinction between matching failure and the empty alternative has been added for improving support for formulating rewriting strategies as ρ-**calculus** terms. Since matching failure in the ρ-**calculus** is an expression constant, it can occur as function or constructor argument, and proper propagation of failure must be enforced by call-by-value evaluation.

Maranget [13] describes "automata with failures" as used by several compilers. Translated into our formalism, this introduces a `default` matching that never matches, but transfers control to the closest enclosing alternative containing a wildcard (variable) pattern. This can be seen as closely related with our rule ($\oslash \rhd c \to \lightning$); Maranget-1994 used this feature as a way of allowing backtracking during pattern matching. In [12], this is extended to labelled exceptions, and can also be understood as a way of implementing sharing between alternatives.

9 Conclusion and Outlook

The pattern matching calculus $\mathsf{PMC}_{\oslash}$ turns out to be a simple and elegant formalisation of the operational pattern matching semantics of current non-strict functional programming languages. $\mathsf{PMC}_{\oslash}$ is a confluent reduction system with a simple deterministic normalising strategy, and therefore does not require the complex priorisation mechanisms of the functional rewriting strategy or other pattern matching definitions.

In addition, we have shown how changing a single rule produces the new calculus $\mathsf{PMC}_{\lightning}$, which results in "more successful" evaluation, but is still confluent and normalising, Therefore, $\mathsf{PMC}_{\lightning}$ is a promising foundation for further exploration of the "failure as exception" approach proposed by Erwig and Peyton Jones, for turning it into a basis for programming language implementations, and for relating it with Maranget's approach.

The technical report [10] shows, besides other details, also a simple polymorphic typing discipline, and the inclusion of an explicit fixedpoint combinator, which preserves confluence and normalisation.

The next step will be an investigation of theory and denotational semantics of both calculi: For $\mathsf{PMC}_{\oslash}$, the most natural approach will be essentially the Maybe semantics of pattern matching as proposed in [21,9]. For $\mathsf{PMC}_{\lightning}$, the semantic domain for expressions needs to include an alternative for failure, too, to represent the semantics of empty expressions $\oslash$.

We also envisage that pattern matching calculi would be a useful basis for an interactive program transformation and reasoning systems for Haskell, similar to what SPARKLE [14] is for CLEAN.

References

1. P. ACZEL. *A general Church-Rosser theorem.* unpublished note, see [20], 1978.
2. G. BARTHE et al.. *Pure Patterns Type Systems.* In: POPL 2003. ACM, 2003.
3. V. BREAZU-TANNEN, D. KESNER, L. PUEL. *A Typed Pattern Calculus.* In: Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science, pp. 262–274, Montreal, Canada, 1993. IEEE Computer Society Press.
4. H. CIRSTEA, C. KIRCHNER. *Combining Higher-Order and First-Order Computation Using ρ-calculus: Towards a semantics of* ELAN. In D. GABBAY, M. DE RIJKE, eds., Frontiers of Combining Systems 2, Oct. 1998, pp. 95–120. Wiley, 1999.
5. H. CIRSTEA, C. KIRCHNER. *The rewriting calculus — Part I* and *II*. Logic Journal of the Interest Group in Pure and Applied Logics **9** 427–498, 2001.
6. M. ERWIG, S. P. JONES. *Pattern Guards and Transformational Patterns.* In: Haskell Workshop 2000, ENTCS **41 no. 1**, pp. 12.1–12.27, 2001.
7. G. FAURE, C. KIRCHNER. *Exceptions in the rewriting calculus.* In S. TISON, ed., RTA 2002, LNCS **2378**, pp. 66–82. Springer, 2002.
8. J. FOREST, D. KESNER. *Expression Reduction Systems with Patterns.* In R. NIEUWENHUIS, ed., RTA 2003, LNCS **2706**, pp. 107–122. Springer, 2003.
9. W. L. HARRISON, T. SHEARD, J. HOOK. *Fine Control of Demand in Haskell.* In: Mathematics of Program Construction, MPC 2002, LNCS. Springer, 2002.
10. W. KAHL. *Basic Pattern Matching Calculi: Syntax, Reduction, Confluence, and Normalisation.* SQRL Rep. 16, Software Quality Res. Lab., McMaster Univ., 2003.
11. J. W. KLOP. *Combinatory Reduction Systems.* Mathematical Centre Tracts 127, Centre for Mathematics and Computer Science, Amsterdam, 1980. PhD thesis.
12. F. LE FESSANT, L. MARANGET. *Optimizing Pattern Matching.* In X. LEROY, ed., ICFP 2001, pp. 26–37. ACM, 2001.
13. L. MARANGET. *Two Techniques for Compiling Lazy Pattern Matching.* Technical Report RR 2385, INRIA, 1994.
14. M. DE MOL, M. VAN EEKELEN, R. PLASMEIJER. *Theorem Proving for Functional Programmers —* SPARKLE*: A Functional Theorem Prover.* In T. ARTS, M. MOHNEN, eds., IFL 2001, LNCS **2312**, pp. 55–71. Springer, 2001.
15. T. NIPKOW, L. C. PAULSON, M. WENZEL. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, LNCS **2283**. Springer, 2002.
16. V. VAN OOSTROM. *Lambda Calculus with Patterns.* Technical Report IR 228, Vrije Universiteit, Amsterdam, 1990.
17. V. VAN OOSTROM, F. VAN RAAMSDONK. *Comparing combinatory reduction systems and higher-order rewrite systems.* In J. HEERING, K. MEINKE, B. MÖLLER, T. NIPKOW, eds., HOA '93, LNCS **816**, pp. 276–304. Springer, 1993.
18. S. L. PEYTON JONES. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.
19. R. PLASMEIJER, M. VAN EEKELEN. *Functional Programming and Parallel Graph Rewriting.* International Computer Science Series. Addison-Wesley, 1993.
20. F. VAN RAAMSDONK. *Higher-order Rewriting.* In TERESE, ed., Term Rewriting Systems, Chapt. 11, pp. 588–667. Cambridge Univ. Press, 2003.
21. M. TULLSEN. *First Class Patterns.* In E. PONTELLI, V. SANTOS COSTA, eds., PADL 2000, LNCS **1753**, pp. 1–15. Springer, 2000.
22. R. VESTERGAARD, J. BROTHERSTON. *A Formalised First-Order Confluence Proof for the λ-Calculus using One-Sorted Variable Names.* In A. MIDDELDORP , ed., RTA 2001, LNCS **2051,** pp. 306–321. Springer, 2001.
23. P. WADLER. *Views: A Way for Pattern Matching to Cohabit with Data Abstraction.* In S. MUNCHNIK, ed., POPL 1987, pp. 307–313. ACM, 1987.

Derivation of Deterministic Inverse Programs Based on LR Parsing

Robert Glück[1] and Masahiko Kawabe[2]

[1] PRESTO, JST & University of Copenhagen, DIKU
DK-2100 Copenhagen, Denmar
glueck@acm.org
[2] Waseda University, Graduate School of Science and Engineering
Tokyo 169-8555, Japan
kawabe@futamura.info.waseda.ac.jp

Abstract. We present a method for automatic program inversion of functional programs based on methods of LR parsing. We formalize the transformation and illustrate it with the inversion of a program for run-length encoding. We solve one of the main problems of automatic program inversion—the elimination of nondeterminism—by viewing an inverse program as a context-free grammar and applying to it methods of LR parsing to turn it into a recursive, deterministic inverse program. This improves the efficiency of the inverse programs and greatly expands the application range of our earlier method for program inversion.

1 Introduction

The contribution of this paper is an automatic method for program inversion based on methods of LR parsing. This transformation improves the efficiency of the inverse programs and extends the application range of our earlier method by allowing the inversion of programs based on a global transformation of a nondeterministic inverse program. We make use of a self-inverse primitive function for the duplication of values and testing of equality, which we introduced in recent work [8], and a symmetric program representation to simplify inversion. To eliminate nondeterminism in a global view, we apply methods for LR parsing by viewing an inverse program as a context-free grammar and generating a deterministic inverse program if that grammar has certain properties (*e.g.,* without parsing conflicts). This greatly expands the application range of our recent method for program inversion.

The idea of program inversion can be traced back to reference [6]. Recent work [14] has focused on the converse of a function theorem [4], inverse computation of functional programs [2], and the transformation of interpreters into inverse interpreters by partial evaluation [9]. Logic programming is suited to find multiple solutions and can be used for *inverse interpretation,* while in this paper we are interested in *program inversion* (for a detailed description of these notions, see reference [1]). We consider one-to-one functional programs and not relations with multiple solutions. An example is the generation of a program for

© Springer-Verlag Berlin Heidelberg 2004

$$
\begin{array}{lll}
q ::= d_1 \ldots d_n & & \text{(program)} \\
d ::= f(x_1,\ldots,x_n) = t & & \text{(definition)} \\
t ::= (l_1,\ldots,l_m) & & \text{(return)} \\
\quad \mid \ \textbf{case}\ l\ \textbf{of}\ \{p_i \rightarrow t_i\}_{i=1}^{m} & & \text{(case-expression)} \\
\quad \mid \ \textbf{let}\ (y_1,\ldots,y_m)=f(l_1,\ldots,l_n)\ \textbf{in}\ t & & \text{(let-expression)} \\
l ::= x & & \text{(variable)} \\
\quad \mid \ c(l_1,\ldots,l_n) & & \text{(constructor)} \\
\quad \mid \ \lfloor l \rfloor & & \text{(duplication/equality)} \\
p ::= c(x_1,\ldots,x_n) & & \text{(pattern)}
\end{array}
$$

Fig. 1. Abstract syntax of the source language

decoding data given a program for encoding data, and vice versa. In general, the goal of a program inverter is to find an *inverse program* $q^{-1} : B \rightarrow A$ of a program $q : A \rightarrow B$ such that for all values $x \in A$ and $y \in B$ we have

$$q(x) = y \iff q^{-1}(y) = x \ .$$

This tells us that, if a program q terminates on input x and returns output y, then the inverse program q^{-1} terminates on y and returns x, and vice versa. This implies that both programs are *injective*; they need not be surjective or total. Here, equality means strong equivalence: either both sides of an equation are defined and equal, or both sides are undefined. In practice, even when it is certain that an efficient inverse program q^{-1} exists, the automatic generation of such a program from q may be difficult or impossible.[3]

The first method developed for automatic program inversion of first-order functional programs appears to be the program inverter by Korf and Eppstein [13,7] (we call it KEinv for short). It is one of only two general-purpose automatic program inverters that have been built (the other one is InvX [12]). Manual methods [6,11,4,14] and semi-automatic methods [5] exist, but require ingenuity and human insight. Our goal is to achieve further automation of general-purpose program inversion.

This paper is organized as follows. First, we define the source language (Sect. 2). Then we discuss our solution of the main challenges of program inversion (Sect. 3) and present our inversion method (Sect. 4). We discuss related work (Sect. 5), and then give a conclusion (Sect. 6). We assume that the reader is familiar with the principles of LR parsing, *e.g.,* as presented in [3].

2 Source Language

We are concerned with a first-order functional language. A program q is a sequence of function definitions d where the body of each definition is a term t

[3] There exists a program inverter that returns, for every q, a *trivial inverse* q^{-1} [1].

```
$pack(s) =$ case $s$ of $[\,] \to ([\,])$
                        $c{:}r \to$ let $(p,l){=}len(c,\mathrm{O},r)$ in $(p{:}pack(l))$
$len(c,n,s) =$ case $s$ of $[\,] \to (\langle c,n\rangle,[\,])$
                          $d{:}r \to$ case $\lfloor\langle c,d\rangle\rfloor$ of
                                   $\langle e\rangle \to$ let $(p,t){=}len(e,\mathrm{S}(n),r)$ in $(p,t)$
                                   $\langle e,f\rangle \to (\langle e,n\rangle,f{:}r)$
```

Fig. 2. Program *pack*

constructed from variables, constructors, function calls, case- and let-expressions (Fig. 1 where $m > 0$, $n \geq 0$). For reasons of symmetry, functions may return multiple output values which is denoted by syntax $(l_1, \ldots, l_m)$. Arity and coarity of functions and constructors are fixed. The language has a call-by-value semantics. A value v in the language is a constructor c with arguments $v_1, ..., v_n$:

$$v ::= c(v_1 \ldots v_n) .$$

An example is the program for run-length encoding (Fig. 2): function *pack* encodes a list of symbols as a list of symbol-number pairs, where the number specifies how many copies of a symbol have to be generated upon decoding. For instance, $pack([\mathrm{AABCCC}]) = [\langle \mathrm{A},2\rangle\langle \mathrm{B},1\rangle\langle \mathrm{C},3\rangle]$.[4] Function *pack* maximizes the counter: we never have an encoding like $\langle \mathrm{C},2\rangle\langle \mathrm{C},1\rangle$, but rather, always $\langle \mathrm{C},3\rangle$. This implies that the symbols in two adjacent symbol-number pairs are never equal. Fig. 3 shows the inverse function $pack^{-1}$. In the implementation, we use unary numbers where O denotes One and S the Successor. The primitive function $\lfloor\cdot\rfloor$ checks the equality of two values: $\lfloor\langle v,v'\rangle\rfloor = \langle v\rangle$ if $v = v'$. In the absence of equality, the values are returned unchanged: $\lfloor\langle v,v'\rangle\rfloor = \langle v,v'\rangle$ if $v \neq v'$. This will be defined below.

We consider only *well-formed* programs. As usual, we require that no two patterns p_i and p_j in a case-expression contain the same constructor and that all patterns are linear (no variable occurs more than once in a pattern). We also require that each variable be defined before its use and, for simplicity, that no defined variable be redefined by a case- or let-expression.

Duplication and Equality One of our key observations [8] was that duplication and equality testing are two sides of the same coin in program inversion: the duplication of a value in a program becomes an equality test in the inverse program, and vice versa. To simplify inversion, we introduce a primitive function $\lfloor\cdot\rfloor$ defined as follows:

[4] We use the shorthand notation $x{:}xs$ and [ ] for the constructors $\mathrm{Cons}(x,xs)$ and Nil. For $x_1{:}x_2{:}\ldots{:}x_n{:}[\,]$ we write $[x_1x_2\ldots x_n]$, or sometimes $x_1x_2\ldots x_n$. A tuple $\langle x_1,\ldots,x_n\rangle$ is a shorthand notation for an n-ary constructor $\mathrm{C}_n(x_1,\ldots,x_n)$.

```
$f_{[0]}(s) =$ let $(r)=f_{[1]}(s)$ in $(r)$
$f_{[1]}(s) =$ case $s$ of $[\,] \rightarrow ([\,])$
                           $p{:}r \rightarrow$ let $(l)=f_{[1]}(r)$ in
                                   let $(c,n,a)=f_{[26]}(l,p)$ in
                                   let $(b)=f_{[11,10,9]}(n,c,a)$ in $(b)$
$f_{[11,10,9]}(n,c,s) =$ case $n$ of $\mathrm{O} \rightarrow (c{:}s)$
                                   $\mathrm{S}(m) \rightarrow$ case $\lfloor\langle c\rangle\rfloor$ of
                                                  $\langle d,e\rangle \rightarrow$ let $(a)=f_{[11,10,9]}(m,d,e{:}s)$ in $(a)$
$f_{[26]}(s,p) =$ case $s$ of $[\,] \rightarrow$ case $y$ of $\langle c,n\rangle \rightarrow (c,n,[\,])$
                              $c{:}r \rightarrow$ case $p$ of
                                         $\langle d,n\rangle \rightarrow$ case $\lfloor\langle d,c\rangle\rfloor$ of $\langle e,f\rangle \rightarrow (e,n,f{:}r)$
```

Fig. 3. Program *pack*$^{-1}$

$$\lfloor\langle v\rangle\rfloor \stackrel{\text{def}}{=} \langle v,v\rangle \qquad \text{(duplication)}$$

$$\lfloor\langle v,v'\rangle\rfloor \stackrel{\text{def}}{=} \begin{cases} \langle v\rangle & \text{if } v = v' \\ \langle v,v'\rangle & \text{if } v \neq v' \end{cases} \qquad \text{(equality test)}$$

There are mainly two ways of using this function: duplication and equality testing. In the former case, given a single value, a pair with identical values is returned; in the latter case, given a pair of identical values, a single value is returned; otherwise the pair is returned unchanged. The advantage of this unusual function definition is that it makes it easier to deal with duplication and equality testing in program inversion. The function has a useful property, namely that it is *self-inverse*, which means it is its own inverse: $\lfloor\cdot\rfloor^{-1} = \lfloor\cdot\rfloor$.

For example, in function *len* (Fig. 2) the equality of two adjacent symbols, c and d, is tested in the innermost case-expression. The assertion that those symbols are not equal is checked in forward and backward computation.

3 Challenges to Program Inversion

The most challenging point in program inversion is the inversion of conditionals (here, case-expressions). To calculate the input from a given output, we must know which of the m branches in a case-expression the source program took to produce that output, since *only one* of the branches was executed in the forward calculation (our language is deterministic). To make this choice in an inverse program, we must know m postconditions, R_i, one for each branch, such that for each pair of postconditions, we have: $R_i \wedge R_j = \mathit{false}$ $(1 \leq i < j \leq m)$. This divides the set of output values into m disjoint sets, and we can choose the correct branch by testing the given output value using the postconditions.

Postconditions that are suitable for program inversion can be derived by hand (*e.g.,* [6,11]). In automatic program inversion they must be inferred from a source program. The program inverter KEinv [7] uses a heuristic method, and the language in which its postconditions are expressed consists of the primitive predicates available for the source language's value domain consisting of lists and integers. In general, there is no automatic method that would always find mutually exclusive postconditions, even if they exist.

A nondeterministic choice is an unspecified choice from a number of alternatives. Not every choice will lead to a successful computation. If there is only one choice to choose from, then the computation is deterministic.

In previous work [8, Sect.4], we gave a local criteria for checking whether the choice in an inverse program will be deterministic. We viewed the body of a function as a tree with the head of the function definition as the root, and required that the expressions in the leaves of the tree return disjoint sets of values. For example, the expressions in the two leaves of function *pack* are $([\,])$ and $(\_:\_)$. Clearly, both represent disjoint sets of values. This works surprisingly well for a class of programs, but is too restricted for other cases. For example, consider function *len* (Fig. 2). It has three leaf expressions each of which returns two values (a symbol-number pair and the remaining list of symbols):

$$1.\ (\langle c,n\rangle,[\,]) \qquad 2.\ (p,t) \qquad 3.\ (\langle e,n\rangle,f{:}r)$$

Our local criterion can distinguish the set of values returned by leaf (1) and (3), but it is not sufficient for leaf (2). The set of values represented by (2) is not disjoint from (1) and (3). In fact, it is the union of (1) and (3).

This paper deals with this limitation of our previous method by applying methods from LR parsing to determine whether the choice is deterministic and to generate a recursive inverse program. These techniques replace our local criteria. As we shall see, a deterministic inverse program $pack^{-1}$ can be derived from *pack* by the method introduced in this paper.

Dead Variables Another problematic point in program inversion is when input values are discarded. Consider the selection function *first* defined by

$$first(x) = \textbf{case}\ x\ \textbf{of}\ h{:}t \rightarrow h$$

When we invert such a program, we have to guess 'lost values' (here, a value for t). In general, there are infinitely many possible guesses. We adopted a straightforward solution which we call the "preservation of values" requirement. For a program *well-formed for inversion,* we also require that each defined variable be used exactly once. Thus, a variable's value is always part of the output, and the only way to "diminish the amount of output" is to reduce pairs of values into singletons by $\lfloor\cdot\rfloor$. For example, we write **case** $\lfloor\langle x,y\rangle\rfloor$ **of** $\langle z\rangle \rightarrow \ldots$. This expression ensures that no information is lost because all values need to be identical.

$q ::= d_1 \ldots d_n$	(program)
$d ::= f \to t_1 \ldots t_n$	(definition)
$t ::= \mathbf{in}(x_1, \ldots, x_n)$	(input)
$\mid\ \mathbf{out}(y_1, \ldots, y_n)$	(output)
$\mid\ c(x_1, \ldots, x_n){=}y$	(constructor)
$\mid\ x{=}c(y_1, \ldots, y_n)$	(pattern matching)
$\mid\ \lfloor x \rfloor{=}y$	(duplication/equality)
$\mid\ f(x_1, \ldots, x_n){=}(y_1, \ldots, y_m)$	(function call)

Fig. 4. Abstract syntax of the symmetric language

4 A Method for Program Inversion

We now present a method for the automatic inversion of programs that are well-formed for inversion. Our method uses symmetric representation of a program as internal representation for inverting primitive operators and a grammar representation for eliminating nondeterminism. It consists of translations $SYM[\![\cdot]\!]$, $GRAM[\![\cdot]\!]$ and $FCT[\![\cdot]\!]$ that translate from the source language to the internal representation and vice versa. Local inversion $INV[\![\cdot]\!]$ is then performed by backward reading of the symmetric representation. Finally, $DET[\![\cdot]\!]$ attempts to eliminate nondeterminism by LR parsing techniques. To simplify inversion of a program, inversion is carried out on a symmetric representation, rather than on the source program. We now give its definition and explain each of its components in the remainder of this section.

Definition 1 (program inverter). *Let q be a program well-formed for inversion. Then* program inverter $[\![\cdot]\!]^{-1}$ *is defined by*

$$[\![q]\!]^{-1} \overset{\text{def}}{=} FCT[\![\, DET[\![\, GRAM[\![\, INV[\![\, SYM[\![\, q \,]\!] \,]\!] \,]\!] \,]\!] \,]\!]$$

4.1 Translation to the Symmetric Language

The translation of a function to a symmetric representation makes it easier to invert the function. During the translation, each construct is decomposed into a sequence of atomic operations. The syntax of the symmetric language is shown in Fig. 4. An atomic operation t is either a construct that marks several variables as input $\mathbf{in}(x_1, \ldots, x_n)$ or as output $\mathbf{out}(y_1, \ldots, y_n)$, an equality representing a constructor application $c(x_1, \ldots, x_n){=}y$, a pattern matching $x{=}c(y_1, \ldots, y_n)$, an operator application $\lfloor x \rfloor{=}y$, or a function call $f(x_1, \ldots, x_n){=}(y_1, \ldots, y_m)$. As a convention, the left-hand side of an equation is defined only in terms of input variables (here x) and the right-hand side is defined only in terms of output variables (here y). The intended forward reading of a sequence of equalities is from left to right; the backward reading will be from right to left. A function is

$$
\begin{aligned}
&\mathit{Sym}\llbracket\ f(xs) = t\ \rrbracket && = \mathit{symt}\llbracket\ t,\ \mathbf{in}(xs),\ f\ \rrbracket \\
&\mathit{symt}\llbracket\ (l_1,\dots,l_n),\ ts,\ f\ \rrbracket && = \{f \rightarrow \mathit{syml}\llbracket\ l_n,\ \hat{x}_n, \dots \mathit{syml}\llbracket\ l_1,\ \hat{x}_1,\ ts\ \rrbracket \dots\ \rrbracket\ \mathbf{out}(\hat{x}_1,\dots,\hat{x}_n)\} \\
&\mathit{symt}\llbracket\ \mathbf{case}\ l\ \mathbf{of}\ \{p_i \rightarrow t_i\}_{i=1}^{m},\ ts,\ f\ \rrbracket && = \bigcup_{i=1}^{m} \mathit{symt}\llbracket\ t_i,\ \mathit{syml}\llbracket\ l,\ \hat{x},\ ts\ \rrbracket\ \hat{x}{=}p_i,\ f\ \rrbracket \\
&\mathit{symt}\llbracket\ \mathbf{let}\ (ys){=}f(xs)\ \mathbf{in}\ t,\ ts,\ f\ \rrbracket && = \mathit{symt}\llbracket\ t,\ ts\ f(xs){=}(ys),\ f\ \rrbracket \\
\\
&\mathit{syml}\llbracket\ x,\ y,\ ts\ \rrbracket && = ts\{x \mapsto y\} \\
&\mathit{syml}\llbracket\ c(l_1,\dots,l_n),\ y,\ ts\ \rrbracket && = \mathit{syml}\llbracket\ l_n,\ \hat{x}_n, \dots \mathit{syml}\llbracket\ l_1,\ \hat{x}_1,\ ts\ \rrbracket \dots\ \rrbracket\ c(\hat{x}_1,\dots,\hat{x}_n){=}y \\
&\mathit{syml}\llbracket\ \lfloor l \rfloor,\ y,\ ts\ \rrbracket && = \mathit{syml}\llbracket\ l,\ \hat{x},\ ts\ \rrbracket\ \lfloor \hat{x} \rfloor{=}y
\end{aligned}
$$

Fig. 5. Translation from the functional language to the symmetric language

represented by one or more linear sequences of atomic operations. If a match operation in a sequence fails, the next sequence is tried. For instance, examine the result of translating function *pack* into the symmetric representation in Fig. 12. The translation is defined in Fig. 5. Function $\mathit{symt}\llbracket\ \cdot\ \rrbracket$ performs a recursive decent over t expressions until it reaches a return expression; function $\mathit{syml}\llbracket\ \cdot\ \rrbracket$ translates l expressions. The translation fixes an evaluation order when translating expressions with multiple arguments (other orders are possible). Notation $\hat{x}$ denotes a fresh variable; they act as liaison variables.

Definition 2 (frontend). *Let d be a definition in a program well-formed for inversion. Then, the translation from the functional language to the symmetric language is defined by*

$$\mathit{SYM}\llbracket\ q\ \rrbracket \overset{\text{def}}{=} \bigcup_{d \in q} \mathit{Sym}\llbracket\ d\ \rrbracket$$

4.2 Local Inversion of a Symmetric Program

Operations in the symmetric representation are easily inverted by reading the intended meaning backwards. Every construct in the symmetric language has an inverse construct. Each function definition is inverted separately. The idea of inverting programs by 'backward reading' is not new and can be found in [6,11]. The rules for our symmetric representation are shown in Fig. 6. Global inversion of a program at this stage is based on the local invertibility of atomic operations.

The inverse of $\mathbf{in}(x_1,\dots,x_n)$ is $\mathbf{out}(x_1,\dots,x_n)$ and vice versa; the inverse of constructor application $c(x_1,\dots,x_n){=}y$ is pattern matching $y{=}c(x_1,\dots,x_n)$ and vice versa; the inverse of function call $f(x_1,\dots,x_n){=}(y_1,\dots,y_m)$ is $f^{-1}(y_1,\dots,y_m){=}(x_1,\dots,x_n)$. As explained in Sect. 2, primitive function $\lfloor\cdot\rfloor$ is its own inverse. Thus, the inverse of $\lfloor x \rfloor{=}y$ is $\lfloor y \rfloor{=}x$. Observe that the inversion performs no unfold/fold on functions. It terminates on all programs.

$$\begin{array}{ll}
Inv[\![f \rightarrow t_1 \ldots t_n]\!] & = f^{-1} \rightarrow inv[\![t_n]\!] \ldots inv[\![t_1]\!] \\
inv[\![\mathbf{in}(x_1, \ldots, x_n)]\!] & = \mathbf{out}(x_1, \ldots, x_n) \\
inv[\![\mathbf{out}(x_1, \ldots, x_n)]\!] & = \mathbf{in}(x_1, \ldots, x_n) \\
inv[\![c(x_1, \ldots, x_n)=y]\!] & = y=c(x_1, \ldots, x_n) \\
inv[\![x=c(y_1, \ldots, y_n)]\!] & = c(y_1, \ldots, y_n)=x \\
inv[\![\lfloor x \rfloor=y]\!] & = \lfloor y \rfloor=x \\
inv[\![f(x_1, \ldots, x_n)=(y_1, \ldots, y_m)]\!] & = f^{-1}(y_1, \ldots, y_m)=(x_1, \ldots, x_n)
\end{array}$$

Fig. 6. Rules for local inversion

The result of backward reading the symmetric representation of *pack* is shown in Fig. 12. Compare *pack* before and after the inversion. Each atomic operation is inverted according to the rules in Fig. 6. Program $pack^{-1}$ is inverse to *pack*, but nondeterministic. We cannot translate len^{-1} directly into a functional program since the call $len^{-1}(p, t)=(c, z, r)$ is not guarded by pattern matching—the reader is welcome to try.

Definition 3 (local inversion). *Let q be a symmetric program well-formed for inversion. Then, local inversion of q is defined by*

$$INV[\![q]\!] \overset{\mathrm{def}}{=} \{Inv[\![d]\!] \mid d \in q\}$$

4.3 Translation to the Grammar Language

After the inversion of atomic operations, nondeterminism can be eliminated by viewing the program as a grammar. To make it easier to manipulate programs, we hide variables by translating them into a grammar-like language. That language operates on a stack instead of an environment. Each atomic operation in the symmetric language is converted into a sequence of stack operations. The syntax of the grammar language is shown in Fig. 7. A stack operation t is either a constructor application $c!$, a pattern matching $c?$, an application of $\lfloor \_ \rfloor$, a function call f, or a selection $(i_1, \ldots, i_n)$. Each stack operation operates on top of the stack for input/output of the corresponding number of values, except for selection which moves each i_jth stack element to the jth position on the stack. This is convenient for reordering the stack. For instance, the sequence (2) _:_**?** swaps the two top-most values and, if the new top-most value is a cons, pops it and pushes its head and tail components; otherwise the sequence fails. The result of translating *pack* from the symmetric language into the grammar language is shown in Fig. 12. The translation is defined in Fig. 8 where " $+\!\!+$ " appends two lists.

Definition 4 (midend). *Let d be a definition in a program well-formed for inversion. Then, the translation from the symmetric language to the grammar language is defined by*

$q ::= d_1 \dots d_n$	(program)
$d ::= f \to t_1 \dots t_n$	(definition)
$t ::= c!$	(constructor)
$\mid\ c?$	(pattern matching)
$\mid\ \lfloor\text{-}\rfloor$	(duplication/equality)
$\mid\ f$	(function call)
$\mid\ (i_1, \dots, i_n)$	(selection)

Fig. 7. Abstract syntax of the grammar language

$$\begin{array}{ll}
\mathit{Gram}[\![\ f \to \mathbf{in}(xs)\ ts\ ]\!] & = f \to \mathit{gram}[\![\ ts,\ xs\ ]\!] \\
\mathit{gram}[\![\ \mathbf{out}(xs),\ xs\ ]\!] & = \epsilon \\
\mathit{gram}[\![\ c(xs){=}y\ ts,\ zs\ ]\!] & = (is)\ c!\ \mathit{gram}[\![\ ts,\ y{:}zs|_{xs}\ ]\!] \\
\mathit{gram}[\![\ x{=}c(ys)\ ts,\ zs\ ]\!] & = (i)\ c?\ \mathit{gram}[\![\ ts,\ ys +\!\!+ zs|_{x}\ ]\!] \\
\mathit{gram}[\![\ \lfloor x \rfloor{=}y\ ts,\ zs\ ]\!] & = (i)\ \lfloor\text{-}\rfloor\ \mathit{gram}[\![\ ts,\ y{:}zs|_{x}\ ]\!] \\
\mathit{gram}[\![\ f(xs){=}(ys)\ ts,\ zs\ ]\!] & = \begin{cases} f\ \mathit{gram}[\![\ ts,\ ys +\!\!+ zs|_{xs}\ ]\!] & \text{if } x_j = z_j \text{ for } 1 \le j \le n \\ (is)\ f\ \mathit{gram}[\![\ ts,\ ys +\!\!+ zs|_{xs}\ ]\!] & \text{otherwise} \end{cases}
\end{array}$$

Notation: given xs and zs, (is) is an abbreviation for selection $(i_1, \dots, i_n)$ where number i_j is the index of x_j in zs for $1 \le j \le n$; in particular, i is the index of x in zs; notation $zs|_{xs}$ denotes the deletion of all xs in zs.

Fig. 8. Translation from the symmetric language to the grammar language

$$\mathit{GRAM}[\![\ q\ ]\!] \overset{\text{def}}{=} \{\mathit{Gram}[\![\ d\ ]\!] \mid d \in q\}$$

4.4 Eliminating Nondeterminism

An LR(k) parser generator produces a deterministic parser given a context free grammar, provided that the grammar is LR(k). This class of parsing methods is used in practically all parser generators (*e.g.*, yacc) because it allows to parse most programming language grammars. Our goal is to eliminate nondeterminism from an inverse program. For this we will resort to the particular method of LR(0) parsing. This parsing method is simpler than LR(1) parsing in that it does not require the use of a lookahead operation in the generated parsers.

We found that LR(0) covers a large class of inverse programs. For example, a tail-recursive program can be viewed as a right-recursive grammar; the recursive call is at the end. Local inversion of a tail-recursive program always leads to an inverse program that corresponds to a left-recursive grammar, the recursive call is now at the beginning. Immediately, we face the problem of nondeterminism because it represents an unguarded choice between immediately choosing the recursive call or the base case. Such a program cannot be represented in a functional language. This requires a transformation into a functionally equivalent form where each choice is guarded by a conditional (see also Sect. 3). LR(0) parsing allows us to deal directly with this type of grammars and, in many cases, to convert the program into a deterministic version.

This is a main motivation for applying the method of LR(0) parsing, namely to derive deterministic inverse programs. Our method makes use of some of the methods of classical LR(0) parser generation, for example the construction of item sets by a closure operation, but generates a functional program instead of a table- or program-driven parser:

1. Item sets: given the grammar representation of a program, the items sets are computed by a closure operation.
2. Code generation: given conflict-free item sets, a deterministic functional program is generated.

We will now discuss these operations in more detail. We assume that the reader is familiar with the main principles of LR parsing, *e.g.,* as presented in [3]. Due to space limitations we cannot further review LR parsing and use standard terminology without further definitions (*e.g.,* item set, closure operation, shift/reduce action). We show how these operations are adopted to our grammar language.

Remark: In our previous work [8, Sect.4], we applied a local criterion to a source program to ensures that the inverse program corresponds to an LL grammar. Since LL parsing is strictly weaker than LR parsing, we conclude that applying an LR parsing approach to program inversion leads to a strictly stronger inversion algorithm. Recall that LL parsing cannot directly deal with left-recursive grammars and that any LL grammar can be parsed by an LR parser, but not vice versa.

Item Sets We define a parse item of the grammar language (Fig. 7) by

$$f \rightarrow ts_1 \cdot ts_2$$

where '$\cdot$' denotes the current position. To compute the sets of items sets, we define two operations which correspond to determining the parse actions: *shift* $I \overset{t}{\rightsquigarrow} I'$ from item set I to item set I' under symbol t and *reduce* $I \overset{n}{\hookrightarrow} f$ from item set I by function symbol f and number of operations n.

$$I_1 \overset{t}{\rightsquigarrow} I_2 \iff I_2 = \{f \rightarrow ts_1\ t \cdot ts_2 \mid f \rightarrow ts_1 \cdot t\ ts_2 \in closure[\![\ I_1\ ]\!]\} \quad \text{(shift)}$$
$$I \overset{n}{\hookrightarrow} f \iff f \rightarrow t_1 \ldots t_n \cdot \in closure[\![\ I\ ]\!] \quad \text{(reduce)}$$

Given an initial item set I_0, the set $\mathcal{I}$ of all reachable item sets is defined by

$$\mathcal{I} = \{I \mid I_0 \overset{*}{\rightsquigarrow} I\}$$

$$I_0 = \{\mathit{entry} \rightarrow \cdot\,\mathit{pack}^{-1}\}$$

$$I_1 = \left\{\begin{array}{l} \mathit{pack}^{-1} \rightarrow (1)\cdot [\,]?\ ()\ [\,]!\ (1) \\ \mathit{pack}^{-1} \rightarrow (1)\cdot \_{:}\_?\ (2)\ \mathit{pack}^{-1}\ (2,1)\ \mathit{len}^{-1}\ (2)\ \mathrm{O}?\ (1,2)\ \_{:}\_!\ (1) \end{array}\right\}$$

$$I_2 = \{\mathit{pack}^{-1} \rightarrow (1)\ [\,]?\cdot ()\ [\,]!\ (1)\}$$

$$\dots$$

$$I_5 = \{\mathit{pack}^{-1} \rightarrow (1)\ [\,]?\ ()\ [\,]!\ (1)\cdot\}$$

$$I_6 = \{\mathit{pack}^{-1} \rightarrow (1)\ \_{:}\_?\cdot (2)\ \mathit{pack}^{-1}\ (2,1)\ \mathit{len}^{-1}\ (2)\ \mathrm{O}?\ (1,2)\ \_{:}\_!\ (1)\}$$

$$\dots$$

$$I_9 = \{\mathit{pack}^{-1} \rightarrow (1)\ \_{:}\_?\ (2)\ \mathit{pack}^{-1}\ (2,1)\cdot \mathit{len}^{-1}\ (2)\ \mathrm{O}?\ (1,2)\ \_{:}\_!\ (1)\}$$

$$I_{10} = \left\{\begin{array}{l} \mathit{pack}^{-1} \rightarrow (1)\ \_{:}\_?\ (2)\ \mathit{pack}^{-1}\ (2,1)\ \mathit{len}^{-1}\cdot (2)\ \mathrm{O}?\ (1,2)\ \_{:}\_!\ (1) \\ \mathit{len}^{-1} \rightarrow \mathit{len}^{-1}\cdot (2)\ \mathrm{S}?\ (2)\ \langle\_\rangle!\ (1)\ \lfloor\_\rfloor(1)\ \langle\_,\_\rangle?\ (2,4)\ \_{:}\_!\ (2,3,1) \end{array}\right\}$$

$$I_{11} = \left\{\begin{array}{l} \mathit{pack}^{-1} \rightarrow (1)\ \_{:}\_?\ (2)\ \mathit{pack}^{-1}\ (2,1)\ \mathit{len}^{-1}\ (2)\cdot \mathrm{O}?\ (1,2)\ \_{:}\_!\ (1) \\ \mathit{len}^{-1} \rightarrow \mathit{len}^{-1}\ (2)\cdot \mathrm{S}?\ (2)\ \langle\_\rangle!\ (1)\ \lfloor\_\rfloor(1)\ \langle\_,\_\rangle?\ (2,4)\ \_{:}\_!\ (2,3,1) \end{array}\right\}$$

$$I_{12} = \{\mathit{pack}^{-1} \rightarrow (1)\ \_{:}\_?\ (2)\ \mathit{pack}^{-1}\ (2,1)\ \mathit{len}^{-1}\ (2)\ \mathrm{O}?\cdot (1,2)\ \_{:}\_!\ (1)\}$$

$$\dots$$

$$I_{15} = \{\mathit{pack}^{-1} \rightarrow (1)\ \_{:}\_?\ (2)\ \mathit{pack}^{-1}\ (2,1)\ \mathit{len}^{-1}\ (2)\ \mathrm{O}?\ (1,2)\ \_{:}\_!\ (1)\cdot\}$$

$$I_{16} = \{\mathit{len}^{-1} \rightarrow \mathit{len}^{-1}\ (2)\ \mathrm{S}?\cdot (2)\ \langle\_\rangle!\ (1)\ \lfloor\_\rfloor\ (1)\ \langle\_,\_\rangle?\ (2,4)\ \_{:}\_!\ (2,3,1)\}$$

$$\dots$$

$$I_{25} = \{\mathit{len}^{-1} \rightarrow \mathit{len}^{-1}\ (2)\ \mathrm{S}?\ (2)\ \langle\_\rangle!\ (1)\ \lfloor\_\rfloor\ (1)\ \langle\_,\_\rangle?\ (2,4)\ \_{:}\_!\ (2,3,1)\cdot\}$$

$$I_{26} = \left\{\begin{array}{l} \mathit{len}^{-1} \rightarrow (2)\cdot [\,]?\ (1)\ \langle\_,\_\rangle?\ ()\ [\,]!\ (2,3,1) \\ \mathit{len}^{-1} \rightarrow (2)\cdot \_{:}\_?\ (3)\ \langle\_,\_\rangle?\ (1,3)\ \langle\_,\_\rangle!\ (1)\ \lfloor\_\rfloor\ (1)\ \langle\_,\_\rangle?\ (2,4)\ \_{:}\_!\ (2,3,1) \end{array}\right\}$$

$$I_{27} = \{\mathit{len}^{-1} \rightarrow (2)\ [\,]?\cdot (1)\ \langle\_,\_\rangle?\ ()\ [\,]!\ (2,3,1)\}$$

$$\dots$$

$$I_{32} = \{\mathit{len}^{-1} \rightarrow (2)\ [\,]?\ (1)\ \langle\_,\_\rangle?\ ()\ [\,]!\ (2,3,1)\cdot\}$$

$$I_{33} = \{\mathit{len}^{-1} \rightarrow (2)\ \_{:}\_?\cdot (3)\ \langle\_,\_\rangle?\ (1,3)\ \langle\_,\_\rangle!\ (1)\ \lfloor\_\rfloor\ (1)\ \langle\_,\_\rangle?\ (2,4)\ \_{:}\_!\ (2,3,1)\}$$

$$\dots$$

$$I_{44} = \{\mathit{len}^{-1} \rightarrow (2)\ \_{:}\_?\ (3)\ \langle\_,\_\rangle?\ (1,3)\ \langle\_,\_\rangle!\ (1)\ \lfloor\_\rfloor\ (1)\ \langle\_,\_\rangle?\ (2,4)\ \_{:}\_!\ (2,3,1)\cdot\}$$

Fig. 9. pack^{-1}: item sets

where $I_1 \overset{*}{\leadsto} I_2 \iff I_1 = I_2 \vee \exists t\, \exists I' .\ I_1 \overset{t}{\leadsto} I' \wedge I' \overset{*}{\leadsto} I_2$. For our running example, several selected item sets are listed in Fig. 9.

As known from LR parsing, some item sets may be *inadequate,* that is, they contain a shift/reduce or a reduce/reduce conflict. In addition to these two classical conflicts, we have a conflict which is specific to our problem domain (a shift/shift conflict): only pattern matching operations are semantically significant *wrt* to the choice of alternatives; while other operations do not contribute to such a choice. Both shift must pass over different matching operations. With Match we denote the set of all matching operations $c?$ in the grammar language.

$$
\begin{array}{lll}
\mathit{Det}\llbracket\, I_i,\ \mathit{is}\,\rrbracket & = \{ f_{i:is} \rightarrow a\ \mathit{ctxt}\llbracket\, I_i,\ I_j,\ \mathit{is}\,\rrbracket \mid I_i \overset{a}{\rightsquigarrow} I_j \} & \\
\mathit{gen}\llbracket\, I_i,\ \mathit{is}\,\rrbracket & = a\ \mathit{ctxt}\llbracket\, I_i,\ I_j,\ \mathit{is}\,\rrbracket & \text{if } S_i = \{(a, I_j)\} \\
\mathit{gen}\llbracket\, I_i,\ \mathit{is}\,\rrbracket & = f_{i:is} & \text{if } S_i \supset \{(a, I_j)\} \\
\mathit{gen}\llbracket\, I_i,\ \mathit{is}\,\rrbracket & = \mathit{cut}\llbracket\, I_i,\ \mathit{is},\ f,\ n\,\rrbracket & \text{if } I_i \overset{n}{\hookrightarrow} f \\
\mathit{ctxt}\llbracket\, I_0,\ I_j,\ \mathit{is}\,\rrbracket & = \mathit{gen}\llbracket\, I_j,\ \mathit{is}\,\rrbracket & \\
\mathit{ctxt}\llbracket\, I_i,\ I_j,\ \mathit{is}\,\rrbracket & = \mathit{gen}\llbracket\, I_j,\ [\,]\,\rrbracket\ \mathit{cut}\llbracket\, I_i,\ i{:}\mathit{is},\ f,\ n\,\rrbracket & \text{if } R_j = \{(f, n)\} \\
\mathit{ctxt}\llbracket\, I_i,\ I_j,\ \mathit{is}\,\rrbracket & = \mathit{gen}\llbracket\, I_j,\ i{:}\mathit{is}\,\rrbracket & \text{otherwise} \\
\mathit{cut}\llbracket\, I_i,\ [i_1, \ldots, i_m],\ f,\ n\,\rrbracket & = \epsilon & \text{if } m < n \\
\mathit{cut}\llbracket\, I_i,\ [i_1, \ldots, i_n, \ldots, i_m],\ f,\ n\,\rrbracket & = \mathit{ctxt}\llbracket\, I_{i_n},\ I_j,\ [i_{n+1}, \ldots, i_m]\,\rrbracket & \text{if } I_{i_n} \overset{f}{\rightsquigarrow} I_j \\
\qquad\text{where} & S_i = \{(a, I_j) \mid I_i \overset{a}{\rightsquigarrow} I_j\} & \\
 & R_i = \{(f, n) \mid f \rightarrow t_1 \ldots t_n \cdot ts \in I_i\} &
\end{array}
$$

Fig. 10. Code generation

$$
\begin{array}{ll}
I \overset{t}{\rightsquigarrow} I' \wedge I \overset{n}{\hookrightarrow} f & \text{(shift/reduce)} \\
I \overset{n_1}{\hookrightarrow} f_1 \wedge I \overset{n_2}{\hookrightarrow} f_2 \wedge (f_1, n_1) \neq (f_2, n_2) & \text{(reduce/reduce)} \\
I \overset{t_1}{\rightsquigarrow} I_1 \wedge I \overset{t_2}{\rightsquigarrow} I_2 \wedge t_1 \neq t_2 \wedge \{t_1, t_2\} \not\subseteq \text{Match} & \text{(shift/shift)}
\end{array}
$$

Code Generation Given the shift and reduce relations, we now define the code generation. Code generation is only applied if all sets of items are conflict-free. Instead of generating a table- or procedure-driven parser, we generate a program in our grammar representation, which will then be converted into a functional program. The main task of the code generation is to produce for each item set a new function definition in the grammar language. The algorithm makes use of the shift and reduce relations for the given grammar program. It compresses redundant transitions between calls on the fly.

Fig. 12 shows the result for our running example. Inversion is successful. Finally, the grammar representation is translated into a syntactically correct functional program. This translation (not defined here) reintroduces variables and converts each operation into functional language construct. It also determines the arity and coarity of functions. This representation is easier to read, but less easy to manipulate. The inverse program pack^{-1} is shown in Fig. 3. We have automatically produced an unpack function from a pack function. For instance, to unpack a packed symbol list: $\mathit{pack}^{-1}([\langle \mathrm{A}, 2\rangle\langle \mathrm{B}, 1\rangle\langle \mathrm{C}, 3\rangle]) = [\mathrm{AABCCC}]$.

For simplicity, we assume that all item sets can be identified by a unique index (I_1, I_2, *etc.*). These indices will be used to generate new function names and tell us about the context of a function call. For each item set I_i we compute a ‘Shift’ set S_i and a ‘Reduce’ set R_i. Set S_i tells us the item set I_j to which

$R_1 = \{(pack^{-1}, 1)\}$	$S_1 = \{([\,]?, I_2), (\_:\_?, I_6)\}$	
$R_2 = \{(pack^{-1}, 2)\}$	$S_2 = \{((), I_3)\}$	
...		
$R_5 = \{(pack^{-1}, 5)\}$		$I_5 \overset{5}{\hookrightarrow} pack^{-1}$
$R_6 = \{(pack^{-1}, 2)\}$	$S_6 = \{((2), I_7)\}$	
$R_7 = \{(pack^{-1}, 3)\}$	$S_7 = \{((1), I_1)\}$	$I_7 \overset{pack^{-1}}{\rightsquigarrow} I_8$
...		
$R_9 = \{(pack^{-1}, 5)\}$	$S_9 = \{((2), I_{26})\}$	$I_9 \overset{len^{-1}}{\rightsquigarrow} I_{10}$
$R_{10} = \{(pack^{-1}, 6), (len^{-1}, 1)\}$	$S_{10} = \{((2), I_{11})\}$	
$R_{11} = \{(pack^{-1}, 7), (len^{-1}, 2)\}$	$S_{11} = \{(\mathrm{O}?, I_{12}), (\mathrm{S}?, I_{16})\}$	
$R_{12} = \{(pack^{-1}, 8)\}$	$S_{12} = \{((1, 2), I_{13})\}$	
...		
$R_{15} = \{(pack^{-1}, 11)\}$		$I_{15} \overset{11}{\hookrightarrow} pack^{-1}$
$R_{16} = \{(len^{-1}, 3)\}$	$S_{16} = \{((2), I_{17})\}$	
...		
$R_{25} = \{(len^{-1}, 12)\}$		$I_{25} \overset{12}{\hookrightarrow} len^{-1}$
$R_{26} = \{(len^{-1}, 1)\}$	$S_{26} = \{([\,]?, I_{27}), (\_:\_?, I_{33})\}$	
$R_{27} = \{(len^{-1}, 2)\}$	$S_{27} = \{((1), I_{28})\}$	
...		
$R_{32} = \{(len^{-1}, 7)\}$		$I_{32} \overset{7}{\hookrightarrow} len^{-1}$
$R_{33} = \{(len^{-1}, 2)\}$	$S_{33} = \{((3), I_{34})\}$	
...		
$R_{44} = \{(len^{-1}, 13)\}$		$I_{44} \overset{13}{\hookrightarrow} len^{-1}$

Fig. 11. $pack^{-1}$: sets for code generation

we reach by performing operation a; set R_i tells us the names f of the functions used in an item set and the number n of operations passed. Functions $gen[\![\cdot]\!]$ and $ctxt[\![\cdot]\!]$ make use of these sets. They are defined in Fig. 10; the R and S sets for our running example are shown in Fig. 11.

Definition 5 (backend). *Let q be a grammar program and I_0 be the initial item set for q. Then, the generation of a deterministic program for a (possibly) nondeterministic program q is defined by*

$$DET[\![q]\!] \overset{\text{def}}{=} DET'[\![Det[\![I_0, [\,]]\!]]\!]$$

$$DET'[\![q]\!] \overset{\text{def}}{=} \begin{cases} q & \textit{if } q' = q \\ DET'[\![q']\!] & \textit{if } q' \neq q \end{cases}$$

$$\textit{where } q' = \bigcup_{f \to ts\ f'_{i:is}\ ts' \in q} Det[\![I_i, is]\!] \ \cup\ q$$

1) Function-to-symmetric translation:

$pack \rightarrow \mathbf{in}(s),\ s{=}[\,],\ [\,]{=}x,\ \mathbf{out}(x)$

$pack \rightarrow \mathbf{in}(s),\ s{=}c{:}r,\ \mathrm{O}{=}x,\ len(c,x,r){=}(p,l),\ pack(l){=}(y),\ p{:}y{=}z,\ \mathbf{out}(z)$

$len \rightarrow \mathbf{in}(c,n,s),\ s{=}[\,],\ \langle c,n\rangle{=}x,\ [\,]{=}y,\ \mathbf{out}(x,y)$

$len \rightarrow \mathbf{in}(c,n,s),\ s{=}d{:}r,\ \langle c,d\rangle{=}x,\ \lfloor x\rfloor{=}y,$
$\quad y{=}\langle c\rangle,\ \mathrm{S}(n){=}z,\ len(c,z,r){=}(p,t),\ \mathbf{out}(p,t)$

$len \rightarrow \mathbf{in}(c,n,s),\ s{=}d{:}r,\ \langle c,d\rangle{=}x,\ \lfloor x\rfloor{=}y,$
$\quad y{=}\langle e,f\rangle,\ \langle e,n\rangle{=}z,\ f{:}r{=}w,\ \mathbf{out}(z,w)$

2) Local inversion:

$pack^{-1} \rightarrow \mathbf{in}(x),\ x{=}[\,],\ [\,]{=}s,\ \mathbf{out}(s)$

$pack^{-1} \rightarrow \mathbf{in}(z),\ z{=}p{:}y,\ pack^{-1}(y){=}(l),\ len^{-1}(p,l){=}(c,x,r),\ x{=}\mathrm{O},\ c{:}r{=}s,\ \mathbf{out}(s)$

$len^{-1} \rightarrow \mathbf{in}(x,y),\ y{=}[\,],\ x{=}\langle c,n\rangle,\ [\,]{=}s,\ \mathbf{out}(c,n,s)$

$len^{-1} \rightarrow \mathbf{in}(p,t),\ len^{-1}(p,t){=}(c,z,r),\ z{=}\mathrm{S}(n),\ \langle c\rangle{=}y,$
$\quad \lfloor y\rfloor{=}x,\ x{=}\langle c,d\rangle,\ d{:}r{=}s,\ \mathbf{out}(c,n,s)$

$len^{-1} \rightarrow \mathbf{in}(z,w),\ w{=}f{:}r,\ z{=}\langle e,n\rangle,\ \langle e,f\rangle{=}y$
$\quad \lfloor y\rfloor{=}x,\ x{=}\langle c,d\rangle,\ d{:}r{=}s,\ \mathbf{out}(c,n,s)$

3) Symmetric-to-grammar translation:

$pack^{-1} \rightarrow (1)\ [\,]\mathbf{?}\ ()\ [\,]\mathbf{!}\ (1)$

$pack^{-1} \rightarrow (1)\ \_{:}\_\mathbf{?}\ (2)\ pack^{-1}\ (2,1)\ len^{-1}\ (2)\ \mathrm{O}\mathbf{?}\ (1,2)\ \_{:}\_\mathbf{!}\ (1)$

$len^{-1} \rightarrow (2)\ [\,]\mathbf{?}\ (1)\ \langle\_,\_\rangle\mathbf{?}\ ()\ [\,]\mathbf{!}\ (2,3,1)$

$len^{-1} \rightarrow len^{-1}\ (2)\ \mathrm{S}\mathbf{?}\ (2)\ \langle\_\rangle\mathbf{!}\ (1)\ \lfloor\_\rfloor\ (1)\ \langle\_,\_\rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1)$

$len^{-1} \rightarrow (2)\ \_{:}\_\mathbf{?}\ (3)\ \langle\_,\_\rangle\mathbf{?}\ (1,3)\ \langle\_,\_\rangle\mathbf{!}\ (1)\ \lfloor\_\rfloor\ (1)\ \langle\_,\_\rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1)$

4) Elimination of non-determinism:

$f_{[0]} \rightarrow (1)\ f_{[1]}$

$f_{[1]} \rightarrow [\,]\mathbf{?}\ ()\ [\,]\mathbf{!}\ (1)$

$f_{[1]} \rightarrow \_{:}\_\mathbf{?}\ (2)\ (1)\ f_{[1]}\ (2,1)\ (2)\ f_{[26]}\ (2)\ f_{[11,10,9]}$

$f_{[11,10,9]} \rightarrow \mathrm{O}\mathbf{?}\ (1,2)\ \_{:}\_\mathbf{!}\ (1)$

$f_{[11,10,9]} \rightarrow \mathrm{S}\mathbf{?}\ (2)\ \langle\_\rangle\mathbf{!}\ (1)\ \lfloor\_\rfloor\ (1)\ \langle\_,\_\rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1)\ (2)\ f_{[11,10,9]}$

$f_{[26]} \rightarrow [\,]\mathbf{?}\ (1)\ \langle\_,\_\rangle\mathbf{?}\ ()\ [\,]\mathbf{!}\ (2,3,1)$

$f_{[26]} \rightarrow \_{:}\_\mathbf{?}\ (3)\ \langle\_,\_\rangle\mathbf{?}\ (1,3)\ \langle\_,\_\rangle\mathbf{!}\ (1)\ \lfloor\_\rfloor\ (1)\ \langle\_,\_\rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1)$

Fig. 12. Inversion of program *pack*

The transformation into a deterministic grammar by $DET[\![\cdot]\!]$ does not terminate iff there exists a loop, $I_j \overset{+}{\leadsto} I_j$, such that all sets R_k of I_k in this loop contain two or more elements. With another transformation that introduces some administrative code, even these programs for which our algorithm does not terminate can be converted into a deterministic grammar program. Our goal was to avoid the introduction of administrative overhead and we found that our transformation is successful for many programs. We omit the definition of $FCT[\![\cdot]\!]$ which translates a grammar program back into a functional program.

5 Related Work

The method presented in this paper is based on the principle of global inversion based on local invertibility [6,11]. The work was originally inspired by KEinv [13,7]. In contrast to KEinv, our method can successfully deal with equality and duplication of variables. Most studies on functional languages and program inversion have involved program inversion by hand (*e.g.,* [14]). They may be more powerful at the price of automation. This is the usual trade-off. Inversion based on Refal graphs [16,10,17,15] is related to the present method in that both use atomic operations for inversion. An algorithm for inverse computation can be found in [1,2]. It performs inverse computation also on programs that are not injective; it does not produce inverse programs but performs the inversion of a program interpretively.

6 Conclusion

We presented an automatic method for deriving deterministic inverse programs by adopting techniques known from LR parsing, in particular, LR(0) parsing. We formalized the transformation and illustrated it with an example. This greatly expands the application of our recent method for program inversion [8] by eliminating nondeterminism from inverse programs by a global transformation. This allows us to invert programs for which this was not possible before. For example, the method in this paper can invert function *tailcons* and the tail-recursive version of function *reverse* [8, Sect.6].

We have also reached the border line where more inverse programs can be made deterministic, but for the price of introducing additional administrative overhead or the use of LR(k), $k > 0$, that is, parsing methods that involve lookahead operations. It will be a task for future work to study the relative gains by adopting such techniques. We used a grammar-like program representation. Other representations are possible and future work will need to identify which representation is most suitable for eliminating nondeterminism.

Acknowledgements We are grateful to the anonymous reviewers for their detailed and useful feedback.

References

1. S. M. Abramov, R. Glück. Principles of inverse computation and the universal resolving algorithm. In T. Æ. Mogensen, D. Schmidt, I. H. Sudborough (eds.), *The Essence of Computation: Complexity, Analysis, Transformation,* LNCS 2566, 269–295. Springer-Verlag, 2002.
2. S. M. Abramov, R. Glück. The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming,* 43(2-3): 193–229, 2002.
3. A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.
4. R. Bird, O. de Moor. *Algebra of Programming.* Prentice Hall International Series in Computer Science. Prentice Hall, 1997.
5. J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence,* 16(1):1–46, 1981.
6. E. W. Dijkstra. Program inversion. In F. L. Bauer, M. Broy (eds.), *Program Construction: International Summer School,* LNCS 69, 54–57. Springer-Verlag, 1978.
7. D. Eppstein. A heuristic approach to program inversion. In *Int. Joint Conference on Artificial Intelligence (IJCAI-85),* 219–221. Morgan Kaufmann, Inc., 1985.
8. R. Glück, M. Kawabe. A program inverter for a functional language with equality and constructors. In A. Ohori (ed.), *Programming Languages and Systems. Proceedings,* LNCS 2895, 246–264. Springer-Verlag, 2003.
9. R. Glück, Y. Kawada, T. Hashimoto. Transforming interpreters into inverse interpreters by partial evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 10–19. ACM Press, 2003.
10. R. Glück, V. F. Turchin. Application of metasystem transition to function inversion and transformation. In *Proceedings of the Int. Symposium on Symbolic and Algebraic Computation (ISSAC'90),* 286–287. ACM Press, 1990.
11. D. Gries. *The Science of Programming,* chapter 21 Inverting Programs, 265–274. Texts and Monographs in Computer Science. Springer-Verlag, 1981.
12. H. Khoshnevisan, K. M. Sephton. InvX: An automatic function inverter. In N. Dershowitz (ed.), *Rewriting Techniques and Applications. Proceedings,* LNCS 355, 564–568. Springer-Verlag, 1989.
13. R. E. Korf. Inversion of applicative programs. In *Int. Joint Conference on Artificial Intelligence (IJCAI-81),* 1007–1009. William Kaufmann, Inc., 1981.
14. S.-C. Mu, R. Bird. Inverting functions as folds. In E. A. Boiten, B. Möller (eds.), *Mathematics of Program Construction. Proceedings,* LNCS 2386, 209–232. Springer-Verlag, 2002.
15. A. P. Nemytykh, V. A. Pinchuk. Program transformation with metasystem transitions: experiments with a supercompiler. In D. Bjørner, M. Broy, I. V. Pottosin (eds.), *Perspectives of System Informatics. Proceedings,* LNCS 1181, 249–260. Springer-Verlag, 1996.
16. A. Y. Romanenko. Inversion and metacomputation. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation,* 12–22. ACM Press, 1991.
17. V. F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming,* 3(3):283–313, 1993.

Author Index

This page intentionally left blank