Lecture Notes in Computer Science 5133

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Philippe Audebaud
Christine Paulin-Mohring (Eds.)

Mathematics of Program Construction

9th International Conference, MPC 2008
Marseille, France, July 15-18, 2008
Proceedings

Springer

Volume Editors

Philippe Audebaud
Ecole Normale Supérieure (ENS) de Lyon
Laboratoire de l'Informatique du Parallélisme (LIP)
46 allée d'Italie, 69364 Lyon CEDEX 07, France
E-mail: Philippe.Audebaud@ens-lyon.fr

Christine Paulin-Mohring
INRIA Saclay - Île-de-France, ProVal
91893 Orsay CEDEX, France
and
LRI, Université Paris Sud, CNRS
Bât. 490, 91405 Orsay CEDEX, France
E-mail: christine.paulin@lri.fr

Library of Congress Control Number: 2008930417

CR Subject Classification (1998): F.3, F.4, D.2, F.1, D.3

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-70593-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-70593-2 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springer.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12324385 06/3180 5 4 3 2 1 0

Preface

This volume contains the proceedings of MPC 2008, the 9th International Conference on the Mathematics of Program Construction. This series of conferences aims to promote the development of mathematical principles and techniques that are demonstrably useful in the process of constructing computer programs, whether implemented in hardware or software. The focus is on techniques that combine precision with conciseness, enabling programs to be constructed by formal calculation. Within this theme, the scope of the series is very diverse, including programming methodology, program specification and transformation, programming paradigms, programming calculi, and programming language semantics.

The quality of the papers submitted to the conference was in general very high, and the number of submissions was comparable to that for the previous conference. Each paper was refereed by at least four, and often more, committee members.

This volume contains 18 papers selected for presentation by the Program Committee from 41 submissions, 1 invited paper which was reviewed as well, and the abstracts for two invited talks.

The conference took place in Marseille-Luminy, France. The previous eight conferences were held in 1989 in Twente, The Netherlands; in 1992 in Oxford, UK; in 1995 in Kloster Irsee, Germany; in 1998 in Marstrand near Göteborg, Sweden; in 2000 in Ponte de Lima, Portugal; in 2002 in Dagstuhl, Germany; in 2004, in Stirling, UK; and in 2006 in Kuressaare, Estonia. The proceedings of these conferences were published as LNCS 375, 669, 947, 1422, 1837, 2386, 3125 and 4014, respectively.

We are grateful to the members of the Program Committee and their referees for their care and diligence in reviewing the submitted papers.

The review process and compilation of the proceedings were greatly helped by Andrei Voronkov's EasyChair system that we can only recommend to every Program Chair.

May 2008

Christine Paulin-Mohring
Philippe Audebaud

Organization

Program Chairs

Philippe Audebaud (Ecole Normale Supérieure Lyon, France)
Christine Paulin-Mohring (INRIA-Université Paris-Sud, France)

Program Committee

Ralph-Johan Back (Abo Akademi University, Finland)
Eerke Boiten (University of Kent, UK)
Venanzio Capretta (University of Nijmegen, The Netherlands)
Sharon Curtis (Oxford Brookes University, UK)
Jules Desharnais (Université Laval, Québec, Canada)
Peter Dybjer (Chalmers University of Technology, Sweden)
Jeremy Gibbons (University of Oxford, UK)
Lindsay Groves (Victoria University of Wellington, New Zealand)
Ian Hayes (University of Queensland, Australia)
Eric Hehner (University of Toronto, Canada)
Johan Jeuring (Utrecht University, The Netherlands)
Dexter Kozen (Cornell University, USA)
Christian Lengauer (Universität Passau, Germany)
Lambert Meertens (University of Utrecht, The Netherlands)
Bernhard Möller (Universität Augsburg, Germany)
Carroll Morgan (University of New South Wales, Australia)
Shin-Cheng Mu (Academia Sinica, Taiwan)
Jose Nuno Oliveira (Universidade do Minho, Portugal)
Tim Sheard (Portland State University, USA)
Tarmo Uustalu (Institute of Cybernetics Tallin, Estonia)

External Reviewers

Andreas Abel
Ki Yung Ahn
Jose Bacelar Almeida
Thorsten Altenkirch
Sven Apel
Mats Aspnäs
Luis Barbosa
Bruno Barras
Luc Bougé
Edwin Brady
Robert Colvin
Alcino Cunha
Jean-Lou De Carufel
Catherine Dubois
Roger Duke
Jean-Christophe Filliâtre
Johan Glimming
Stéphane Glondu
Roland Glück
Armin Groesslinger
Peter Hancock
Ichiro Hasuo
Christoph Herrmann
Thomas Hildebrandt
Stefan Holdermans
Peter Höfner
Patrik Jansson
Yasuo Kawahara
Sean Leather
Michael Leuschel

Chuan-Kai Lin
Nathan Linger
Andres Loeh
José Pedro Magalhães
Adam Megacz
Larissa Meinicke
Bruno Oliveira
Randy Pollack
Viorel Preoteasa
Alexey Rodriguez
Ando Saabas
Jorge Sousa Pinto
Barney Stratford
Wouter Swierstra
Varmo Vene
Meng Wang
Jan Westerholm
Kirsten Winter
Martin Ziegler

Local Organization

Philippe Audebaud, Christine Paulin-Mohring and Marie-Renée Donnadieu (Université de la Méditerranée Luminy, France).

Host Institution

The conference was hosted by the Centre International de Recherches en Mathématiques Luminy (CIRM), at Luminy, near Marseille, France.

We would like to thanks INRIA and CIRM for their financial support to the conference.

We are grateful to the CIRM management team, LRI staff, and the INRIA Service communication et colloques (especially Emmanuelle Perrot) for their help in the organization of the conference.

Table of Contents

Exploiting Unique Fixed Points*

Ralf Hinze

Computing Laboratory, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
ralf.hinze@comlab.ox.ac.uk
http://www.comlab.ox.ac.uk/ralf.hinze/

Abstract. Functional programmers happily use equational reasoning and induction to prove properties of recursive programs. To show properties of corecursive programs they employ coinduction, per perhaps less enthusiastically. Coinduction is often considered as a rather low-level proof method, especially, as it seems to depart rather radically from equational reasoning. In this talk we introduce an alternative proof technique based on unique fixed points. To make the idea concrete, consider the simplest example of a coinductive type: the type of streams, where a stream is an infinite sequence of elements. In a lazy functional language, such as Haskell, streams are easy to define and many textbooks on Haskell reproduce the folklore examples of Fibonacci or Hamming numbers defined by recursion equations over streams. One has to be a bit careful in formulating a recursion equation basically avoiding that the sequence defined swallows its own tail. However, if this care is exercised, the equation even possesses a unique solution, a fact that is not very widely appreciated. Uniqueness can be exploited to prove that two streams are equal: if they satisfy the same recursion equation, then they are! We will use this proof technique to infer some intriguing facts about particular streams and to develop the basics of finite calculus. Quite attractively, the resulting proofs have a strong equational flavour. In a nutshell, the proof method brings equational reasoning to the coworld. Of course, it is by no means restricted to streams and can be used equally well to prove properties of infinite trees or the observational equivalence of instances of an abstract datatype.

* Invited Lecture.

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, p. 1, 2008.

Scrap Your Type Applications

Barry Jay[1] and Simon Peyton Jones[2]

[1] University of Technology, Sydney
[2] Microsoft Research Cambridge

Abstract. System **F** is ubiquitous in logic, theorem proving, language meta-theory, compiler intermediate languages, and elsewhere. Along with its type abstractions come *type applications*, but these often appear redundant. This redundancy is both distracting and costly for type-directed compilers.

We introduce System **IF**, for *implicit* System **F**, in which many type applications can be made implicit. It supports decidable type checking and strong normalisation. Experiments with Haskell suggest that it could be used to reduce the amount of intermediate code in compilers that employ System **F**.

System **IF** constitutes a first foray into a new area in the design space of typed lambda calculi, that is interesting in its own right and may prove useful in practice.

1 Introduction

The polymorphic lambda calculus or System **F** is ubiquitous in many areas of computer science such as logic, e.g. (Girard et al. 1989; Girard 1990), programming, e.g. (Reynolds 1974), theorem-proving, e.g. (Coq), and intermediate languages for compilers, e.g. (Peyton Jones 2003; Harper and Morrisett 1995). System **F** is, however, tiresomely verbose. For example, suppose the first projection from a pair is given by `fst` : $\forall a.\forall b.(a, b) \rightarrow a$. Then the first projection of the pair `(3,True)` is given by

```
fst Int Bool (3,True)
```

where the two type arguments, `Int` and `Bool`, are required to instantiate the type variables a and b. To a naive observer, the type arguments seem redundant. After all, if we were to write simply

```
fst (3,True)
```

then it is clear how to instantiate a and b! And indeed many *source* languages omit type arguments, relying on type inference to fill them in. However our interest is in typed *calculi* with the power of System F, for which type inference known to be undecidable (Wells 1994). More precisely, we address the following question: can we omit type arguments in a polymorphic calculus with the full expressive power of System **F**, without losing decidable type checking? Our contributions are as follows:

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 2–27, 2008.

- We present a new, explicitly-typed lambda calculus, System **IF** (short for "implicit System **F**") that is precisely as expressive as System **F**, but allows many type application to be be scrapped (Section 3). However, it requires four new reduction rules.
- System **IF** enjoys the same desirable properties as System **F**; in particular, type checking is decidable, and reduction is type preserving, confluent, and strongly normalising (Section 3.4). Furthermore, **IF** shares System **F**'s property that every term has a unique type (Section 2.2), which is particularly useful when the calculus is used as intermediate language for a compiler (Section 4.1).
- Every System **F** term is also an System **IF** term; and conversely there is translation from System **IF** to System **F** that preserves typing, type erasure, term equality and inequality (Section 3.3). Reduction itself is not preserved since one reduction rule is reversed during translation.
- We regard System **IF** as of interest in its own right, but it potentially has some practical importance because compilers for higher-order, typed languages often use an explicitly-typed intermediate language based on System **F** (Peyton Jones et al. 1993; Tarditi et al. 1996; Shao 1997), and there is evidence that cost of processing types is a significant problem in practice (Shao et al. 1998; Petersen 2005). To get some idea of whether System **IF** is useful in this context, we adapted the Glasgow Haskell Compiler (GHC), a state-of-the-art compiler for Haskell, to use System **IF** as its intermediate language (Section 4). The results are mixed: 80% of all type applications can be removed, reducing the total size of the code by 12%, but the "bottom line" of compiler execution time is not improved (Section 4).

Our work complements other approaches that reduce the burden of type information in intermediate languages (Section 5), but it is distinctively different: to the best of our knowledge no previous such work specifically considers how to eliminate type applications.

Although the emphasis in this paper is on intermediate languages, the ideas may be of broader significance. In programming, for example, quantified function types allow for the possibility of combining cases whose types quantify different numbers of type variables. In each case the instantiation of type variables will be determined by the argument, not by the programmer, in a form of dynamic dispatch for type variables (Jay 2006).

2 System F

This section recalls System **F** and discusses its redundant type applications.

2.1 The System

The syntax, notation, type system and dynamic semantics for System **F** are given for easy reference in Figure 1. They should be familiar, but they serve to establish our notational conventions.

Syntax

$$\begin{aligned} a, b, c &::= \langle\text{type variables}\rangle \\ f, g, x, y, z &::= \langle\text{term variables}\rangle \end{aligned}$$

$$\begin{aligned} \sigma, \tau, \phi, \psi &::= a \mid \sigma \rightarrow \sigma \mid \forall a.\sigma \\ r, s, t, u &::= x^{\sigma} \mid t\ t \mid t\ \sigma \mid \lambda x^{\sigma}.t \mid \Lambda a.t \\ \Gamma &::= x_1^{\sigma_1}, \ldots, x_n^{\sigma_n} \quad (n \geq 0) \\ \Delta &::= a_1, \ldots, a_n \quad (n \geq 0, a_i \text{ distinct}) \end{aligned}$$

Notation

$$\begin{aligned} \forall\Delta.\sigma &\equiv \forall a_1.\ldots.\forall a_n.\sigma \\ \Lambda\Delta.t &\equiv \Lambda a_1.\ldots.\Lambda a_n.t \\ t\ \Delta &\equiv t\ a_1 \ldots a_n \\ t\ (\theta\ \Delta) &\equiv t\ (\theta a_1)\ \ldots\ (\theta a_n) \quad \text{for } \theta \text{ a type substitution} \\ \mathsf{FTV}(\sigma) &\equiv \text{the free type variables of } \sigma \end{aligned}$$

Type system

$$(\text{fvar})\ \frac{}{\Gamma, x^{\sigma} \vdash x^{\sigma} : \sigma}$$

$$(\text{fapp})\ \frac{\begin{array}{c}\Gamma \vdash r : \sigma \rightarrow \phi \\ \Gamma \vdash u : \sigma\end{array}}{\Gamma \vdash r\ u : \phi} \qquad (\text{fabs})\ \frac{\Gamma, x^{\sigma} \vdash s : \phi \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \lambda x^{\sigma}.s : \sigma \rightarrow \phi}$$

$$(\text{tapp})\ \frac{\Gamma \vdash t : \forall a.\sigma}{\Gamma \vdash t\ \psi : \{\psi/a\}\sigma} \qquad (\text{tabs})\ \frac{\Gamma \vdash s : \sigma \quad a \notin \mathsf{FTV}(\Gamma)}{\Gamma \vdash \Lambda a.s : \forall a.\sigma}$$

Dynamic semantics

$$\begin{aligned} (\beta 1)\ (\lambda x^{\sigma}.s)\ u &\longrightarrow \{u/x\}s \\ (\beta 2)\ \ (\Lambda a.s)\ \psi &\longrightarrow \{\psi/a\}s \end{aligned}$$

Fig. 1. Definition of System **F**

The standard definitions apply for: the free type variables $\mathsf{FTV}(\sigma)$ of a type σ; the free type variables $\mathsf{FTV}(t)$ of a term t; the free term variables $\mathsf{ftv}(t)$ of t; type and term substitutions; and the α-equivalence relations for re-naming bound type and term variables. The notation $\{\sigma_1/a_1, \ldots, \sigma_n/a_n\}$ is the substitution mapping a_i to σ_i, and $\{\sigma_1/a_1, \ldots, \sigma_n/a_n\}\phi$ is the result of applying that substitution to the type ϕ. (And similarly for terms.) A type environment Γ is a partial function of finite domain, from term variables to types. The notation $x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$ may be used to denote the function that maps x_i to σ_i. The domain of Γ is denoted $\mathsf{dom}(\Gamma)$.

The symbol Δ stands for a sequence of distinct type variables $a_1, \ldots, a_n$, and we may write $\forall\Delta.\sigma$ to abbreviate $\forall a_1.\ldots.\forall a_n.\sigma$. Similar syntactic sugar abbreviates type abstractions $\Lambda\Delta.t$ and type applications $t\ \Delta$ and even $t\ (\theta\Delta)$ for a type substitution θ.

The dynamic semantics is expressed using the reduction rules ($\beta 1$) and ($\beta 2$) which generate a rewriting relation in the usual way.

In our examples, we often use type constants `Int` and `Bool`, along with corresponding term constants `0`, `1`, ..., `True`, `False`, `(+)`, `(&&)`, and so on. We also assume a pair type (σ, ϕ), and a list type $\texttt{List}\ \sigma$; with term constructors

$$\begin{aligned} \texttt{Pair} &: \forall a, b. a \to b \to (a, b) \\ \texttt{Nil} &: \forall a. \texttt{List}\ a \\ \texttt{Cons} &: \forall a. a \to \texttt{List}\ a \to \texttt{List}\ a. \end{aligned}$$

Formally, these are just notations for their Church encodings, but no harm comes from considering them as extensions of the language.

2.2 Uniqueness of Types

In Girard's original notation, each term variable is annotated with its type, at its occurrences as well as its binding site, but there are no type environments. Later treatments commonly include a type environment which enforces the invariant that every free occurrence of a term variable has the same type, an approach we follow in Figure 1. The type environment allows the type annotation to be dropped from term-variable occurrences, but we nevertheless retain them as a source of type information which will prove useful in the new system. For example, we write

$$(\lambda x^{\texttt{Int}}.\texttt{negate}^{\texttt{Int}\to\texttt{Int}}\ x^{\texttt{Int}}).$$

This notation looks somewhat verbose, in conflict with the goals of the paper, and indeed in our informal examples we often omit type attribution on variable occurrences. However, in practice there is no overhead to this apparent redundancy, as we discuss in Section 4.1, and it has an important practical benefit: every term has a unique type.

Theorem 1 (Uniqueness of types). *If $\Gamma \vdash t : \sigma$ and $\Gamma' \vdash t : \sigma'$ then $\sigma = \sigma'$.*

We can therefore write, without ambiguity, $t : \sigma$ or t^σ to mean that there is some Γ such that $\Gamma \vdash t : \sigma$.

This unique-type property is extremely convenient for a compiler that uses System **F** as an intermediate language. Why? Because every term has a unique type *independent of the context in which the term appears*. More concretely, the compiler may straightforwardly compute the type of any given term with a single, bottom-up traversal of the term, applying the appropriate rule of Figure 1 at each node of the term. System **IF** is carefully designed to retain this property.

2.3 Redundant Type Applications

Although System **F**'s *definition* is beautifully concise, its *terms* are rather verbose. This subsection will demonstrate this through some examples, as a motivation for the new systems that will follow.

Our particular focus is on *type applications*, the term form $(t\ \sigma)$. Human beings dislike writing type applications in source programs, because it is usually obvious what they should be, and they are burdensome to write. Rather, in programming languages such as ML and Haskell a type inference system, such as Hindley-Milner, fills them in.

Why are the missing type applications "obvious"? Because the types of the arguments of an application typically determine the appropriate type arguments. The introduction gave one example, but here is another. Suppose we are given terms $\texttt{map} : \forall a, b.(a \to b) \to \texttt{List}\ a \to \texttt{List}\ b$ and $\texttt{treble} : \texttt{Int} \to \texttt{Int}$. Then an application of `map` might look like this:

$$\texttt{map Int Int treble}$$

It is obvious that the type of `treble` immediately fixes both type arguments of `map` to be `Int`, so no information is lost by writing simply `(map treble)`.

The type arguments in System **F** can occasionally become onerous even to a computer. We encountered this in practice when implementing derivable type classes (Hinze and Peyton Jones 2000) in the Glasgow Haskell Compiler (GHC). The implementation transforms an N-ary data constructor into a nested tuple. For example, the Haskell term ($\texttt{C}\ e_1\ e_2\ e_3\ e_4\ e_5$) is transformed to a nested tuple, whose System **F** representation looks like this (where $e_i : \sigma_i$):

$$\begin{array}{l}\texttt{Pair}\ \sigma_1\ (\sigma_2,(\sigma_3,(\sigma_4,\sigma_5)))\ e_1 \\ \qquad (\texttt{Pair}\ \sigma_2\ (\sigma_3,(\sigma_4,\sigma_5))\ e_2 \\ \qquad\qquad (\texttt{Pair}\ \sigma_3\ (\sigma_4,\sigma_5)\ e_3\ (\texttt{Pair}\ \sigma_4\ \sigma_5\ e_4\ e_5))).\end{array}$$

Note the quadratic blow-up in the size of the term, because the type argument at each level of the nest repeats all the types already mentioned in its arguments. We are not the first to notice this problem, and we discuss in Section 5.2 ways of exploiting sharing to reduce the size of the term or of its representation. But it is more direct to simply *omit* the repeated types than to compress them! Furthermore, omitting them is entirely possible in this example: no information is lost by writing just

$$\texttt{Pair}\ e_1\ (\texttt{Pair}\ e_2\ (\texttt{Pair}\ e_3\ (\texttt{Pair}\ e_4\ e_5))).$$

Omitting obviously-redundant type applications in System **F** can therefore lead to an asymptotic reduction in the size of the term. When the calculus is used as an intermediate language in a compiler, reducing the size of terms may lead to improvements in compilation time.

This abbreviated form is, of course, exactly what an ML programmer would write, relying on type inference to fill in missing type arguments. So the reader might wonder: why not simply omit type applications and use type inference to reconstruct them when necessary? We discuss this question in Section 5, but the short answer is this: type inference is undecidable for System **F** (Wells 1994). The trouble is that System **F** is far too expressive for type inference to work. In particular, in System **F** *type arguments may be quantified types.* For example, one might write:

$$\texttt{Pair}\ (\forall a.a \to a)\ \texttt{Int}\ (\Lambda a.\lambda x^a.\, x)\ 4$$

That is, System **F** is *impredicative*. Very few source languages are impredicative, but System **F** certainly is. Furthermore, a compiler that uses System **F** as its typed intermediate language may well exploit that expressiveness. For example, when desugaring mutually recursive bindings, GHC builds tuples whose components are polymporhic values, which in turn requires the tuple to be instantiated at those polytypes.

3 System IF

Thus motivated, we introduce System **IF**, short for "implicit System F", whose definition is given for reference in Figure 2. The key idea is this:

> System **IF** is just like System **F**, except that many type applications may be omitted.

In fact, System **F** is embedded in System **IF**: they have exactly the same types, the same term syntax, and every well-typed term in System **F** is a well-typed term in System **IF**.

But System **IF** has additional well-typed terms that System **F** lacks. In particular, a term of the form $r\ \psi_1\ \ldots\ \psi_n\ u$ in System **F** may be replaced by an equivalent term $r\ u$ of System **IF** provided that the types $\psi_1, \ldots, \psi_n$ can be recovered from the types of r and u. Of course, scrapping such type applications has knock-on effects on the type system and dynamic semantics.

Syntax as for System **F**

Notation as for System **F** plus

$$\begin{aligned} \{\psi/[\Delta]\,\sigma\} &\equiv \text{the most general substitution } \theta \text{ (if any)} \\ &\qquad \text{such that } \mathsf{dom}(\theta) \subseteq \Delta \text{ and } \theta\sigma = \psi \\ \Delta\backslash\sigma &\equiv \Delta\backslash\mathsf{FTV}(\sigma) \end{aligned}$$

Type system as for System **F**, but replacing (fapp) by

$$\text{(ifapp)}\ \frac{\Gamma \vdash r : \forall\Delta.\sigma \to \phi \quad \Gamma \vdash u : \psi}{\Gamma \vdash r\ u : \forall(\Delta\backslash\sigma).\{\psi/[\Delta]\,\sigma\}\phi}\,\mathsf{FTV}(\psi) \cap (\Delta\backslash\sigma) = \emptyset$$

Dynamic semantics as for System **F**, plus

$$\begin{array}{lll} (\xi 1) & r^{\forall a,\Delta.\sigma\to\phi}\ \psi\ u \longrightarrow r\ u & \text{if } a \in \mathsf{FTV}(\sigma)\backslash\Delta \\ (\xi 2) & r^{\forall a,\Delta.\sigma\to\phi}\ \psi\ u \longrightarrow r\ u\ \psi & \text{if } a \notin \mathsf{FTV}(\sigma)\backslash\Delta \\ (\mu 1) & (\Lambda a.t^{\forall\Delta.\sigma\to\phi})\ u^{\psi} \longrightarrow \{\psi/[a,\Delta]\sigma\}t\ u & \text{if } a \in \mathsf{FTV}(\sigma)\backslash\Delta \\ & & \text{and } \mathsf{FTV}(t) \cap \Delta = \emptyset \\ (\mu 2) & (\Lambda a.t^{\forall\Delta.\sigma\to\phi})\ u^{\psi} \longrightarrow \Lambda a.(t\ u) & \text{if } a \notin \mathsf{FTV}(u) \cup (\mathsf{FTV}(\sigma)\backslash\Delta) \end{array}$$

Fig. 2. Definition of System **IF**

3.1 Type System of IF

In System **F**, a term r of type $\forall a.\sigma$ can only be applied to a *type*, but in System **IF** it may also be applied to a *term*. The single new typing rule, (ifapp), specifies how such term applications are typed in System **IF**. The idea is this: if $r : \forall\Delta.\sigma\rightarrow\phi$ and $u : \psi$, then use ψ to instantiate all those type variables $a_i \in \Delta$ that appear free in σ.

For example, suppose $r : \forall a, b.(a \rightarrow \texttt{Int} \rightarrow b) \rightarrow \phi$ and $u : \texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Bool}$. Then if we see the application $(r\ u)$ it is plain that we must instantiate a to `Int` and b to `Bool`; and that no other instantiation will do. To derive this instantiation:

we *match* $(a\rightarrow\texttt{Int}\rightarrow b)$ against $(\texttt{Int}\rightarrow\texttt{Int}\rightarrow\texttt{Bool})$
to find a *substitution* for $\{a, b\}$
namely $\{\texttt{Int}/a, \texttt{Bool}/b\}$.

(The notation $\{\psi/a, \tau/b\}$ stands for a substitution, as explained in Section 2.1.) More formally, we define a *match* of σ against ψ *relative to* a sequence of type variables Δ, written $\{\psi/[\Delta]\sigma\}$, to be a substitution θ whose domain is within Δ such that $\theta\sigma = \phi$. Such a match is *most general* if any other such match factors through it.

The rules for matching are straightforward, rather like those for unification, the only novelty being the treatment of quantified types. For example, to match $\forall d.(\forall b.b\rightarrow b)\rightarrow d$ against $\forall c.a\rightarrow c$, relative to a, first use α-conversion to identify c with d and then bind a to $\forall b.b \rightarrow b$. Note, however, that matching cannot employ such bound variables in either the range or domain of the substitution. For example, $\{\forall c.c \rightarrow a\,/[a]\,\forall c.c \rightarrow c\}$ fails, since one cannot substitute c for a under the quantifier $\forall c$; and similarly $\{\forall c.c \rightarrow c\,/[c]\,\forall c.Int \rightarrow c\}$ fails, since we cannot substitute for c under the quantifier.

These observations motivate the following definition: a type substitution υ *avoids* a type variable a if a is not in the domain or the range of υ.

Theorem 2. *If there is a match of type σ against a type ψ relative to a sequence of type variables Δ, then there is a most general such match. Further, there is at most one most general match, which is denoted $\{\psi/[\Delta]\sigma\}$.*

Proof. Now the most general match is defined as follows:

$$\begin{array}{rll}
\{\psi/[\Delta]a\} = & \{\psi/a\} & \text{if } a \in \Delta \\
\{a/[\Delta]a\} = & \{\} & \text{if } a \notin \Delta \\
\{\psi_1\rightarrow\psi_2/[\Delta]\sigma_1\rightarrow\sigma_2\} = & \text{let } \upsilon_1 = \{\psi_1/[\Delta]\sigma_1\} \text{ in} & \\
& \text{let } \upsilon_2 = \{\upsilon_1\psi_1/[\Delta]\upsilon_1\sigma_1\} \text{ in} & \\
& \upsilon_2 \circ \upsilon_1 & \\
\{\forall a.\psi/[\Delta]\forall a.\sigma\} = & \{\psi/[\Delta]\sigma\} & \text{if this avoids } a \\
\{\psi/[\Delta]\sigma\} = & \text{undefined} & \text{otherwise.}
\end{array}$$

The proof details are by straightforward induction. □

Now let us return to the typing of an application $(r\ u)$. What if the argument type of r does not mention all r's quantified type variables? For example, suppose $r : \forall a, b.(\texttt{Int} \to a) \to \phi$ and $u : \texttt{Int} \to \texttt{Bool}$. Then in the application $(r\ u)$ it is clear that we must instantiate a to `Bool`, but we learn nothing about the instantiation of b. In terms of matching, the match $\{\texttt{Int} \to \texttt{Bool}/[a, b]\texttt{Int} \to a\} = \{\texttt{Bool}/a\}$ does not act on b. That is, $(r\ u)$ is still polymorphic in b, which fact can be expressed by giving $(r\ u)$ the type $\forall b.\{\texttt{Bool}/a\}\phi$. Rather than allow b to become free, the solution is to bind it again in the result type.

Generalising from this example gives the following typing rule for applications, which is shown in Figure 2 and replaces (fapp) in Figure 1:

$$\text{(ifapp)}\ \frac{\Gamma \vdash r : \forall \Delta.\sigma \to \phi \quad \Gamma \vdash u : \psi}{\Gamma \vdash r\ u : \forall(\Delta \backslash \sigma).\{\psi/[\Delta]\,\sigma\}\phi}\mathsf{FTV}(\psi) \cap (\Delta \backslash \sigma) = \emptyset$$

Here $\Delta\backslash\sigma$ is the sub-sequence of Δ consisting of those variables not free in σ; these are the quantified variables that are not fixed by u. Note that (ifapp) only applies if the match exists, which is not always the case; for example there is no match of `Int` against `Bool`.

To illustrate (ifapp) in action, suppose `x` : `Int` and `y` : `Bool`. Then here are some terms and their types:

$$\begin{aligned}
\texttt{Pair Int Bool} &: \texttt{Int} \to \texttt{Bool} \to (\texttt{Int}, \texttt{Bool}) && (1)\\
\texttt{Pair Int Bool x} &: \texttt{Bool} \to (\texttt{Int}, \texttt{Bool}) && (2)\\
\texttt{Pair x} &: \forall b.b \to (\texttt{Int}, b) && (3)\\
\texttt{Pair x Bool} &: \texttt{Bool} \to (\texttt{Int}, \texttt{Bool}) && (4)\\
\texttt{Pair x Bool y} &: (\texttt{Int}, \texttt{Bool}) && (5)\\
\texttt{Pair x y} &: (\texttt{Int}, \texttt{Bool}) && (6)
\end{aligned}$$

The first two examples are well-typed System **F** as well as System **IF**, with the expected types; you do not *have* to drop type applications in System **IF**! Example (3) is more interesting; here the value argument `x` instantiates one, but only one, of the type variables in `Pair`'s type, leaving a term quantified in just one type variable. Incidentally, it would have made no difference if the type of `Pair` had been quantified in the other order $(\forall b, a.a \to b \to (a, b))$: example (3) would still have the same type. Examples (4–6) illustrate that the polymorphic function `(Pair x)` can be applied to a type (examples (4,5)) or to a term (example (6)).

Like **F**, System **IF** is robust to program transformation. For example, suppose $f : \forall\Delta.\sigma_1 \to \sigma_2 \to \phi$, and $g : \forall\Delta.\sigma_2 \to \sigma_1 \to \phi$, so that their types differ only the order of the two arguments. Then if $(f\ t_1\ t_2)$ is well-typed, so is $(g\ t_2\ t_1)$. This property relies on the ability of (ifapp) to re-abstract the variables not bound by the match. In particular, suppose $f : \forall a.a \to \texttt{Int} \to a$, $g : \forall a.\texttt{Int} \to a \to a$. Now consider the application $(f\ \texttt{True}\ 3)$. The partial application $(f\ \texttt{True})$ will instantiate a to `Bool`, yielding a result of type $\texttt{Int} \to \texttt{Bool}$ which is then applied to `3`. Now consider the arguments in the other order, in $(g\ 3\ \texttt{True})$. The partial application $(g\ 3)$ yields a match that does *not* bind a, so the result is re-abstracted over a to give $\forall a.a \to a$. This function can be applied to `True` straightforwardly.

3.2 Dynamic Semantics of System IF

Since (ifapp) admits more terms than (fapp), we need more reductions, too. In particular, it is necessary to reduce terms of the form $(\Lambda a.s)\ u$, where a type-lambda abstraction is applied to a term. A minimum requirement is that reduction should support the following generalisation of ($\beta 1$) to handle the type abstractions:

$$\begin{array}{ll}(\beta 1')\ (\Lambda \Delta.\lambda x^{\sigma}.t)\ u^{\psi} \longrightarrow & \Lambda(\Delta\backslash\sigma).\,\{u/x\}\{\psi/[\Delta]\sigma\}t \\ & \text{if } \mathsf{FTV}(\psi)\cap(\Delta\backslash\sigma)=\emptyset.\end{array}$$

This reduction matches (ifapp) closely. First it finds the match $\{\psi/[\Delta]\sigma\}$ that describes how to instantiate the type variables Δ (or rather, those that appear free in σ). Then it applies this substitution to t, and re-abstracts over the remaining type variables of Δ. The side condition, which can always be made to hold by α-conversion, simply ensures that the new abstraction does not capture any variables in u.

Notationally, this reduction makes use of Theorem 1, extended to System **IF**, which says that every term has a unique type. In the left-hand side of the reduction we write this type as a superscript on the *term*, thus u^{ψ}, although that is not part of the syntax of System **IF**. We could instead re-state the reduction in a way that is more faithful to the operational reality like this:

$$\begin{array}{ll}(\beta 1')\ (\Lambda \Delta.\lambda x^{\sigma}.t)\ u \longrightarrow & \Lambda(\Delta\backslash\sigma).\,\{u/x\}\{\psi/[\Delta]\sigma\}t \\ & \text{if } \mathsf{FTV}(\psi)\cap(\Delta\backslash\sigma)=\emptyset \\ & \text{and } \exists\Gamma.\ \Gamma\vdash u:\psi.\end{array}$$

Now the left-hand side can be matched purely structurally, while ψ is fixed by the new condition on the right-hand side. In operational terms, there is an easy algorithm to find ψ given a (well-typed) term u.

Although it is essential that the left-hand side of ($\beta 1'$) reduces to its right hand side, it is rather unsatisfactory as a one-step reduction *rule*. From a practical perspective, it eliminates a string of lambdas all at once, and may require reduction under a big lambda to bring the left-hand side to the required form. From a theoretical perspective, it will prove to be a consequence of other rules to be introduced now.

The whole point of System **IF** is that one can omit many type applications, but this is not intended to express or create new meanings: when $(r\ u)$ and $(r\ \psi\ u)$ have the same meaning they should have the same normal form. For example, with the rules we have so far, these two terms would be distinct normal forms:

$$(\lambda x^{\forall a.a\to a}.\,x\ \mathtt{Bool}\ \mathtt{True}) \quad\text{and}\quad (\lambda x^{\forall a.a\to a}.\,x\ \mathtt{True}).$$

Their equality is achieved by adding reduction rules ($\xi 1$) and ($\xi 2$) (they also appear in Figure 2):

$$\begin{array}{l}(\xi 1)\ r^{\forall a,\Delta.\sigma\to\phi}\ \psi\ u \longrightarrow r\ u \quad\ \ \text{if } a\in\mathsf{FTV}(\sigma)\backslash\Delta \\ (\xi 2)\ r^{\forall a,\Delta.\sigma\to\phi}\ \psi\ u \longrightarrow r\ u\ \psi\ \text{if } a\notin\mathsf{FTV}(\sigma)\backslash\Delta\end{array}$$

The first eliminates a type application altogether when the immediately following term argument fixes the instantiation of the polymorphic function – that is, when a is mentioned in σ[1]. For example, if Leaf : $\forall a.a \rightarrow$ Tree a then

$$\texttt{Leaf}\ \psi\ u^{\psi} \longrightarrow \texttt{Leaf}\ u^{\psi}.$$

By itself, this rule is not enough, as can be seen in Pair $\psi\ \tau\ s\ t$ since the type of s does not determine the value of τ which is instead given by t. This situation is handled by rule ($\xi 2$).

These two reductions may also be regarded as optimisation rules, that can be applied statically to remove redundant type applications from an System **IF** program. For example:

$$\begin{aligned}\texttt{Pair}\ \psi\ \tau\ s\ t &\longrightarrow \texttt{Pair}\ \psi\ s\ \tau\ t \text{ by } (\xi 2)\\ &\longrightarrow \texttt{Pair}\ s\ \tau\ t \quad\ \text{by } (\xi 1)\\ &\longrightarrow \texttt{Pair}\ s\ t \qquad\ \text{by } (\xi 1)\end{aligned}$$

as desired. Here is another example. Suppose Inl : $\forall a, b.a \rightarrow a + b$, and Inr : $\forall a.\forall b.b \rightarrow a + b$. Then

$$\begin{aligned}\texttt{Inl Int Bool 3} &\longrightarrow \texttt{Inl Int 3 Bool} \text{ by } (\xi 2)\\ &\longrightarrow \texttt{Inl 3 Bool} \qquad \text{by } (\xi 1)\end{aligned}$$

$$\begin{aligned}\texttt{Inr Int Bool True} &\longrightarrow \texttt{Inr Int True} \quad \text{by } (\xi 1)\\ &\longrightarrow \texttt{Inr True Int} \quad \text{by } (\xi 2)\end{aligned}$$

In these two examples, notice that one type application necessarily remains, because the argument of Inl and Inr only fixes one of the type parameters. (At least, it necessarily remains if Inr is regarded as a constant; if it is replaced by its Church encoding then further reductions can take place.) Exactly the same thing happens with the Nil of the list type.

Note that ($\xi 1$) and ($\xi 2$) overlap with the rule ($\beta 2$) since a term of the form $(\Lambda a.t)\ \psi\ u$ can sometimes be reduced to both $\{\psi/a\}t\ u$ (by $\beta 2$) and $(\Lambda a.t)\ u$ (by $\xi 1$). Now, if t is λ-abstraction $\lambda x.s$ then both sides reduce to $\{u/x\}\{\psi/a\}s$ by ($\beta 1'$). However, if t and u are variables then $(\Lambda a.t)\ u$ is irreducible by the existing rules so that confluence would fail. That is, ($\beta 1'$) is just too coarse.

The solution is to add rules that act on terms of the form $(\Lambda a.t)\ u$ namely:

$$\begin{array}{lll}(\mu 1)\ (\Lambda a.t^{\forall \Delta.\sigma \rightarrow \phi})\ u^{\psi} \longrightarrow \{\psi/[a, \Delta]\sigma\}t\ u & & \text{if } a \in \mathsf{FTV}(\sigma)\backslash\Delta \\ & & \text{and } \mathsf{FTV}(t) \cap \Delta = \emptyset \\ (\mu 2)\ (\Lambda a.t^{\forall \Delta.\sigma \rightarrow \phi})\ u^{\psi} \longrightarrow \Lambda a.(t\ u) & & \text{if } a \notin \mathsf{FTV}(u) \cup \mathsf{FTV}(\sigma)\backslash\Delta\end{array}$$

Rule ($\mu 1$) is akin to ($\beta 2$), in that it substitutes for the type variable a in the body t. Unlike System **F**, however, the type to substitute is not explicit; instead, it is found by matching the type ψ of the value argument u against the argument type σ of t. This match yields the substitution $\{\psi/[a, \Delta]\sigma\}$, which will certainly bind a, but

[1] Technically we need $a \in \mathsf{FTV}(\sigma)\backslash\Delta$, since nothing prevents a appearing in Δ.

also may (and perhaps must) bind variables in Δ. The side condition $\mathsf{FTV}(t) \cap \Delta = \emptyset$, which can always be made true by α-conversion, ensures that the substitution does not accidentally instantiate an unrelated free type variable of t.

The rule ($\mu 2$) is more straightforward. It simply commutes the abstraction and application when they do not interfere. Note that the side condition $a \notin \mathsf{FTV}(u)$ can be made true by α-conversion. Now ($\beta 1'$) is a consequence of ($\beta 1$) and ($\mu 1$) and ($\mu 2$) so the full set of reduction rules is given by the β-rules of System **F** plus the four new rules in Figure 2.

3.3 Translation to System F

This subsection formalises the close relationship between System **IF** and System **F**.

Theorem 3. *The embedding of System* **F** *into System* **IF** *preserves typing, type erasure and reduction.*

Note that some normal forms of System **F** are reducible in System **IF**. For example, it may happen that a normal form $x\ \psi\ u$ of System **F** reduces to $x\ u$ by ($\xi 1$).

$$\begin{aligned}
\llbracket x^\sigma \rrbracket &= x^\sigma \\
\llbracket r^{\forall \Delta.\sigma \to \phi}\ u^\psi \rrbracket &= \Lambda(\Delta \backslash \sigma).\llbracket r \rrbracket\ \{\psi/[\Delta]\sigma\}\Delta\ \llbracket u \rrbracket \\
\llbracket \lambda x^\sigma.s \rrbracket &= \lambda x^\sigma.\llbracket s \rrbracket \\
\llbracket r\ \psi \rrbracket &= \llbracket r \rrbracket\ \psi \\
\llbracket \Lambda a.s \rrbracket &= \Lambda a.\llbracket s \rrbracket
\end{aligned}$$

Fig. 3. Translation from System **IF** to System **F**

This natural embedding is complemented by a translation in the opposite direction, from System **IF** into System **F**, that makes the implicit type applications explicit. The translation is shown in Figure 3. Although it is fairly well behaved, it does not preserve the reduction rule ($\xi 2$). Rather, the left-hand and right-hand sides of ($\xi 2$) translate repectively to

$$\Lambda a.\Lambda(\Delta \backslash \sigma).\llbracket r \rrbracket\ \psi\ \Delta'\ \llbracket u \rrbracket \qquad \text{and} \qquad (\Lambda a.\Lambda(\Delta \backslash \sigma).\llbracket r \rrbracket\ a\ \Delta'\ \llbracket u \rrbracket)\ \psi$$

for some Δ'. Here, the right-hand side reduces to the left-hand side by ($\beta 2$). In this case the permutation of the arguments has caused the direction of reduction to be reversed. However, although the translation does not preserve reduction, it does preserve equality, the symmetrical relation generated by the rules.

The same type erasure mechanism used for System **F** can be applied to terms of System **IF** to yield pure λ-terms.

Theorem 4. *There is a translation* $\llbracket - \rrbracket$ *from terms of System* **IF** *to those of System* **F** *that preserves typing, type erasure and equality of terms. Further, if* t *is a term in System* **F** *then* $\llbracket t \rrbracket$ *is* t *itself.*

Proof. That the translation in Figure 3 preserves type derivations follows by induction on the structure of the derivation. That it preserves type erasure is immediate. The preservation of terms in System **F** follows by induction on the structure of the term. The only non-trivial case concerns an application $r\ u$ but then the type of r cannot bind any type variables so that the translation becomes

$$[\![r^{\sigma\to\phi}\ u^{\sigma}]\!] = [\![r]\!]\ [\![u]\!].$$

Now consider equality. It is enough to show that when both sides of each reduction rule are translated then they are either equal, or have a common reduct. The rule ($\beta 1$) is preserved since $[\![(\lambda x^{\sigma}.s)\ u]\!] = (\lambda x^{\sigma}.[\![s]\!])\ [\![u]\!] \longrightarrow \{[\![u]\!]/x\}[\![s]\!]\} = [\![\{u/x\}s]\!]$ where the last equation is a trivial lemma. A similar argument applies to ($\beta 2$). Both sides of the rule ($\xi 1$) have the same translation. The argument for ($\xi 2$) is already given. The translation of ($\mu 1$) is given by the ($\beta 2$) reduction

$$\Lambda(\Delta\backslash\sigma).(\Lambda a.[\![t]\!])\ \theta a\ \ \theta\Delta\ [\![u]\!] \longrightarrow \{\theta a/a\}(\Lambda(\Delta\backslash\sigma).[\![t]\!]\ \theta\Delta\ [\![u]\!])$$

where θ is $\{\psi/[a,\Delta]\sigma\}$. The translation of ($\mu 2$) is given by the ($\beta 2$) reduction

$$\Lambda a.\Lambda(\Delta\backslash\sigma).(\Lambda a.[\![t]\!])\ a\ \ \theta\Delta\ [\![u]\!] \longrightarrow \Lambda a.\Lambda(\Delta\backslash\sigma).[\![t]\!]\ \theta\Delta\ [\![u]\!]$$

where θ is $\{\psi/[a,\Delta]\sigma\}$ which is also $\{\psi/[\Delta]\sigma\}$ since $a \notin \mathsf{FTV}(\sigma)$.

This completes the proof that the translation preserves the equality generated by rewriting. □

3.4 Properties of System IF

System **IF** is a little more complicated than System **F**, but they share many of the same good properties, including being strongly normalising and confluent. Notably, it shares the unique-type property described in Section 2.2, as can be seen by inspection of the typing rules.

Lemma 1 (Substitution Lemma).

1. *If there is a derivation of $t : \sigma$ and θ is a type substitution then there is a derivation of $\theta t : \theta\sigma$.*
2. *If $s : \phi$ is a term and x^{σ} is a variable that may be free in s and $u : \sigma$ is a term then there is a derivation of $\{u/x\}s : \phi$.*

Proof. The proofs are by straightforward induction on the structure of the terms. □

Theorem 5. *Reduction in System* **IF** *preserves typing.*

Proof. The proof is by induction on the structure of the reduction. Without loss of generality, the reduction is a rule. If it is either ($\beta 1$) or ($\beta 2$) then apply the Substitution Lemma.

If the rule is ($\xi 1$) with $u : \tau$ then $r\ \psi : \{\psi/a\}(\forall\Delta.\sigma \to \phi)$ and so $r\ \psi\ u : \forall(\Delta\backslash\sigma).\{\tau/[\Delta]\{\psi/a\}\sigma\}\{\psi/a\}\phi$ while

$$r\ u : \forall(a,\Delta\backslash\sigma).\{\tau/[a,\Delta]\sigma\}\phi.$$

Now $a \in \mathsf{FTV}(\sigma)$ implies that $\{\tau/[a,\Delta]\sigma\} = \{\tau/[\Delta]\{\psi/a\}\sigma\} \circ \{\psi/a\}$ (where $\circ$ denotes composition of substitutions). This yields the result.

If the rule is ($\xi 2$) with $u : \tau$ then $r\ \psi : \{\psi/a\}(\forall\Delta.\sigma \to \phi)$ whose type is also $\forall\Delta.\sigma \to \{\psi/a\}\phi)$ since a is not free in σ. Hence $r\ \psi\ u$ has type $\forall(\Delta\backslash\sigma).\{\tau/[\Delta]\sigma\}\{\psi/a\}\phi$ which is the type $\forall(\Delta\backslash\sigma).\{\psi/a\}\{\tau/[\Delta]\sigma\}\phi$ of $r\ u\ \psi$.

If the rule is ($\mu 1$) then the left-hand side has type $\forall(\Delta\backslash\sigma).\upsilon\phi$ where $\upsilon = \{\psi/[a,\Delta]\sigma\}$. Also, the right-hand side has type

$$\forall(\Delta\backslash\sigma).\{\psi/[\Delta]\{\upsilon a/a\}\sigma\}\{\upsilon a/a\}\phi$$

which is the same since $\upsilon = \{\psi/[\Delta]\{\upsilon a/a\}\sigma\} \circ \{\upsilon a/a\}$.

If the rule is ($\mu 2$) then the left-hand side has type $\forall a.\forall(\Delta\backslash\sigma).\upsilon\phi$ where $\upsilon = \{\psi/[a,\Delta]\sigma\}$ is also $\{\psi/[\Delta]\sigma\}$ since a is not free in σ. Hence, the type is that of the right-hand side, too. □

Theorem 6. *Type erasure maps ($\beta 1$) to β-reduction of the pure λ-calculus and maps all other rules to equations.*

Proof. The proof is immediate. □

Theorem 7. *Reduction in System* **IF** *is strongly normalising.*

Proof. Observe that if t is a term in System **F** then its type erasure is strongly normalising in the pure λ-calculus, since any reduction of the erasure is the image of some non-empty reduction sequence in System **F**. Since the translation from System **IF** to System **F** preserves the type erasure and ($\beta 1$) this property extends to all of System **IF**. Thus any reduction sequence in System **IF** contains finitely many instances of ($\beta 1$). Hence, to prove strong normalisation, it suffices to consider reduction sequences without any uses of ($\beta 1$). The remaining reduction rules all reduce the rank ρ of the terms, as defined by

$$\begin{aligned} \rho(x^\sigma) &= 0 \\ \rho(r\ u) &= 2\rho(r) + \rho u \\ \rho(r\ U) &= \rho(r) + 1 \\ \rho(\Lambda a.s) &= 2\rho(s) + 1. \end{aligned}$$

For example, $\rho(r\ \psi\ u) = 2(\rho(r) + 1) + \rho(u) > 2\rho(r) + \rho(u) + 1 = \rho(r\ u\ \psi)$ and $\rho((\Lambda a.s)\ u) = 2(\rho(s)+1) + \rho(u) > 2\rho(s) + \rho(u) + 1 = \rho(\Lambda a.s\ u)$. The other three reduction rules are easily checked. □

Theorem 8. *Reduction in System* **IF** *is Church-Rosser.*

Proof. Since reduction is strongly normalising, it is enough to prove that every critical pair can be resolved. Those which are already present in System **F** are resolved using its Church-Rosser property. The new reduction rules cannot overlap with ($\beta 1$) for typing reasons. Nor can they overlap with each other. Hence the only remaining critical pairs involve ($\beta 2$) and ($\xi 1$) or ($\xi 2$).

For ($\xi 1$), a term of the form $(\Lambda a.s)\ \psi_1\ u^\psi$ rewrites to both $\{\psi_1/a\}s\ u$ and $(\Lambda a.s)\ u$. The latter term further reduces by ($\mu 1$) to $\theta s\ u$ where θ is the restriction of $\{\psi/[a,\Delta]\sigma\}$ to a. Now this must map a to ψ_1 since a is free in

σ and $\{\psi/[\Delta]\{\psi_1/a\}\sigma\}$ exists (from the typing of the original term). Hence $\{\psi_1/[a,\Delta]\sigma\}s\ u$ is exactly $\{\psi/a\}s\ u$.

For (ξ2) a term of the form $(\Lambda a.s)\ \psi_1\ u$ rewrites to both $\{\psi_1/a\}s\ u$ and $(\Lambda a.s)\ u\ \psi_1$. The latter term further reduces to $(\Lambda a.s\ u)\ \psi_1$ and thence to the former term. □

3.5 Extension to Higher Kinds

The side conditions for the novel reduction rules constrain the appearance of bound type variables in quantified types, and the matching process inspects their syntactic structure. Type safety requires that these properties be stable under substitution.

In System **IF**, like System **F**, this is automatically true. But what if one were to add functions at the type level, as is the case in **F**ω, for example? Then, b is free in the type $(a\ b)$, but if we were to substitute $\lambda x.\texttt{Int}$ for a, then $(a\ b)$ would reduce to `Int` in which b is no longer free. Similarly, computing the match $\{\texttt{List Int}/[b](a\ b)\}$, as defined in Figure 2, would yield the substitution $\{\texttt{Int}/b\}$; but if we were to substitute $\lambda x.x$ for a, the match would become $\{\texttt{List Int}/[b]b\}$, which yields quite a different result.

None of these problems arise, however, in the fragment of **F**ω that is used by Haskell (Peyton Jones 2003). Haskell permits higher-kinded constants (introduced by data type declarations), but has no type-level lambda, and hence no reductions at the type level. In this setting, simple first-order matching suffices despite the use of higher kinds. For example, adding `List` as a constant of kind $\star \rightarrow \star$ (with no reduction rules) causes no difficulty; indeed, this same property is crucial for type inference in the Haskell source language.

In short, System **IF** as presented above can readily be extended with type constants and type variables of higher kind, to serve as an intermediate language for Haskell. The syntax of types would then become

$$\begin{aligned} S, T &::= \langle \text{type constructors} \rangle \\ \sigma, \tau, \phi, \psi &::= a \mid \sigma \rightarrow \sigma \mid \forall a.\sigma \mid T \mid \sigma\ \phi \end{aligned}$$

(note no lambda), together with kinding rules to ensure that types are well-kinded. We leave for future work the question of whether and how **IF** can be further extended to accommodate full type-level lambda.

3.6 Eta-Rules

In pure λ-calculus, the (η)-equality

$$\lambda x.r\ x = r$$

reflects the intuition that everything is a function. It is of interest here for two reasons. One is to tighten the connection between System **IF** and System **F**. The other is to support its use in compiler optimisations.

Traditionally, (η) it has been expressed as a contraction (reducing $\lambda x.r\ x$ to r), which is appropriate in practice. Note, however, that for many purposes it is better thought of as an expansion (Jay and Ghani 1995; Ghani 1997). In System **F** the η-contraction rules are

$$\begin{array}{l} (\eta 1)\ \lambda x.r\ x \longrightarrow r \quad \text{if } x \notin \mathsf{ftv}(r) \\ (\eta 2)\ \Lambda a.r\ a \longrightarrow r \text{ if } a \notin \mathsf{FTV}(r). \end{array}$$

In System **IF**, the rule $(\eta 1)$ must be made more general, to reflect the implicit action on type variables. Given a term $r : \forall \Delta.\sigma \rightarrow \phi$ and a variable $x : \sigma$ then $r\ x$ will instantiate the type variables in $\Delta \cap \mathsf{FTV}(\sigma)$ while leaving the rest bound. Hence $r\ x\ (\Delta \backslash \sigma) : \phi$ and so

$$\lambda x^{\sigma}.r\ x\ (\Delta \backslash \sigma) : \sigma \rightarrow \phi$$

has the same type as $r\ \Delta$. In this way, the rules become:

$$\begin{array}{lrl} (\eta 1')\ \lambda x^{\sigma}.r^{\forall \Delta.\sigma \rightarrow \phi}\ x\ (\Delta \backslash \sigma) & \longrightarrow r\ \Delta & \text{if } x \notin \mathsf{ftv}(r) \\ (\eta 2) \qquad\qquad \Lambda a.r\ a & \longrightarrow r & \text{if } a \notin \mathsf{FTV}(r). \end{array}$$

Of course, if the redex of $(\eta 1')$ is well-typed in System **F** then Δ is empty and the standard rule emerges. When these η-contractions are added, the resulting systems are called System **F**η and System **IF**η respectively.

The rule $(\eta 1')$ is unlike all previous rules considered, in that it is not stable under substitution for type variables in Δ. If θ is a type substitution then the reduction relation defined by the rules must be broadened to include

$$\lambda x^{\theta\sigma}.r^{\forall \Delta.\sigma \rightarrow \phi}\ x\ \theta(\Delta \backslash \sigma) \longrightarrow r\ (\theta\Delta) \text{ if } x \notin \mathsf{ftv}(r).$$

To apply the rule in this form, it suffices to discover θ by type matching of σ against the given type of x etc.

The good properties established in the last two sub-sections continue to hold in the presence of the η-rules.

Theorem 9. *The translation $[\![-]\!]$ from terms of System* **IF** *to those of System* **F** *maps $(\eta 2)$ to $(\eta 2)$ and maps $(\eta 1')$ to a sequence of reductions using $(\eta 2)$ followed by $(\eta 1)$. Further, for each term t in System* **IF**η*, its translation $[\![t]\!]$ reduces to t in System* **IF**η*.*

Proof. The translation of the redex of $(\eta 1')$ is

$$\lambda x^{\sigma}.(\Lambda(\Delta \backslash \sigma).[\![r]\!]\ \Delta\ x)\ (\Delta \backslash \sigma)$$

which reduces by $(\eta 2)$ to $\lambda x.[\![r]\!]\ \Delta\ x$ and then to $[\![r]\!]\ \Delta$ by $(\eta 1)$. The translation of $(\eta 2)$ is $(\eta 2)$.

The proof that $[\![t]\!] \longrightarrow t$ is by induction on the structure of t. The only non-trivial case is an application $r\ u$. To its translation apply the ξ-rules to get a term of the form $\Lambda(\Delta \backslash \sigma).[\![r]\!]\ [\![u]\!]\ (\Delta \backslash \sigma)$ which reduces by $(\eta 2)$ to $[\![r]\!]\ [\![u]\!]$. In turn, this reduces to $r\ u$ by induction.

Theorem 10. *Reduction in System* $\mathbf{IF}\eta$ *is strongly normalising.*

Proof. The proof is as for System **IF**, but now relying on the translation to System $\mathbf{F}\eta$.

Theorem 11. *Reduction in System* $\mathbf{IF}\eta$ *is Church-Rosser.*

Proof. It suffices to check the new critical pairs that arise from interaction with the η-rules. The critical pair involving $(\beta 2)$ and $(\eta 2)$ is resolved as in **F**. The other critical pairs are of $(\eta 1')$ with $(\beta 1), (\xi 1), (xi2), (\mu 1)$ and $(\mu 2)$. Let us consider them in turn, in relation to the rule $(\eta 1')$.

If r is $\lambda x.s$ then Δ is empty and $\lambda x.(\lambda x.s)\ x$ reduces by $(\eta 1')$ and $(\beta 1)$ to $\lambda x.s$.

If r is $t\ a$ for some $t : \forall a.\forall \Delta.\sigma \rightarrow \phi$ then $\lambda x.t\ a\ x\ (\Delta\backslash\sigma)$ reduces to $t\ a\ \Delta$ by $(\eta 1')$. The result of applying either $(\xi 1)$ or $(\xi 2)$ to the original term can be further reduced by $(\eta 1')$ to $t\ a\ \Delta$. More generally, if r is some $t\ \psi$ then the applications of $(\eta 1')$ must have the substitution of ψ for a applied to them.

If r is $\Lambda a.s$ then $(\eta 1')$ reduces this to $(\Lambda a.s)\ a\ (\Delta\backslash\sigma)$. This in turn reduces to $s\ (\Delta\backslash\sigma)$. If $a \in \mathsf{FTV}(\sigma)\backslash\Delta$ then this is also the result of applying first $(\mu 1)$ and then $(\eta 1')$. If $a \notin \mathsf{FTV}(\sigma)\backslash\Delta$ then this is also the result of applying first $(\mu 2)$ and then $(\beta 2)$ and $(\eta 1')$.

4 Practical Implications

Apart from its interest as a calculus in its own right, we were interested in applying System **IF** in a real compiler. We therefore modified the Glasgow Haskell Compiler (GHC), a state-of-the-art compiler for Haskell, to use System **IF** as its intermediate language. This process turned out (as hoped) to be largely straightforward: the changes were highly localised, and only a few hundred lines of code in a 100,000-line compiler had to be modified in a non-trivial way.

GHC already uses a mild extension of System **F** as its intermediate language: at the type level it admits generalised algebraic data types (GADTs) and higher-kinded type variables; while in terms it allows recursive `let` bindings, data constructors (and applications thereof), and `case` expressions to decompose constructor applications. We extended System **IF** in an exactly analogous way, a process which presented no difficulties, although we do not formalise it here.

While GHC maintains full type information throughout its optimisation phases, it performs type erasure just before code generation, so types have no influence at run-time. The effect of using System **IF** instead of **F** is therefore confined to compile time.

4.1 Variable Bindings and Occurrences

In practical terms, the reader may wonder about the space implications of attaching a type to every variable *occurrence*, rather than attaching the variable's type only to its *binding* (Section 2.1). In fact, GHC already does exactly this, reducing the additional space costs (compared to annotating only the binding sites) to zero by sharing. The data type that GHC uses to represent terms looks like this:

```
data CoreExpr = Var Id | Lam Id CoreExpr | ...
data Id = MkId Name Type
```

An `Id` is a pair of a `Name` (roughly, a string), and its `Type`, so it represents the formal notation x^σ. An `Id` is used both at the *binding site* of the variable (constructor `Lam`) and at its *occurrences* (`Var`). GHC is careful to ensure that every occurrence shares the same `Id` node that is used at the binding site; in effect, all the occurrences point to the binder. This invariant is maintained by optimisation passes, and by operations such as substitution.

As a consequence, like System **F** itself, every term in GHC's intermediate language has a unique type, independent of its context (Section 2.2). That is, we can (and GHC does) provide a function `exprType` that extracts the type of any term:

```
exprType :: CoreExpr -> Type
```

Notice that `exprType` does not require an environment; it can synthesise the type of a term by inspecting the term alone, in a single, bottom-up traversal of the term. (Here, we assume that the term is well-typed; `exprType` is not a type-checking algorithm.) This function `exprType` makes concrete the idea that every term has a unique type (Section 2.1), and it is extremely valuable inside GHC, which is why maintaining the unique-type property was a key requirement of our design.

4.2 Optimisation and Transformation

One might wonder whether, in changing from System **F** to **IF**, we had to rewrite every transformation or optimisation in the compiler. Happily, we did not. Almost all transformations now rewrite **IF** to **IF** without change, and certainly without introducing and re-eliminating the omitted type applications.

The non-trivial changes were as follows:

- The new typing rule (ifapp) amounts to extending the `exprType` function, and Core Lint. The latter is a type checker for GHC's intermediate language, used only as a consistency check, to ensure that the optimised program is still well typed. If the optimisations are correct, the check will always succeed, but it is extremely effective at finding bugs in optimisations.

  It turned out to be worth implementing the type-matching function (Section 3.1) twice. For Core Lint the full matching algorithm is required, but `exprType` operates *under the assumption that the term is well typed*. In the latter case several short-cuts are available: there is no need to check that matching binds a variable consistently at all its occurrences, kind-checks (usually necessary in GHC's higher-kinded setting) can be omitted, and matching can stop as soon as all the quantified variables have been bound.

- The new reduction rules ($\xi 1$), ($\xi 2$), ($\mu 1$), ($\mu 2$), ($\eta 1'$) all appear as new optimising transformations. GHC's optimiser goes to considerable lengths to ensure that transformations "cascade" well, so that many transformations can be appplied successively in a single, efficient pass over the program. Adding the new rules while retaining this property was trickier than we expected.
- Eta-reduction, one of GHC's existing transformations, becomes somewhat more complicated (Section 3.6).
- There is one place that that we *do* reconstruct the omitted type arguments, namely when simplifying a `case` that scrutinises a constructor application, where the constructor captures existential variables. For example, consider the following data type:

  ```
  data T where { MkT :: forall a. a -> (a->Int) -> T }
  ```

  Now suppose we want to simplify the term

  ```
      case (MkT 'c' ord) of
        MkT a (x:a) (f:a->Int) -> ...
  ```

  (Here `ord` has type `Char->Int`.) Then we must figure out that the existential type variable `a`, bound by the pattern, should be instantiated to `Char`, which in turn means that we must re-discover the omitted type argument of the `MkT` constructor.

Module	GHC Core		System **IF**		Reduction(%)	
	Size	Type	Size	Type	Size	Type
		apps		apps		apps
Data.Tuple	169,641	10,274	131,349	0	23%	100%
Data.Generics.Instances	36,038	1,774	32,488	578	10%	67%
Data.Array.Base	32,068	2,498	26,468	397	17%	84%
Data.Sequence	29,468	2,124	24,532	354	17%	83%
Data.Map	21,217	1,566	17,958	334	15%	79%
Data.Array.Diff	16,286	895	14,067	73	14%	92%
GHC.Float	16,100	414	15,353	50	5%	88%
Data.IntMap	14,614	1,025	12,363	209	15%	80%
System.Time	14,499	914	12,934	338	11%	63%
GHC.Read	14,210	959	11,110	141	22%	85%
GHC.Real	14,094	355	13,400	54	5%	85%
GHC.IOBase	13,698	711	11,494	212	16%	70%
GHC.Handle	13,504	902	11,938	192	12%	79%
GHC.Arr	13,455	837	11,490	49	15%	94%
Data.ByteString	12,732	1,104	10,632	230	16%	79%
Foreign.C.Types	12,633	359	11,987	48	5%	87%
. . . and 114 other modules . . .						
TOTAL	792,295	46,735	676,615	7,257	15%	84%
TOTAL (omitting Data.Tuple)	622,654	36,461	545,266	7,257	12%	80%

Fig. 4. Effect of using System **IF**

4.3 Code Size

We compiled all 130 modules of the `base` library packages, consisting of some 110,000 lines of code. Each module was compiled to GHC's Core language, which is a variant of System **F**. We wrote a special pass to eliminate redundant type applications by applying rules ($\xi 1$) and ($\xi 2$) exhaustively, and measured the size of the program before and after this transformation. This "size" counts the number of nodes in the syntax tree of the program including the sizes of types, except the types at variable occurrences since they are always shared. Apart from the sharing of a variable's binding and its occurrences we do not assume any other sharing of types (see Section 5.2). We also counted the number of type applications before and after the transformation.

The results are shown in Figure 4, for the largest 16 modules, although the total is taken over all 130 modules. The total size of all these modules taken together was reduced by around 15%, eliminating nearly 84% of all type applications. These figures are slightly skewed by one module, `Data.Tuple`, which consists entirely of code to manipulate tuples of various sizes, and which is both very large and very uncharacteristic (all of its type applications are removed). Hence, the table also gives the totals excluding that module; the figures are still very promising, with 80% of type applications removed.

Manual inspection shows that the remaining type applications consist almost exclusively of

- Data constructors where the arguments do not fully specify the result type (such as `Nil`).
- Calls to Haskell's `error` function, whose type is

$$\texttt{error} : \forall a.\texttt{String} \rightarrow a$$

There are a handful of other places where type applications are retained. For example, given `map` and `reverse` with their usual types, and `xs` : `List Int`, then in the term

```
(map (reverse Int) xs)
```

the type application `(reverse Int)` cannot be eliminated by our rules. In practice, however, data constructors and error calls dominate, and that is fair enough, because the type applications really are necessary if every term is to have unique, synthesisable type.

4.4 Compilation Time

The proof of the pudding is in the eating. System **IF** has smaller terms, but its reduction rules are more complicated, and computing the type of a term involves matching prior to instantiation. Furthermore, although fewer types appear explicitly in terms, some of these omitted types might in practice be constructed on-the-fly during compilation, so it is not clear whether the space saving will translate into a time saving.

We hoped to see a consistent improvement in the execution time of the compiler, but the results so far are disappointing. We measured the total number of bytes allocated by the compiler (a repeatable proxy for compiler run-time) when compiling the same 130 modules as Section 4.3. Overall, allocation decreased by a mere 0.1%. The largest reduction was 4%, and the largest increase was 12%, but 120 of the 130 modules showed a change of less than 1%. Presumably, the reduction in work that arises from smaller types is balanced by the additional overheads of System **IF**.

On this evidence, the additional complexity introduced by the new reduction rules does not pay its way. Nevertheless, these are matters that are dominated by nitty-gritty representation details, and the balance might well be different in another compiler.

5 Related Work

5.1 Type Inference

In source languages with type inference, such as ML or Haskell, programmers *never* write type applications — instead, the type inference system infers them. The classic example of such a system is the Hindley-Milner type system (Milner 1978), which has been hugely influential; but there are many other type inference systems that allow a richer class of programs than Hindley-Milner. For example, Pierce and Turner's Local Type Inference combines type information from arguments to choose the type instantiation for a function, in the context of a language with subtyping (Pierce and Turner 1998). Their paper also introduced the idea of *bidirectional* type inference, which Peyton Jones *et al* subsequently applied in the context of Haskell to improve type inference for higher-rank types (Peyton Jones et al. 2007).

Stretching the envelope even further, Le Botlan and Rémy's ML^F language supports *impredicative* polymorphism, in which a polymorphic function can be called at a polytype (Le Botlan and Rémy 2003). Pfenning goes further still, describing the problem of *partial type inference* for full $F\omega$, including lambda at the type level (Pfenning 1988). Type abstractions ($\Lambda a.t$) are retained, but type annotations may be omitted from term abstractions (thus $\lambda x.t$ rather than $\lambda x^{\sigma}.t$), and type applications may be abbreviated to mere placeholders (thus $t\ [\,]$ instead of $t\ \sigma$). However, type inference in this system requires higher-order unification, and it lacks the unique-type property. Even more general use of such placeholders can be found in dependent type systems, which may be used to compress the size of proof-carrying code (Necula and Lee 1998) or to support dependently-typed programming languages (Norell 2007).

Such inference engines are invariably designed for *source* languages, and are less well suited for use as a calculus, or as a compiler intermediate language.

– Most type inference systems are less expressive than System **F**. For example, in Hindley-Milner, function arguments always have a monomorphic type; in many other systems types must be of limited rank, and the system is usually predicative.

This lack of expressiveness matters, because the compiler may use a richer type system internally than is directly exposed to the programmer. Examples include closure conversion (Minamide et al. 1996), and the dictionary-passing translation for type classes (Wadler and Blott 1989).

- Type inference can be regarded as a constraint-solving problem, where the constraints are gathered non-locally from the program text, and the solver may be somewhat costly to run. In contrast, in a compiler intermediate language one needs to answer the question "what is the type of this sub-term?", and to do so cheaply, and using locally-available information. For this purpose, the unique-type property of Section 2.2 is extremely helpful, but it is rare indeed for a system based on type inference to possess it.
- The restrictions that allow type inference are never fully robust to program transformation, and (in every case except the least expressive, Hindley-Milner) require *ad hoc* type annotations. For example, even if t is well-typed in a particular environment, $(\lambda f.t)\ f$ may not be, because, say, f's type may not be a monotype. Compilers perform inlining and abstraction all the time. Another way to make the same point is this: type inference systems usually treat a *fixed* source program text; they are never thought of as a *calculus* equipped with a type-preserving reduction semantics.

In short, type inference systems focus on programmers, whereas our focus is on compiler writers and logicians. Nevertheless, to the extent that one might regard System **F** as a language for programmers, we believe that **IF** should serve the same role as well or better.

5.2 Reducing the Cost of Types

A handful of papers address the question of the overheads of type information in type-preserving compilers. The most popular approach is to exploit common sub-structure by *sharing* common types or parts of types. These techniques come in two varieties: ones that *implicitly* share the representation of terms, and ones that express sharing *explicitly* in the syntax of terms.

For example, Petersen (Petersen 2005, Section 9.6) and Murphy (Murphy 2002) describe several approaches to type compression in the TILT compiler. Most of these are representation techniques, such as hash-consing and de-Bruijn representation, that can be used to implement type operations more efficiently. Shao *et al* devote a whole paper to the same subject, in the context of their FLINT compiler (Shao et al. 1998). Their techniques are exclusively of the implementation variety: hash-consing, memoisation, and advanced lambda encoding.

Working at the representation level is tricky, and seems to be sensitive to the context. For example Murphy reports a slow-down from using hash-consing, whereas Shao *et al* report a significant speed-up, and Petersen found that without hash-consing type-checking even a small program exhausted the memory on a 1Gbyte machine. (These differences are almost certainly due to the other techniques deployed at the same time, as we mention shortly.) Another reason that representation-level sharing is tricky is that it is not enough for two types to share memory; the sharing must also be *observable.* Consider, say, substitution. Unless sharing is observable, the substitution will happen once for each identical copy, and the results will be laboriously re-hash-consed together. Memory may be saved, but time is not.

A complementary approach, and the one that we discuss in this paper, is to change the intermediate language itself to represent programs more efficiently. For example, TILT has a `lettype` construct which provides for explicit sharing in type expressions. For example, using `lettype` we could write the nested `Pair` example from Section 2.3 like this:

$$\begin{array}{ll}
\texttt{lettype} & a_1 = \sigma_1; \ldots ; a_5 = \sigma_5 \\
 & b_1 = (a_1, a_2) \\
 & b_2 = (a_3, a_4) \\
 & b_3 = (b_2, a_5) \\
\texttt{in Pair} & b_1 \; b_3 \\
 & (\texttt{Pair} \; a_1 \; a_2 \; e_1 \; e_2) \\
 & (\texttt{Pair} \; b_2 \; a_5 \; (\texttt{Pair} \; a_3 \; a_4 \; e_3 \; e_4) \; e_5).
\end{array}$$

Such an approach carries its own costs, notably that type equivalence is modulo the environment of `lettype` bindings, but since TILT has a very rich notion of type equivalence anyway including full β-reduction in types, the extra pain is minimal. The gain appears to be substantial: FLINT does not have `lettype`, and Murphy suggests that this may be the reason that FLINT gets a bigger relative gain from hash-consing — `lettype` has already embodied that gain in TILT by construction.

Another example of changing the intermediate language to reduce type information is Chilpala *et al*'s work on strict bidirectional type checking (Chilpala et al. 2005). Their main goal is to drop the type annotation from a binder, for example writing $\lambda x.t$ instead of $\lambda x^\sigma.t$. It is possible to do this when the occurrence(s) of x completely fix its type. The paper only describes a simply-typed language, whereas our focus is exclusively on the type applications that arise in a polymorphic language. Furthermore, our approach relies on every term having a unique type, whereas theirs relies on inferring the unique types of the free variables of a term, starting from the type of the term itself. It remains to be seen whether the two can be combined, or which is more fruitful in practice.

Our focus is exclusively on reducing the cost of *compile-time* type manipulation. Other related work focuses on the cost of *run-time* type manipulation, for systems that (unlike GHC) do run-time type passing (Tolmach 1994; Saha and Shao 1998).

5.3 Pattern Calculus

The approach to type quantification developed in this paper was discovered while trying to type pattern-matching functions (Jay and Kesner 2006) in which each case may have a different (but compatible) polymorphic type (Jay 2006). For example, consider a function $\texttt{toString} : \forall a.a \rightarrow \texttt{String}$ which is to have special cases for integers, floats, pairs, lists, etc. A natural approach is to define special cases for each type setting, as follows:

$$\begin{aligned}\texttt{toStringInt} &: \texttt{Int} \rightarrow \texttt{String}\\ \texttt{toStringFloat} &: \texttt{Float} \rightarrow \texttt{String}\\ \texttt{toStringPair} &: \forall b, c.(b, c) \rightarrow \texttt{String}\\ \texttt{toStringList} &: \forall a.\texttt{List}\ a \rightarrow \texttt{String}\end{aligned}$$

and then try to combine them into a single function. Then the application `toString` (`List` a) could reduce to `toStringList` a and `toString` (b, c) could reduce to `toStringPair` $b\ c$ etc. in a form of `typecase`. However, this makes types central to reduction so that they cannot be erased. By making type applications implicit, it is possible to let the choice of special case be determined by the structure of the function argument instead of the type, as in

$$\begin{aligned}\texttt{toString}\ 3 &\longrightarrow \texttt{toStringInt}\ 3\\ \texttt{toString}\ 4.4 &\longrightarrow \texttt{toStringFloat}\ 4.4\\ \texttt{toString}\ (\texttt{Pair}\ x\ y) &\longrightarrow \texttt{toStringPair}\ (\texttt{Pair}\ x\ y)\\ \texttt{toString}\ \texttt{Nil} &\longrightarrow \texttt{toStringList}\ \texttt{Nil}.\end{aligned}$$

The proper development of this approach is beyond the scope of this paper; the point here is that implicit type quantification is the natural underpinning for this approach. Note, too, that there are natural parallels with object-orientation, where the object determines how to specialise the methods it invokes.

6 Further Work

The focus of this paper is on removing redundant type applications but type applications are not the only source of redundant type information. For example, as (Chilpala et al. 2005) point out, non-recursive `let` expressions are both common (especially in A-normalised code) and highly redundant; in the expression (`let` x^σ=r `in` t) the type annotation on x is redundant since it can be synthesised from r. A similar point can be made for `case` expressions; for example, consider the expression

$$\lambda x^\sigma.\,\texttt{case}\ x\ \texttt{of}\ (p^\phi, q^\psi)\texttt{->}\ t$$

Here, the type annotations on p and q are redundant, since they can be synthesised from the type of x.

One approach is to follow (Chilpala et al. 2005) by dropping these readily-synthesisable types at the language level. But nothing is gained from dropping type annotations on binders unless we also drop the annotations on *occurrences*, and that in turn loses the highly-desirable property that every term has a synthesisable type. An alternative, and perhaps more promising, approach is to work at the representation level, by regarding the type on such a binder simply as a cached or memoised call to `exprType`. In this way, if the type of the right-hand side of a non-recursive `let` binding was very large, the chances are that much sub-structure of that large type would be shared with variables free in that term. We have no data to back up these speculations, but it would be interesting to try.

So far we have assumed that the back end of the compiler performs type erasure, so that there is no run-time type passing. However, suppose one wants run-time type passing, to support `typecase` or reflection. It is immediately obvious how to compile System **F** to machine code, and still support run-time type passing — just make all type arguments into value arguments — but matters are not so obvious for System **IF**. This is an area for future work.

System **IF** contains a mixture of implicit and explicit type applications. This is extremely useful for backwards compatibility with System **F** but the resulting calculus is hardly minimal. Hence, there are a number of other possibilities for handling type applications. In particular, one can insist that all such are implicit, in a calculus of *quantified function types* (Jay 2006) whose types are given by

$$\sigma ::= a \mid [\Delta]\sigma \to \sigma$$

where the well-formed quantified function types $[\Delta]\sigma \to \phi$ play the same role as the type $\forall\Delta.\sigma \to \phi$ in System **F** but come with a guarantee that all type applications can be made implicit.

7 Conclusions

The formal beauty of System **F** together with the practical success of the Hindley-Milner type system have combined to set a high standard for anyone attempting to improve in either direction. For studying the concept of parametric polymorphism System **F** appears ideal, despite its redundancies. For avoiding types, the Hindley-Milner system is spectacularly successful. The key observation of this paper is that one can eliminate much of the redundant type information in System **F** without sacrificing any expressive power or basic properties. The price of this approach is the need to add some redundancy to the reduction rules, so that there are several ways to reduce a type application.

System **IF** should be of interest as a calculus in its own right. On the practical side, we hoped that System **IF** would allow us to reduce the size of program terms in a type-preserving compiler, and thereby reduce compilation time. In the particular context of the Glasgow Haskell Compiler we successfully demonstrated the former, but not the latter. The balance of costs and benefits might, however, be different in other settings.

As indicated in the related work, there is a rich design space of highly-expressive calculi in which some type information is implicit or abbreviated. Among these, System **IF** appears to be the only one that shares System **F**'s desirable unique-type property. Whether it is possible to elide yet more type information without losing this property remains an open question.

Acknowledgements

We thank Bob Harper, Jeremy Gibbons, Thomas Given-Wilson, Shin-Cheng Mu, Tony Nguyen, Leaf Petersen, Didier Rémy, and the anonymous referees of an earlier submission, for their helpful feedback on drafts of the paper.

References

Chilpala, A., Petersen, L., Harper, R.: Strict bidirectional type checking. In: ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI 2005), Long Beach. ACM Press, New York (2005)

Coq. Coq (2007), `http://pauillac.inria.fr/coq/`

Ghani, N.: Eta-expansions in dependent type theory – the calculus of constructions. In: de Groote, P., Hindley, J.R. (eds.) TLCA 1997. LNCS, vol. 1210, pp. 164–180. Springer, Heidelberg (1997)

Girard, J.-Y.: The System F of variable types: fifteen years later. In: Huet, G. (ed.) Logical Foundations of Functional Programming. Addison-Wesley, Reading (1990)

Girard, J.-Y., Lafont, Y., Taylor, P.: Proofs and Types. Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge (1989)

Harper, R., Morrisett, G.: Compiling polymorphism using intensional type analysis. In: Conference Record of POPL 1995: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, pp. 130–141 (January 1995)

Hinze, R., Jones, S.P.: Derivable type classes. In: Hutton, G. (ed.) Proceedings of the 2000 Haskell Workshop, Montreal (September 2000); Nottingham University Department of Computer Science Technical Report NOTTCS-TR-00-1

Jay, B., Kesner, D.: Pure Pattern Calculus. In: Sestoft, P. (ed.) ESOP 2006 and ETAPS 2006. LNCS, vol. 3924, pp. 100–114. Springer, Heidelberg (2006), `www-staff.it.uts.edu.au/~cbj/Publications/purepatterns.pdf`

Jay, C.B.: Typing first-class patterns. In: Higher-Order Rewriting, electronic proceedings (2006), `http://hor.pps.jussieu.fr/06/proc/jay1.pdf`

Jay, C.B., Ghani, N.: The virtues of eta-expansion. J. of Functional Programming 5(2), 135–154 (1995); Also appeared as tech. report ECS-LFCS-92-243

Le Botlan, D., Rémy, D.: MLF: raising ML to the power of System F. In: ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden, pp. 27–38. ACM Press, New York (2003)

Milner, R.: A theory of type polymorphism in programming. JCSS 13(3) (December 1978)

Minamide, Y., Morrisett, G., Harper, R.: Typed closure conversion. In: 23rd ACM Symposium on Principles of Programming Languages (POPL 1996), St Petersburg Beach, Florida, pp. 271–283. ACM Press, New York (1996)

Murphy, T.: The wizard of TILT: efficent, convenient, and abstract type representations. Technical Report CMU-CS-02-120, Carnegie Mellon University (2002), `http://reports-archive.adm.cs.cmu.edu/anon/2002/CMU-CS-02-120.pdf`

Necula, G.C., Lee, P.: Efficient representation and validation of proofs. In: Logic in Computer Science, pp. 93–104 (1998), `citeseer.ist.psu.edu/article/necula98efficient.html`

Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden (2007), `http://www.cs.chalmers.se/~ulfn/papers/thesis.pdf`

Petersen, L.: Certifying Compilation for Standard ML in a Type Analysis Framework. PhD thesis, Carnegie Mellon University (2005), `http://reports-archive.adm.cs.cmu.edu/anon/2005/CMU-CS-05-135.pdf`

Jones, S.P.: Haskell 98 language and libraries: the revised report. Cambridge University Press, Cambridge (2003)

Jones, S.P., Hall, C., Hammond, K., Partain, W., Wadler, P.: The Glasgow Haskell Compiler: a technical overview. In: Proceedings of Joint Framework for Information Technology Technical Conference, Keele, DTI/SERC, pp. 249–257 (March 1993), `http://research.microsoft.com/~simonpj/Papers/grasp-jfit.ps.Z`

Jones, S.P., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. Journal of Functional Programming 17, 1–82 (2007)

Pfenning, F.: Partial polymorphic type inference and higher-order unification. In: LFP 1988: Proceedings of the 1988 ACM conference on LISP and functional programming, pp. 153–163. ACM, New York (1988)

Pierce, B.C., Turner, D.N.: Local type inference. In: 25th ACM Symposium on Principles of Programming Languages (POPL 1998), San Diego, pp. 252–265. ACM, New York (1998)

Reynolds, J.: Towards a theory of type structure. In: Robinet, B. (ed.) Programming Symposium 1974. LNCS, vol. 19. Springer, Heidelberg (1974)

Saha, B., Shao, Z.: Optimal type lifting. In: Types in Compilation, pp. 156–177 (1998), `http://citeseer.ist.psu.edu/article/saha98optimal.html`

Shao, Z.: An overview of the FLINT/ML compiler. In: Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC 1997), Amsterdam, The Netherlands (June 1997)

Shao, Z., League, C., Monnier, S.: Implementing typed intermediate languages. In: ACM SIGPLAN International Conference on Functional Programming (ICFP 1998), ACM SIGPLAN Notices, Baltimore, vol. 34(1), pp. 106–119. ACM Press, New York (1998)

Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: TIL: A type-directed optimizing compiler for ML. In: ACM Conference on Programming Languages Design and Implementation (PLDI 1996), pp. 181–192. ACM, Philadelphia (1996)

Tolmach, A.: Tag-free garbage collection using explicit type parameters. In: ACM Symposium on Lisp and Functional Programming, pp. 1–11. ACM, Orlando (1994)

Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proc 16th ACM Symposium on Principles of Programming Languages, Austin, Texas. ACM Press, New York (1989)

Wells, J.B.: Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In: Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science. IEEE Computer Society Press, Los Alamitos (1994)

Programming with Effects in Coq[*]

Greg Morrisett

School of Engineering and Applied Sciences
Harvard University
greg@eecs.harvard.edu

Abstract. Next-generation programming languages will move beyond simple type systems to include support for formal specifications and mechanically- checked proofs of adherence to those requirements. Already, in the imperative world, languages such as ESC/Java and Spec# integrate Hoare- style pre- and post-conditions into the underlying type system. However, we argue that neither the program logics used in these systems, nor the decision procedures used to discharge verification conditions, are sufficient for establishing deep properties of modular software.

In contrast, the Coq proof development environment provides a powerful program logic (CiC) coupled with an extensible, interactive environment that can combine deep insights from humans with automation to discharge deep proof obligations. Unfortunately, the language at the core of Coq is limited to purely functional programming.

In the Ynot project, we are attempting to address this problem by extending Coq with a new type constructor (the Hoare-triple type), and a few carefully chosen axioms that can be used to build imperative programs in a style quite close to Haskell. I will report on our progress thus far, both in using Ynot to construct modular, extensible libraries for imperative programs, as well as our new compiler infrastructure for generating efficient code from Ynot programs.

[*] Invited Lecture.

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, p. 28, 2008.

Verifying a Semantic $\beta\eta$-Conversion Test for Martin-Löf Type Theory

Andreas Abel[1], Thierry Coquand[2], and Peter Dybjer[2,⋆]

[1] Department of Computer Science, Ludwig-Maximilians-University, Munich
abel@tcs.ifi.lmu.de

[2] Department of Computer Science, Chalmers University of Technology
{coquand,peterd}@cs.chalmers.se

Abstract. Type-checking algorithms for dependent type theories often rely on the interpretation of terms in some semantic domain of *values* when checking equalities. Here we analyze a version of Coquand's algorithm for checking the $\beta\eta$-equality of such semantic values in a theory with a predicative universe hierarchy and large elimination rules. Although this algorithm does not rely on normalization by evaluation explicitly, we show that similar ideas can be employed for its verification. In particular, our proof uses the new notions of *contextual* reification and *strong semantic equality*.

The algorithm is part of a bi-directional type checking algorithm which checks whether a normal term has a certain semantic type, a technique used in the proof assistants Agda and Epigram. We work with an abstract notion of semantic domain in order to accommodate a variety of possible implementation techniques, such as normal forms, weak head normal forms, closures, and compiled code. Our aim is to get closer than previous work to verifying the type-checking algorithms which are actually used in practice.

1 Introduction

Proof assistants based on dependent type theory have now been around for about 25 years. The most prominent representative, Coq [INR07], has become a mature system. It can now be used for larger scale program development and verification, as Leroy's ongoing implementation of a verified compiler shows [Ler06]. Functional programmers have also become more and more interested in using dependent types to ensure program and data structure invariants. New functional languages with dependent types such as Agda 2 [Nor07] and Epigram 2 [CAM07] enjoy increasing popularity.

Although many questions about properties of dependent type theories have been settled in the 1990s, some problems are still waiting for a satisfactory solution. One example is the treatment of equality in implementations of proof

[⋆] Research partially supported by the EU coordination action *TYPES* (510996) and the project *TLCA* of Vetenskapsrådet.

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 29–56, 2008.

assistants. When we check that a dependently typed program is well-typed, we may need to test whether two types are definitionally equal (convertible). Although it is of course impossible for a system to recognize all semantically equal types, a user will feel more comfortable if it can recognize as many as possible. Whenever it fails the user has to resort to *proving* manually that the types are equal. This has the additional drawback of introducing proof-objects for these equalities. In recent years there has therefore been a move from β-equality (computational equality) to the stronger $\beta\eta$-equality (computational and extensional equality).

Recently, algorithms for testing $\beta\eta$-equality have been formulated and verified by the authors both for an untyped notion of conversion [AAD07] and for typed equality judgements [ACD07]. These algorithms use the technique of normalization by evaluation (NbE). However, the algorithms used by proof assistants such as Agda and Epigram [CAM07], use Coquand's $\beta\eta$-conversion test for semantic ”values”, and do not employ the NbE-technique of the above-mentioned papers. Moreover, there is a gap between algorithms on paper and their actual implementation. Proofs on paper are often informal about the treatment of variable names, and they tend to represent values as pieces of abstract syntax. Besides Pollack's [Pol94], Coquand's algorithm [Coq96] is a notable exception: values are represented as closures, and the algorithm explicitly deals with α-equivalence by replacing variables by numbers (de Bruijn levels).

We here continue the work of the second author and verify an implementation of the $\beta\eta$-conversion test close to the one used in practice. In particular:

- Equality is checked incrementally, and not by full normalization followed by a test for syntactical identity.
- The representation of values is abstract. We only require that they form a syntactical applicative structure. In this way, several possible implementations, such as normal forms, closures, and abstract machine code, are covered by our verification.
- The verification approach is extensible: Although we only spell out the proofs for a core of type theory with predicative universes, our development extends to richer languages. We can for example include a unit type, Σ types, proof irrelevance, and inductive types with large eliminations.

Overview. In Sec. 2 we present an abstract type and equality checking algorithm, which only assumes that the domain of values forms a *syntactical applicative structure.* In Sec. 3 inference rules for typing and type equality are given for a version of Martin-Löf type theory with explicit substitutions. An outline of the verification is given in Sec. 4 together with a definition of contextual reification, our main tool for verification. Using contextual reification, an alternative equality test can be formulated, which is shown complete in Sec. 5, by construction of a Kripke model, and proven sound in Sec. 6 via a Kripke logical relation. Completeness of the original algorithm then follows easily in Sec. 7. For soundness, we have to introduce a Kripke logical relation and the concept of strong semantic equality in Sec. 8. In Sec. 9 we discuss the problem of termination of the equality algorithm, which remains open. More proof details can be found in the accompanying technical report [ACD08].

2 Semantic Type and Equality Checking

We consider *dependently typed programs* p to be given as lists of the form

$$\begin{array}{lll} x_0 & : V_0 & = v_0 \\ & \vdots & \\ x_{n-1} & : V_{n-1} & = v_{n-1} \end{array}$$

where x_i is a identifier, V_i its type, and v_i the definition of x_i for $i < n$. (Typically x_i will be a function identifier and v_i the function definition.) The identifiers are not defined simultaneously (which would correspond to mutual recursion), but one-after-another. Hence, V_i and v_i may only refer to previously defined identifiers x_j for $j < i$. A program is *type correct* if each V_i is a well-formed type and each v_i is a term of type V_i. To establish this, the type and definition of the previously defined identifiers may be used. It is reasonable to assume and easy to check that the global identifiers x_i are all distinct; global declarations should not be shadowed. However, local shadowing is allowed; the identifiers x_i may be reused in some of the v_j or V_j. Our type checking algorithm handles shadowing correctly without any informal use of α-conversion.

Dependent types need to be evaluated during type checking. Thus it is common to store them in evaluated form. Without going into further details now, let $v\rho$ ($V\rho$) denote the *evaluation* of term v (type V) in environment ρ. The *environment* maps already checked identifiers to their values. A typing *context* Δ maps already checked identifiers to their types (in evaluated form). Checking a program starts in an empty environment ρ_0 and an empty typing context Δ_0. For $i = 0, \ldots, n-1$, we execute the following steps:

1. Check that V_i is a well-formed type in the current context Δ_i.
2. Evaluate V_i in the current environment: $X_i = V_i\rho_i$.
3. Check that v_i is of type X_i in the current context. This test is written $\Delta_i \vdash v_i \; \delta_{\mathsf{id}} \Uparrow X_i$, where δ_{id} is the identity map on names.
4. Evaluate v_i and extend the current environment by binding x_i to the result: $\rho_{i+1} = (\rho_i, x_i = v_i\rho_i)$.
5. Extend the current context: $\Delta_{i+1} = \Delta, x_i \!:\! X_i$.

The details of type checking depend on the language. We here show how to verify an algorithm for a core language with dependent function types and predicative universes. In the accompanying technical report [ACD08] we consider also natural numbers with primitive recursion. However, the algorithms and proofs in this work directly extend to dependent tuples (Σ and unit type).

2.1 Syntax

Expressions r, s, t are formed from variables x and constants c by application $r\, s$ and function abstraction λxt. The types V_i and terms v_i of a program $p \equiv (x_i : V_i = v_i)_i$ must be in normal form.

$$\begin{array}{lll@{\qquad}l}
\mathsf{Var} & \ni x, y, z & ::= \ldots, \mathsf{x}_1, \mathsf{x}_2, \ldots & \text{variables} \\
\mathsf{Const} & \ni c & ::= \mathsf{Fun} \mid \mathsf{Set}_i & \text{constants } (i \in \mathbb{N}) \\
\mathsf{Exp} & \ni r, s, t & ::= c \mid x \mid \lambda xt \mid r\, s & \text{expressions} \\
\mathsf{Nf} & \ni v, w, V, W & ::= u \mid \lambda xv \mid \mathsf{Fun}\, V\, \lambda xW \mid \mathsf{Set}_i & \beta\text{-normal expressions} \\
\mathsf{Ne} & \ni u & ::= x \mid u\, v & \text{neutral expressions}
\end{array}$$

The set Var of variable identifiers contains, among others, the special variables $\mathsf{x}_1, \mathsf{x}_2, \ldots$ which are called *de Bruijn levels*. To aid the reader, we use the letters A, B, C, V, W for expressions which are to be understood as types and r, s, t, u, v, w for terms. Dependent function types, usually written $\Pi x : A.\, B$, are written $\mathsf{Fun}\, A\, \lambda xB$. When B does not depend on x we have a non-dependent function type and write $A \to B$.

An expression like $\mathsf{Fun}\, A\, \lambda xB$ is parsed as $(\mathsf{Fun}\, A)\, (\lambda xB)$; application is left-associative. To save on parentheses, we introduce the notation $\lambda x.\, t$ where the dot opens a parenthesis which closes as far to the right as syntactically possible. For instance $\lambda x.\, r\, s$ is short for $\lambda x\, (r\, s)$, whereas $\lambda xr\, s$ means $(\lambda xr)\, s$.

The *hello world* program of dependent types, the polymorphic identity, becomes in our notation

$$\mathsf{id} : \mathsf{Fun}\ \mathsf{Set}_0\ \lambda A.\, A \to A = \lambda A \lambda a\, a.$$

The predicative universes Set_i are types of types. A well-formed type $V : \mathsf{Set}_i$ lives in universe i and above. A universe Set_i lives in higher universes Set_j, $j > i$.

2.2 Values

In implementations of dependently typed languages, different representations of values have been chosen: Twelf [PS99] uses de Bruijn terms with explicit substitutions; Agda 2 [Nor07] de Bruijn terms in normal form; and Epigram 2 [CAM07] higher-order abstract syntax. Furthermore, the second author has suggested to use closures [Coq96]. In this article, we abstract over several possible representations, by considering a *syntactical applicative structure with atoms.*

Applicative structure with atoms. This is an applicative structure $(\mathsf{D}, \_\cdot\_)$ which includes all variables and constants as atoms ($\mathsf{Var} \cup \mathsf{Const} \subseteq \mathsf{D}$). Elements of D are denoted by d, e, f, X, Y, Z, E, F and called *values* or *objects. Neutral values* are given inductively by $e, E ::= x \mid e \cdot d$. Neutral application is injective: If $e \cdot d = e' \cdot d'$, then $e = e'$ and $d = d'$. Neutral values are distinct, for instance, $x \cdot \boldsymbol{d} \neq y \cdot \boldsymbol{d'}$. We will sometimes write application as juxtaposition, especially in neutral and constructed values.

The constants Set_i are constructors of arity 0 and Fun is a constructor of arity 2. Constructors are injective, thus, $\mathsf{Fun}\, X\, F = \mathsf{Fun}\, X'\, F'$ implies $X = X'$ and $F = F'$. *Constructed values*, i. e., of the form $c\, \boldsymbol{d}$, are distinguished from each other and from neutral values. It is decidable whether an object is neutral, constructed, or neither. If an object is neutral, we can extract the head variable and the arguments, and similar for constructed objects.

Syntactical applicative structure with atoms. We enrich the applicative structure with an evaluation operation $t\rho \in \mathsf{D}$ for expressions $t \in \mathsf{Exp}$ in an environment $\rho : \mathsf{Var} \rightharpoonup \mathsf{D}$. Let ρ_{id} denote the identity environment. The following axioms must hold for evaluation.

$$\begin{array}{ll} \textsc{eval-c} & c\rho = c \\ \textsc{eval-var} & x\rho = \rho(x) \\ \textsc{eval-fun-e} & (r\, s)\rho = r\rho \cdot s\rho \\ \textsc{app-fun} & (\lambda x t)\rho \cdot d = t(\rho, x\!=\!d) \end{array}$$

An applicative structure which satisfies these equations is called a *syntactical applicative structure* $(\mathsf{D}, \_ \cdot \_, \_\_)$. Barendregt [Bar84, 5.3.1.] adds a sanity condition that the evaluation of an expression may only depend on the valuations of its free variables, but we will not require it. All syntactical λ-models [BL84] [Bar84, 5.3.2.(ii)] are instances of syntactical applicative structures, yet they must additionally fulfill weak extensionality (ξ).

An instance: closures. There are applicative structures which are neither λ-models nor combinatory algebras, for example the representation of values by *closures*. Values are given by the grammar

$$\begin{aligned} \mathsf{D} \ni d &::= [\lambda x t]\rho \mid e \\ e &::= c \mid x \mid e\, d \end{aligned}$$

where $[\lambda x t]\rho$ is a closure such that ρ provides a value for each free variable in $\lambda x t$. Evaluation does not proceed under binders; it is given by the above axioms plus:

$$\begin{array}{ll} \textsc{eval-fun-i} & (\lambda x t)\rho = [\lambda x t]\rho \\ \textsc{app-ne} & e \cdot d = e\, d \end{array}$$

Closures are a standard tool for building interpreters for λ-calculi; the second author has used them to implement a type checker [Coq96]. While for the soundness proof he requires weak extensionality in D, we will not; instead, extensionality of functions in Type Theory is proven via a Kripke model (Section 5).

2.3 Type Checking

In this section, we present a bidirectional type-checking algorithm [PT98] which *checks* a normal term against a type and *infers* the type of a normal expression. In the dependently typed setting, where types may contain computations, the principled approach is to keep types in evaluated form. During the course of type-checking we will have to evaluate terms (see rule INF-FUN-E below). To avoid non-termination, it is crucial to *only evaluate terms which have already been type checked.*

We are ready to present the semantic type checking algorithm, where "semantic" refers to the fact that types are values in D and not type expressions. As usual we describe it using inference rules. These can be read as the clauses of logic programs and we specify the modes (input and output). Note that the modes are not part of the mathematical definition of the inductive judgements—they only describe how the judgements should be executed. Since the rules are deterministic, they also describe a functional implementation of type checking, which can be obtained mechanically from the rules.

In the following definitions δ ranges over special environments, *renamings*, which are finite maps from variables to de Bruijn levels. As before, $t\delta$ denotes the evaluation of t in environment δ.

Semantic (typing) contexts are given by the grammar $\Delta ::= \diamond \mid \Delta, x\!:\!X$, where $x \notin \mathsf{dom}(\Delta)$. If $(x\!:\!X) \in \Delta$ then $\Delta(x) = X$. We write x_Δ for the first de Bruijn level which is not used in Δ, $\mathsf{x}_{\Delta+1}$ for the next one, etc.

Type checking algorithm. We define bidirectional type checking of normal terms and well-formedness checking of normal types by the following three judgements. Herein, $\Delta \in \mathsf{SemCxt}$, $u \in \mathsf{Ne}$, $v, V \in \mathsf{Nf}$, $\delta \in \mathsf{Var} \rightharpoonup \mathsf{Var}$, $X \in \mathsf{D}$ and $i \in \mathbb{N}$.

$\Delta \vdash u\ \delta \Downarrow X$	the type of neutral u is inferred as X
$\Delta \vdash v\ \delta \Uparrow X$	normal v checks against type X
$\Delta \vdash V\ \ \delta \Uparrow \mathsf{Set} \rightsquigarrow i$	V is a well-formed type of inferred level i.

In all judgements we maintain the invariant that Δ assigns types to the free variables in X and $\Delta \circ \delta$ to the free variables in u, v, V.

Type inference. $\Delta \vdash u\ \delta \Downarrow X$. (Inputs: Δ, u, δ. Output: type X of u or fail.)

$$\text{INF-VAR}\ \frac{}{\Delta \vdash x\ \delta \Downarrow \Delta(x\delta)}$$

$$\text{INF-FUN-E}\ \frac{\Delta \vdash u\ \delta \Downarrow \mathsf{Fun}\,X F \qquad \Delta \vdash v\ \delta \Uparrow X}{\Delta \vdash (u\,v)\ \delta \Downarrow F \cdot v\delta}$$

The type of a variable x under renaming δ is just looked up in the context. When computing the type of an application $u\,v$ from the type $\mathsf{Fun}\,X F$ of the function part we need to apply F to the evaluated argument part $v\delta$ (dependent function application). At this point, it is crucial that we have type-checked v already, otherwise the application $F \cdot v\delta$ could diverge.

Type checking. $\Delta \vdash v\ \delta \Uparrow X$. (Inputs: Δ, v, δ, X. Output: succeed or fail.)

$$\textsc{chk-fun-i}\ \frac{\Delta, \mathsf{x}_\Delta : X \vdash v\ (\delta, x{=}\mathsf{x}_\Delta) \Uparrow F \cdot \mathsf{x}_\Delta}{\Delta \vdash (\lambda x v)\ \delta \Uparrow \mathsf{Fun}\, X F}$$

$$\textsc{chk-set}\ \frac{\Delta \vdash V\ \delta \Uparrow \mathsf{Set} \rightsquigarrow i}{\Delta \vdash V\ \delta \Uparrow \mathsf{Set}_j}\ i \le j$$

$$\textsc{chk-inf}\ \frac{\Delta \vdash u\ \delta \Downarrow X \qquad \Delta \vdash X = X' \Uparrow \mathsf{Set} \rightsquigarrow i}{\Delta \vdash u\ \delta \Uparrow X'}$$

When checking an abstraction $\lambda x v$ against a dependent function type value $\mathsf{Fun}\, X F$, we rename x to the next free de Bruijn level x_Δ and check the abstraction body v against $F \cdot \mathsf{x}_\Delta$ in the extended context which binds the abstracted variable to the domain X. To check a neutral term u against X', we infer the type X of u and compare X and X'. The implementation and verification of this comparison will occupy our attention for the remainder of this article.

Type well-formedness. $\Delta \vdash V\ \delta \Uparrow \mathsf{Set} \rightsquigarrow i$. (Inputs: Δ, V, δ. Output: universe level i of V or fail.)

$$\textsc{chk-inf-f}\ \frac{\Delta \vdash V\ \delta \Downarrow \mathsf{Set}_i}{\Delta \vdash V\ \delta \Uparrow \mathsf{Set} \rightsquigarrow i} \qquad \textsc{chk-set-f}\ \frac{}{\Delta \vdash \mathsf{Set}_i\ \delta \Uparrow \mathsf{Set} \rightsquigarrow i+1}$$

$$\textsc{chk-fun-f}\ \frac{\Delta \vdash V\ \delta \Uparrow \mathsf{Set} \rightsquigarrow i \qquad \Delta,\ \mathsf{x}_\Delta : V\delta \vdash W\ (\delta, y{=}\mathsf{x}_\Delta) \Uparrow \mathsf{Set} \rightsquigarrow j}{\Delta \vdash (\mathsf{Fun}\, V\, \lambda y W)\ \delta \Uparrow \mathsf{Set} \rightsquigarrow \max(i,j)}$$

This judgement checks that V is a well-formed type and additionally infers the lowest universe level i this type lives in. The type Set_i is well-formed and lives in level $i+1$. A neutral type is well-formed if its type is computed as Set_i for some i. A function type is well-formed if both domain and codomain are, and it lives in any level both components live in.

Comparison to [Coq96]. The second author has presented a similar type checking algorithm before [Coq96] for unstratified universes $\mathsf{Set} : \mathsf{Set}$. The main difference is that in rule CHK-INF he uses an untyped β-conversion check $X \sim X'$ instead of typed $\beta\eta$-conversion $\Delta \vdash X = X' \Uparrow \mathsf{Set} \rightsquigarrow i$ which we will describe in the following section. A minor difference is that in $\Delta \vdash v\ \delta \Uparrow X$ he uses Δ to assign types to the free variables of term v whereas we use $\Delta \circ \delta$. Consequently, the free variables of X would live in context $\Delta \circ \delta^{-1}$ in his case, however, this is problematic in principle since δ may not be invertible, e. g., in case of shadowing. Since he uses untyped conversion, this is irrelevant, because he never needs to look at the types of free variables in X. In our case, it is crucial.

2.4 Checking Equality

Checking the type of a neutral expression against X' while its type has been inferred as X requires testing the types X and X' for equality. Since types depend on objects, we will also have to compare objects. The principal method to check η-equality is a *type-directed* algorithm. In the following we present such a type-directed algorithm for comparing *values.*

Analogously to type checking, we define three inductive judgements. Herein, $d, d', e, e', X, X' \in \mathsf{D}$, $\Delta \in \mathsf{SemCxt}$, and $i \in \mathbb{N}$.

$\Delta \vdash e = e' \Downarrow X$	neutral e and e' are equal, inferring type X
$\Delta \vdash d = d' \Uparrow X$	d and d' are equal, checked at type X
$\Delta \vdash X = X' \Uparrow \mathsf{Set} \rightsquigarrow i$	X and X' are equal types, inferring universe level i

Inference mode. $\Delta \vdash e = e' \Downarrow X$ (inputs: Δ,e,e', output: X or fail). In inference mode, neutral values e, e' are checked for equality, and their type is inferred.

$$\text{AQ-VAR}\ \frac{}{\Delta \vdash x = x \Downarrow \Delta(x)}$$

$$\text{AQ-FUN}\ \frac{\Delta \vdash e = e' \Downarrow \mathsf{Fun}\, X F \qquad \Delta \vdash d = d' \Uparrow X}{\Delta \vdash e\, d = e'\, d' \Downarrow F \cdot d}$$

A variable is only equal to itself; its type is read from the context. A neutral application $e\,d$ is only equal to another neutral application $e'\,d'$ and the function parts must be equal, as well as the argument parts. The type of e must be a function type $\mathsf{Fun}\, X F$, whose domain X is used to check d and d' for equality.

The type of the application $e\,d$ is computed as $F \cdot d$. We could equally well have chosen to return $F \cdot d'$. That both choices amount to the same follows from the correctness of the equality check; yet we cannot rely on it before we have established correctness. This will be an issue in the correctness proof (Sec. 5); Harper and Pfenning [HP05] have avoided these complications by using simply-typed skeletons to direct the equality algorithm. Their method relies on dependency erasure which works for LF but not for type theories with large eliminations.

Checking mode. $\Delta \vdash d = d' \Uparrow X$ (inputs: Δ, d, d', X, output: succeed or fail).

$$\text{AQ-NE-F}\ \frac{\Delta \vdash e = e' \Downarrow E_1 \qquad \Delta \vdash E_1 = E_2 \Downarrow \mathsf{Set}_i}{\Delta \vdash e = e' \Uparrow E_2}$$

$$\text{AQ-EXT}\ \frac{\Delta, \mathsf{x}_\Delta : X \vdash f \cdot \mathsf{x}_\Delta = f' \cdot \mathsf{x}_\Delta \Uparrow F \cdot \mathsf{x}_\Delta}{\Delta \vdash f = f' \Uparrow \mathsf{Fun}\, X F}$$

$$\text{AQ-TY}\ \frac{\Delta \vdash X = X' \Uparrow \mathsf{Set} \rightsquigarrow i}{\Delta \vdash X = X' \Uparrow \mathsf{Set}_j}\ i \leq j$$

Neutral values e, e' can only be of neutral type E_i, they are passed to inference mode. The check $\Delta \vdash E_1 = E_2 \Downarrow \mathsf{Set}_i$ should actually not be necessary, because we already now that e and e' are well-typed, and types are only present to guide the equality algorithm. However, currently we do not know how to show soundness without it. Such redundant checks are present in other works as well [HP05, p. 77].

Two values f, f' of functional type are equal if applying them to a fresh variable x_Δ makes them equal. This is extensional equality, provided we can substitute arbitrary values for the variable. Had we formulated the algorithm on terms instead of values, this would be a standard substitution theorem. However, in our case it is more difficult. We deal with this issue in Sec. 8.

Values X, X' of type Set_j must be types, we check their equality in the type mode. The inferred universe i must be at most j, otherwise X, X' are not well-typed.

Type mode. $\Delta \vdash X = X' \Uparrow \mathsf{Set} \leadsto i$ (inputs: Δ, X, X', output: i or fail).

$$\text{AQ-TY-NE}\ \frac{\Delta \vdash E = E' \Downarrow \mathsf{Set}_i}{\Delta \vdash E = E' \Uparrow \mathsf{Set} \leadsto i}$$

$$\text{AQ-TY-SET}\ \frac{}{\Delta \vdash \mathsf{Set}_i = \mathsf{Set}_i \Uparrow \mathsf{Set} \leadsto i+1}$$

$$\text{AQ-TY-FUN}\ \frac{\Delta \vdash X = X' \Uparrow \mathsf{Set} \leadsto i \qquad \Delta, \mathsf{x}_\Delta : X \vdash F \cdot \mathsf{x}_\Delta = F' \cdot \mathsf{x}_\Delta \Uparrow \mathsf{Set} \leadsto j}{\Delta \vdash \mathsf{Fun}\, X F = \mathsf{Fun}\, X' F' \Uparrow \mathsf{Set} \leadsto \max(i,j)}$$

A neutral type E can only be equal to another neutral type E', we delegate the test to the inference mode. A universe Set_i is only equal to itself. Function types $\mathsf{Fun}\, X F$ and $\mathsf{Fun}\, X' F'$ are equal if their domains and codomains coincide. For checking the codomains we introduce the fresh variable x_Δ of type X into the context. Again arbitrarily; we could have chosen X' instead. This is another source of asymmetry which complicates the correctness proof; for LF, it can be avoided by considering simply-typed contexts [HP05].

This algorithm is called *semantic* since it compares values. It is part of the core of Agda and Epigram 2. Since it is of practical relevance, it is a worth-while effort to verify it. Correctness of type checking is then a consequence of the correctness of algorithmic equality.

3 Specification: Typing with Explicit Substitutions

We want to prove that our algorithmic equality is correct, so we should say in which sense and provide a specification. There are different ways to present type theory. We choose a formulation with explicit substitutions [ML92] because the typing and equality rules can then be validated directed in any (Kripke) PER model over *any syntactical applicative structure* (see Thm. 1). Altenkirch and Chapman [AC08] exploit this fact for their closure-based definition of values. A formulation with non-explicit (deep) substitution directly validates the inference rules only for PER models over syntactical applicative structures *with extra properties*, e. g., λ-algebras [AC07], or combinatory algebras [ACD07].

Well-formed contexts $\Gamma \vdash$.

$$\textsc{cxt-empty}\ \frac{}{\diamond \vdash} \qquad \textsc{cxt-ext}\ \frac{\Gamma \vdash A}{\Gamma, x\!:\!A \vdash}$$

Well-typed terms $\Gamma \vdash t : A$.

$$\textsc{const}\ \frac{\Gamma \vdash \qquad \Sigma \vdash c : A}{\Gamma \vdash c : A} \qquad \textsc{hyp}\ \frac{\Gamma \vdash \qquad (x\!:\!A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\textsc{conv}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = A'}{\Gamma \vdash t : A'} \qquad \textsc{sub}\ \frac{\Gamma \vdash A : \mathsf{Set}_i}{\Gamma \vdash A : \mathsf{Set}_j}\ i \leq j$$

$$\textsc{fun-f}\ \frac{\Gamma \vdash A : \mathsf{Set}_i \qquad \Gamma, x\!:\!A \vdash B : \mathsf{Set}_i}{\Gamma \vdash \mathsf{Fun}\, A\, \lambda x B : \mathsf{Set}_i}$$

$$\textsc{fun-i}\ \frac{\Gamma, x\!:\!A \vdash t : B}{\Gamma \vdash \lambda x t : \mathsf{Fun}\, A\, \lambda x B} \qquad \textsc{fun-e}\ \frac{\Gamma \vdash r : \mathsf{Fun}\, A\, \lambda x B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\, s : B(\sigma_{\mathsf{id}}, x\!=\!s)}$$

$$\textsc{esubst-f}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma' \vdash t : A}{\Gamma \vdash t\sigma : A\sigma}$$

Well-formed substitutions $\Gamma \vdash \sigma : \Gamma'$.

$$\textsc{subst-ext}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma' \vdash A \qquad \Gamma \vdash s : A\sigma}{\Gamma \vdash (\sigma, x\!=\!s) : (\Gamma', x\!:\!A)}$$

$$\textsc{subst-id}\ \frac{\Gamma \vdash}{\Gamma \vdash \sigma_{\mathsf{id}} : \Gamma} \qquad \textsc{subst-comp}\ \frac{\Gamma_2 \vdash \sigma : \Gamma_3 \qquad \Gamma_1 \vdash \tau : \Gamma_2}{\Gamma_1 \vdash \sigma \circ \tau : \Gamma_3}$$

$$\textsc{subst-weak}\ \frac{\Gamma \vdash \sigma : \Gamma', x\!:\!A, \Gamma''}{\Gamma \vdash \sigma : \Gamma', \Gamma''}$$

Fig. 1. Rules for contexts, types, and terms

We extend the expression syntax by explicit substitutions and introduce syntactical typing contexts:

$$\begin{array}{lll} \mathsf{Exp} \ni r, s, t & ::= \cdots \mid t\sigma & \text{expressions} \\ \mathsf{Subst} \ni \sigma, \tau & ::= (\sigma, x\!=\!t) \mid \sigma_{\mathsf{id}} \mid \sigma \circ \tau & \text{substitutions} \\ \mathsf{Cxt} \ni \Gamma & ::= \diamond \mid \Gamma, x\!:\!A & \text{typing contexts} \end{array}$$

We identify expressions up to α-conversion and adopt the convention that in contexts Γ all variables must be distinct. Hence, we can view Γ as a map from variables to types with finite domain $\mathsf{dom}(\Gamma)$ and let $\Gamma(x) = A$ iff $(x\!:\!A) \in \Gamma$. In context extensions $\Gamma, x : A$ we assume $x \notin \mathsf{dom}(\Gamma)$. As usual $\mathsf{FV}(t)$ is the set of free variables of t. We let $\mathsf{FV}(t_1, \ldots, t_n) = \mathsf{FV}(t_1) \cup \cdots \cup \mathsf{FV}(t_n)$ and $\mathsf{FV}(\Gamma) = \bigcup_{x \in \mathsf{dom}(\Gamma)} \mathsf{FV}(\Gamma(x))$.

Equality $\Gamma \vdash t = t' : A$. Equivalence, hypotheses, conversion.

$$\textsc{eq-refl}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A} \qquad \textsc{eq-sym}\ \frac{\Gamma \vdash t = t' : A}{\Gamma \vdash t' = t : A}$$

$$\textsc{eq-trans}\ \frac{\Gamma \vdash t = t' : A \qquad \Gamma \vdash t' = t'' : A}{\Gamma \vdash t = t'' : A}$$

$$\textsc{eq-const}\ \frac{\Gamma \vdash \qquad \Sigma \vdash c : A}{\Gamma \vdash c = c : A} \qquad \textsc{eq-hyp}\ \frac{\Gamma \vdash \qquad (x\!:\!A) \in \Gamma}{\Gamma \vdash x = x : A}$$

$$\textsc{eq-conv}\ \frac{\Gamma \vdash t = t' : A \qquad \Gamma \vdash A = A'}{\Gamma \vdash t = t' : A'} \qquad \textsc{eq-sub}\ \frac{\Gamma \vdash A = A' : \mathsf{Set}_i}{\Gamma \vdash A = A' : \mathsf{Set}_j}\ i \leq j$$

Dependent functions.

$$\textsc{eq-fun-f}\ \frac{\Gamma \vdash A = A' : \mathsf{Set}_i \qquad \Gamma, x\!:\!A \vdash B = B' : \mathsf{Set}_i}{\Gamma \vdash \mathsf{Fun}\, A\, \lambda x B = \mathsf{Fun}\, A'\, \lambda x B' : \mathsf{Set}_i}$$

$$\textsc{eq-fun-i}\ \frac{\Gamma, x\!:\!A \vdash t = t' : B}{\Gamma \vdash \lambda x t = \lambda x t' : \mathsf{Fun}\, A\, \lambda x B}$$

$$\textsc{eq-fun-e}\ \frac{\Gamma \vdash r = r' : \mathsf{Fun}\, A\, \lambda x B \qquad \Gamma \vdash s = s' : A}{\Gamma \vdash r\, s = r'\, s' : B(\sigma_{\mathsf{id}}, x\!=\!s)}$$

$$\textsc{eq-fun-}\beta\ \frac{\Gamma, x\!:\!A \vdash t : B \qquad \Gamma \vdash s : A}{\Gamma \vdash (\lambda x t)\, s = t(\sigma_{\mathsf{id}}, x\!=\!s) : B(\sigma_{\mathsf{id}}, x\!=\!s)}$$

$$\textsc{eq-fun-}\eta\ \frac{\Gamma \vdash t : \mathsf{Fun}\, A\, \lambda x B}{\Gamma \vdash (\lambda x.\, t\, x) = t : \mathsf{Fun}\, A\, \lambda x B}\ x \notin \mathsf{dom}(\Gamma)$$

Fig. 2. Equality rules

We have extended the language by explicit substitutions so we need to extend the notion of syntactical applicative structure, to ensure substitutions are evaluated reasonably:

$$\begin{array}{lr} \textsc{eval-subst-id} & \sigma_{\mathsf{id}}\rho = \rho \\ \textsc{eval-subst-comp} & (\sigma \circ \sigma')\rho = \sigma(\sigma'\rho) \\ \textsc{eval-subst-ext} & (\sigma, x\!=\!s)\rho = (\sigma\rho, x\!=\!s\rho) \\ \textsc{eval-esubst} & (t\sigma)\rho = t(\sigma\rho) \end{array}$$

Herein, $\sigma\rho$ is defined by $x(\sigma\rho) = (x\sigma)\rho$.

Judgements. Our type theory with explicit substitutions has the following forms of judgement:

$\Gamma \vdash$	Γ is a well-formed context
$\Gamma \vdash A$	A is a well-formed type in Γ
$\Gamma \vdash t : A$	t has type A in Γ
$\Gamma \vdash \sigma : \Gamma'$	σ is a well-formed substitution in Γ
$\Gamma \vdash A = A'$	A and A' are equal types in Γ
$\Gamma \vdash t = t' : A$	t and t' are equal terms of type A in Γ
$\Gamma \vdash \sigma = \sigma' : \Gamma'$	σ and σ' are equal substitutions in Γ

Equivalence rules and weakening.

$$\textsc{eq-subst-refl}\ \frac{\Gamma \vdash \sigma : \Gamma'}{\Gamma \vdash \sigma = \sigma : \Gamma'} \qquad \textsc{eq-subst-sym}\ \frac{\Gamma \vdash \sigma = \sigma' : \Gamma'}{\Gamma \vdash \sigma' = \sigma : \Gamma'}$$

$$\textsc{eq-subst-trans}\ \frac{\Gamma_1 \vdash \sigma = \sigma' : \Gamma_2 \qquad \Gamma_2 \vdash \sigma' = \sigma'' : \Gamma_3}{\Gamma_1 \vdash \sigma = \sigma'' : \Gamma_3}$$

$$\textsc{eq-subst-weak}\ \frac{\Gamma \vdash \sigma = \sigma' : \Gamma', x{:}A, \Gamma''}{\Gamma \vdash \sigma = \sigma' : \Gamma', \Gamma''}$$

Rules of the category of contexts and substitutions.

$$\textsc{eq-subst-id-l}\ \frac{\Gamma \vdash \sigma : \Gamma'}{\Gamma \vdash \sigma_{\mathsf{id}} \circ \sigma = \sigma : \Gamma'} \qquad \textsc{eq-subst-id-r}\ \frac{\Gamma \vdash \sigma : \Gamma'}{\Gamma \vdash \sigma \circ \sigma_{\mathsf{id}} = \sigma : \Gamma'}$$

$$\textsc{eq-subst-assoc}\ \frac{\Gamma_3 \vdash \sigma : \Gamma_4 \qquad \Gamma_2 \vdash \sigma' : \Gamma_3 \qquad \Gamma_1 \vdash \sigma'' : \Gamma_2}{\Gamma_1 \vdash (\sigma \circ \sigma') \circ \sigma'' = \sigma \circ (\sigma' \circ \sigma'') : \Gamma_4}$$

Rules for the empty substitution and substitution extension.

$$\textsc{eq-subst-empty-}\eta\ \frac{\Gamma \vdash \sigma : \diamond \qquad \Gamma \vdash \sigma' : \diamond}{\Gamma \vdash \sigma = \sigma' : \diamond}$$

$$\textsc{eq-subst-ext-}\beta\ \frac{\Gamma_2 \vdash \sigma : \Gamma_3 \qquad \Gamma_3 \vdash A \qquad \Gamma_2 \vdash s : A\sigma \qquad \Gamma_1 \vdash \tau : \Gamma_2}{\Gamma_1 \vdash (\sigma, x{=}s) \circ \tau = (\sigma \circ \tau, x{=}s\tau) : \Gamma_3, x{:}A}$$

$$\textsc{eq-subst-ext-}\eta\ \frac{\Gamma, x{:}A \vdash}{\Gamma, x{:}A \vdash (\sigma_{\mathsf{id}}, x{=}x) = \sigma_{\mathsf{id}} : \Gamma, x{:}A}$$

$$\textsc{eq-subst-ext-weak}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma', x{:}A \vdash \qquad \Gamma \vdash t : A\sigma}{\Gamma \vdash (\sigma, x{=}t) = \sigma : \Gamma'}$$

Congruence rules.

$$\textsc{eq-subst-ext}\ \frac{\Gamma \vdash \sigma = \sigma' : \Gamma' \qquad \Gamma' \vdash A \qquad \Gamma \vdash s = s' : A\sigma}{\Gamma \vdash (\sigma, x{=}s) = (\sigma', x{=}s') : (\Gamma', x{:}A)}$$

$$\textsc{eq-subst-comp}\ \frac{\Gamma_2 \vdash \sigma = \sigma' : \Gamma_3 \qquad \Gamma_1 \vdash \tau = \tau' : \Gamma_2}{\Gamma_1 \vdash \sigma \circ \tau = \sigma' \circ \tau' : \Gamma_3}$$

Fig. 3. Equality rules for substitutions $\Gamma \vdash \sigma = \sigma' : \Delta$

$$\textsc{eq-esubst-f}\ \frac{\Gamma \vdash \sigma = \sigma' : \Delta \qquad \Delta \vdash t = t' : A}{\Gamma \vdash t\sigma = t'\sigma' : A\sigma}$$

$$\textsc{eq-esubst-id}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash t\sigma_{\mathsf{id}} = t : A}$$

$$\textsc{eq-esubst-comp}\ \frac{\Gamma \vdash \tau : \Gamma' \qquad \Gamma' \vdash \sigma : \Gamma'' \qquad \Gamma'' \vdash t : A}{\Gamma \vdash t(\sigma \circ \tau) = (t\sigma)\tau : A(\sigma \circ \tau)}$$

$$\textsc{eq-esubst-c}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Sigma \vdash c : A}{\Gamma \vdash c\sigma = c : A}$$

$$\textsc{eq-esubst-var}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma', x{:}A \vdash \qquad \Gamma \vdash t : A\sigma}{\Gamma \vdash x(\sigma, x{=}t) = t : A\sigma}$$

$$\textsc{eq-esubst-fun-f}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma' \vdash A : \mathsf{Set}_i \qquad \Gamma', x{:}A \vdash B : \mathsf{Set}_i}{\Gamma \vdash (\mathsf{Fun}\, A\, \lambda x B)\sigma = \mathsf{Fun}\, (A\sigma)\, (\lambda x B)\sigma : \mathsf{Set}_i}\ x \notin \mathsf{dom}(\Gamma)$$

$$\textsc{eq-esubst-fun-i}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma', x{:}A \vdash t : B}{\Gamma \vdash (\lambda x t)\sigma = \lambda x.\, t(\sigma, x{=}x) : (\mathsf{Fun}\, A\, \lambda x B)\sigma}\ x \notin \mathsf{dom}(\Gamma)$$

$$\textsc{eq-esubst-fun-e}\ \frac{\Gamma \vdash \sigma : \Gamma' \qquad \Gamma' \vdash r : \mathsf{Fun}\, A\, \lambda x B \qquad \Gamma' \vdash s : A}{\Gamma \vdash (r\, s)\sigma = r\sigma\, s\sigma : B(\sigma, x{=}s\sigma)}$$

Fig. 4. Equality rules for explicit substitutions

For an arbitrary judgement, we write $\Gamma \vdash J$, where J is a collection of syntactic entities (terms, contexts, substitutions) to the right of $\vdash$ in a judgement. $\mathsf{FV}(J)$ is the union of the free variable sets of all entities in J. Exceptions are $\mathsf{FV}(\sigma : \Delta)$, which is defined as $\bigcup_{x \in \mathsf{dom}(\Delta)} \mathsf{FV}(\Delta(x), \sigma(x))$, and $\mathsf{FV}(\sigma = \sigma' : \Delta) = \bigcup_{x \in \mathsf{dom}(\Delta)} \mathsf{FV}(\Delta(x), \sigma(x), \sigma'(x))$.

The judgements on types can be defined in terms of the judgement on terms.

$$\begin{array}{lcll} \Gamma \vdash A & \Longleftrightarrow & \Gamma \vdash A : \mathsf{Set}_i & \text{for some } i \\ \Gamma \vdash A = A' & \Longleftrightarrow & \Gamma \vdash A = A' : \mathsf{Set}_i & \text{for some } i \end{array}$$

The *inference rules* for the other judgements are given in figures 1, 2, 3, and 4. They are inspired by categorical presentations of type theory, in particular, categories with families [Dyb96], which have been inspired by Martin-Löf's substitution calculus [ML92]. Rule CONST relies on an auxiliary judgement $\Sigma \vdash c : A$ meaning constant c can be assigned type A. Its only rule is:

$$\textsc{set-f}\ \frac{}{\Sigma \vdash \mathsf{Set}_i : \mathsf{Set}_{i+1}}$$

In extensions of the core theory, the signature Σ provides the types of constructors of inductive types like the natural numbers.

The judgements enjoy some standard properties, like weakening, inversion of typing, and well-formedness of contexts, types and terms (syntactic validity).

4 Verification Plan and Contextual Reification

A standard method to show completeness of the algorithmic equality would be the following [HP05].

1. Define a Kripke logical relation $\Delta \vdash d = d' : X$ on a semantic context Δ and values d, d' by induction on the type X. We will call this relation *Kripke model.* For base types X, let the relation coincide with algorithmic equality $\Delta \vdash d = d' \Uparrow X$, for function types do the usual functional construction: $\Delta \vdash f = f' : \mathsf{Fun}\, X F$ iff $\Delta' \vdash d = d' : X$ implies $\Delta' \vdash f \cdot d = f' \cdot d' : F \cdot d$ for all d, d' and all extensions Δ' of Δ.
2. Show that if two values are related in the model, then the algorithm relates them as well. Following Schürmann and Sarnat [SS08] we call this the *escape* lemma, since it allows to "get out of the logical relation".
3. Finally show validity of the inference rules w. r. t. the model, i.e., if two terms t, t' are equal of type A, then for each well-formed environment ρ, we have $t\rho = t'\rho : A\rho$ in the model, which implies that the algorithm accepts t, t' as equal.

In particular, the relation $\Delta \vdash \_ = \_ : X$ needs to be a partial equivalence, in order to validate symmetry and transitivity rules. But due to the asymmetric nature of algorithmic equality (rules AQ-FUN and AQ-TY-FUN), this can only be shown if we have soundness, which at this point we cannot obtain.

Normalization-by-evaluation (NbE) [ML75, BS91, Coq94, Dan99] to the rescue! There already are equality algorithms for dependent types with large eliminations which are based on semantics [AAD07, ACD07]. Two semantic values are considered equal if they reify to the same expression. *Reification* at a type X converts a value to a term, η-expanding it on the fly according to type X. It turns out that we can verify the algorithmic equality by relating it to NbE.

We will verify algorithmic equality according to this plan:

1. Define normalization-by-evaluation for our setting. This amounts to defining *contextual reification* $\Delta \vdash d \searrow v \Uparrow X$ which converts value d of type X in context Δ to normal form v.
2. Show completeness of NbE (Sec. 5), meaning that if one takes two judgmentally equal terms t, t', evaluates and reifies them, one arrives at the same normal form v (Cor. 1). To this end, we construct a Kripke model based on reification, meaning that two values are equal at base type if they reify to the same normal form.

3. Show soundness of NbE (Sec. 6), meaning that if we take a term t, evaluate and reify it, we arrive at a normal form v judgmentally equal to t (Cor. 2). The main tool is a Kripke logical relation between well-typed terms t and semantic objects d, which for base types states that d reifies to a normal form v which is judgmentally equal to t.
4. Show completeness of the algorithmic equality (Sec. 7). This is a corollary of completeness of NbE, since we will see that if two values reify to the same normal form, then the algorithm will accept them as equal (Lemma 7).
5. Show soundness of the algorithm (Sec. 8). The direct approach, showing that two algorithmically equal values reify to the same normal form, fails due to the asymmetry of the algorithm. We introduce the concept of strong semantic typing and equality (a super Kripke model, so to say) and prove that the algorithm is sound for strong semantic equality. By establishing that the inference rules are valid in the super Kripke model (Cor. 4), and that equality in the super Kripke model entails equality in the Kripke model, we finally show that well-typed terms whose values are algorithmically equal reify to the same normal form, thus, are judgmentally equal (Thm. 3).

What remains open is termination of algorithmic equality.

4.1 Contextual Reification

Reification [BS91] converts a semantic value to a syntactic term, η-expanding it on the fly. It is defined by recursion on the type of the value. In previous NbE approaches [BS91, ACD07] the semantics of base types has been defined as a set of η-long neutral terms, thus, reification at base type is simply the identity. In our approach, the semantics of base types is a set of neutral values, which are not η-expanded and need to be reified recursively. This can only happen if reification has access to the types of the free variables of a neutral value. For example, to reify $x\,d\,d'$ at base type we need to retrieve the type $\mathsf{Fun}\,X\,F$ of x, recursively reify d at type X, compute $F \cdot d = \mathsf{Fun}\,X'\,F'$ and recursively reify d' at type X'. For this task, we introduce a new form of reification which is parameterized by a semantic typing context Δ, hence the name *contextual reification.*

We simultaneously define three inductive judgements by the rules to follow. Herein, $\Delta \in \mathsf{SemCxt}$, $d, e, X \in \mathsf{D}$, $u \in \mathsf{Ne}$, $v, V \in \mathsf{Nf}$ and $i \in \mathbb{N}$.

$\Delta \vdash e \searrow u \Downarrow X$	neutral value e reifies to u, inferring its type X
$\Delta \vdash d \searrow v \Uparrow X$	value d reifies to normal form v at type X
$\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \leadsto i$	type value X reifies to V, inferring its level i.

Inference mode. $\Delta \vdash e \searrow u \Downarrow X$ (inputs: Δ, e, outputs: u, X or fail).

$$\textsc{reify-var}\ \frac{}{\Delta \vdash x \searrow x \Downarrow \Delta(x)}$$

$$\textsc{reify-fun-e}\ \frac{\Delta \vdash e \searrow u \Downarrow \mathsf{Fun}\,X F \qquad \Delta \vdash d \searrow v \Uparrow X}{\Delta \vdash e\,d \searrow u\,v \Downarrow F \cdot d}$$

Variables reify to themselves and neutral applications to neutral applications. The type information flows out of the context Δ and is used in REIFY-FUN-E to reify d of type X in checking mode.

Checking mode. $\Delta \vdash d \searrow v \Uparrow X$ (inputs: Δ, d, X, output: v or fail).

$$\text{REIFY-NE}\ \frac{\Delta \vdash e \searrow u \Downarrow E_1 \qquad \Delta \vdash E_1 \searrow u' \Downarrow \mathsf{Set}_i \qquad \Delta \vdash E_2 \searrow u' \Downarrow \mathsf{Set}_i}{\Delta \vdash e \searrow u \Uparrow E_2}$$

$$\text{REIFY-EXT}\ \frac{\Delta, x{:}X \vdash f \cdot x \searrow v \Uparrow F \cdot x}{\Delta \vdash f \searrow \lambda x v \Uparrow \mathsf{Fun}\, X F}$$

$$\text{REIFY-TY}\ \frac{\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow i}{\Delta \vdash X \searrow V \Uparrow \mathsf{Set}_j}\ i \le j$$

Any value f of functional type is reified by applying it to a fresh variable x. Note that this can trigger further evaluation, e. g., in the λ-model where functional values are just weak head normal forms or closures. The result of reifying a functional value is always a λ-abstraction, which means that reification returns η-long forms.

At neutral type E_2, objects e need to be neutral and are reified in inference mode. The inferred type E_1 needs to be equal to E_2—this is checked by reifying both types, expecting the same normal form.

Type mode. $\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow i$ (inputs: Δ, X, outputs: V, i or fail).

$$\text{REIFY-TY-NE}\ \frac{\Delta \vdash e \searrow u \Downarrow \mathsf{Set}_i}{\Delta \vdash e \searrow u \Uparrow \mathsf{Set} \rightsquigarrow i}$$

$$\text{REIFY-TY-SET}\ \frac{}{\Delta \vdash \mathsf{Set}_i \searrow \mathsf{Set}_i \Uparrow \mathsf{Set} \rightsquigarrow i+1}$$

$$\text{REIFY-TY-FUN}\ \frac{\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow i \qquad \Delta, x{:}X \vdash F \cdot x \searrow W \Uparrow \mathsf{Set} \rightsquigarrow j}{\Delta \vdash \mathsf{Fun}\, X F \searrow \mathsf{Fun}\, V\, \lambda x W \Uparrow \mathsf{Set} \rightsquigarrow \max(i,j)}$$

Function type values reify to function type expressions in long normal form, universes to universes and neutral type values to neutral type expressions.

We write $\Delta \vdash e, e' \searrow u \Downarrow X$ for $\Delta \vdash e \searrow u \Downarrow X$ and $\Delta \vdash e' \searrow u \Downarrow X$. This gives a basic equality on neutral objects (which is used in rule REIFY-NE, for instance).

We say Δ' *extends* Δ, written $\Delta' \le \Delta$, if $\Delta'(x) = \Delta(x)$ for all $x \in \mathsf{dom}(\Delta)$. (The direction of $\le$ is as in subtyping.) Reification is closed under weakening of contexts, i.e., reifying in an extended context produces an α-equivalent normal form. Reification provides us with a normalization function: given a closed term $t : A$, the normal form v is obtained by $\diamond \vdash t\rho_{\mathsf{id}} \searrow v \Uparrow A\rho_{\mathsf{id}}$.

5 Kripke Model and Completeness of NbE

Dependent types complicate the definition of a logical relation, because one cannot use structural induction on the type expression. Instead one needs to simultaneously define the "good" type values X by induction and their denotation, a relation on objects, by recursion [Dyb00]. We spell out this construction for our Kripke model in sections 5.1–5.3. In Section 5.4 we prove that it models our inference rules.

5.1 An Induction Measure

If $\mathcal{X} \subseteq \mathsf{D}$ and $\mathcal{F}(d) \subseteq \mathsf{D}$ for each $d \in \mathcal{X}$, then the dependent function space

$$\Pi\,\mathcal{X}\,\mathcal{F} = \{f \mid \forall d \in \mathcal{X}.\, f \cdot d \in \mathcal{F}(d)\}$$

is another subset of D. For $i = 0, 1, \ldots$ we define the sets $\mathcal{T}_i \subseteq \mathsf{D} \times \mathcal{P}(\mathsf{D})$ inductively as follows:

$$\frac{}{(E, \mathsf{D}) \in \mathcal{T}_i} \qquad \frac{}{(\mathsf{Set}_j, |\mathcal{T}_j|) \in \mathcal{T}_i}\; j < i$$

$$\frac{(X, \mathcal{X}) \in \mathcal{T}_i \qquad \forall d \in \mathcal{X}.\, (F \cdot d, \mathcal{F}(d)) \in \mathcal{T}_i}{(\mathsf{Fun}\, X F,\ \Pi\,\mathcal{X}\,\mathcal{F}) \in \mathcal{T}_i}$$

Herein, $|\mathcal{T}_i| = \{X \mid \exists \mathcal{X}.\, (X, \mathcal{X}) \in \mathcal{T}_i\}$. We define the relation $: \subseteq \mathsf{D} \times \mathsf{D}$ by

$$d : X \iff \exists \mathcal{X}, i.\, (X, \mathcal{X}) \in \mathcal{T}_i \text{ and } d \in \mathcal{X}$$

As a special case, $X : \mathsf{Set}_i \iff X \in |\mathcal{T}_i|$. We will use the derivation of membership in $\mathcal{T}_i$ as induction measure, quoted as "induction on $X : \mathsf{Set}_i$".

5.2 Construction of the Kripke Model

It is tempting to define $\Delta \vdash d = d' : X$ for base types directly as "$\Delta \vdash d \searrow v \Uparrow X$ and $\Delta \vdash d' \searrow v \Uparrow X$ for some v". However, then the proof of the escape lemma will fail, because during reification of function types, their domains flow into the context. Reifying two function types will soon take place in different contexts. We need to be more general and relate semantic objects at a priori different types in a priori different contexts. Thus, in the following we define a relation $\Delta \vdash d : X \circledS \Delta' \vdash d' : X'$ for the purpose of proving the escape lemma, and we obtain $\Delta \vdash d = d' : X$ as a special case in Sec. 5.3.

By lexicographic induction on i and $X' : \mathsf{Set}_i$ we define the relations:

$$\begin{array}{l} \_ \vdash \_ : \_ \circledS \_ \vdash X' : \mathsf{Set}_i \\ \_ \vdash \_ : \_ \circledS \_ \vdash \_ : X' \end{array}$$

– *Case* $E' : \mathsf{Set}_i$.

$$\begin{array}{l} \Delta \vdash X : Z \circledS \Delta' \vdash E' : \mathsf{Set}_i \\ \quad \iff X = E \text{ neutral and } Z = \mathsf{Set}_i \text{ and} \\ \qquad\quad \Delta \vdash E \searrow U \Downarrow \mathsf{Set}_j \text{ and } \Delta' \vdash E' \searrow U \Downarrow \mathsf{Set}_{j'} \text{ for some } U \text{ and } j, j' \leq i \end{array}$$

$$\begin{array}{l} \Delta \vdash d : X \circledS \Delta' \vdash d' : E' \\ \quad \iff X = E \text{ neutral and } \Delta \vdash d \searrow u \Downarrow \hat{E} \text{ and } \Delta' \vdash d' \searrow u \Downarrow \hat{E}' \text{ and} \\ \qquad\quad \Delta \vdash E, \hat{E} \searrow U \Downarrow \mathsf{Set}_j \text{ and } \Delta' \vdash E', \hat{E}' \searrow U \Downarrow \mathsf{Set}_{j'} \\ \qquad\quad \text{for some } E, \hat{E}, \hat{E}', u, U, \text{ and } j, j' \leq i \end{array}$$

- *Case* $\mathsf{Set}_j : \mathsf{Set}_i$ for $j < i$.

$$\begin{array}{ll} \Delta \vdash X : Z \circledS \Delta' \vdash \mathsf{Set}_j : \mathsf{Set}_i \iff & X = \mathsf{Set}_j \text{ and } Z = \mathsf{Set}_i \\ \Delta \vdash d : X \circledS \Delta' \vdash d' : \mathsf{Set}_j & \text{has already been defined} \end{array}$$

- *Case* $\mathsf{Fun}\, X'\, F' : \mathsf{Set}_i$ where $X' : \mathsf{Set}_i$ and $F' \cdot d : \mathsf{Set}_i$ for all $d : X'$.

$$\begin{array}{l} \Delta \vdash Y : Z \circledS \Delta' \vdash \mathsf{Fun}\, X'F' : \mathsf{Set}_i \\ \quad \iff Y = \mathsf{Fun}\, XF \text{ for some } X, F \text{ and } Z = \mathsf{Set}_i \text{ and} \\ \qquad\quad \Delta \vdash X : \mathsf{Set}_i \circledS \Delta' \vdash X' : \mathsf{Set}_i \text{ and} \\ \qquad\quad \text{for all } \hat{\Delta} \leq \Delta, \hat{\Delta}' \leq \Delta', d, d',\ \hat{\Delta} \vdash d : X \circledS \hat{\Delta}' \vdash d' : X' \\ \qquad\qquad\quad \text{implies } \hat{\Delta} \vdash F \cdot d : \mathsf{Set}_i \circledS \hat{\Delta}' \vdash F' \cdot d' : \mathsf{Set}_i \end{array}$$

$$\begin{array}{l} \Delta \vdash f : Y \circledS \Delta' \vdash f : \mathsf{Fun}\, X'F' \\ \quad \iff Y = \mathsf{Fun}\, XF \text{ for some } X, F \text{ and} \\ \qquad\quad \text{for all } \hat{\Delta} \leq \Delta, \hat{\Delta}' \leq \Delta', d, d',\ \hat{\Delta} \vdash d : X \circledS \hat{\Delta}' \vdash d' : X' \\ \qquad\qquad\quad \text{implies } \hat{\Delta} \vdash f \cdot d : F \cdot d \circledS \hat{\Delta}' \vdash f' \cdot d' : F' \cdot d' \end{array}$$

Lemma 1. $\circledS$ *is symmetric and transitive.*

Lemma 2 (Type conversion). *Let* $X' : \mathsf{Set}_i$*. If* $\Delta \vdash d : X \circledS \Delta' \vdash d' : X'$ *and* $\Delta' \vdash X' : \mathsf{Set}_i \circledS \Delta' \vdash X'' : \mathsf{Set}_i$ *then* $\Delta \vdash d : X \circledS \Delta' \vdash d' : X''$*.*

Proof. By induction on $X' : \mathsf{Set}_i$.

Lemma 3 (Into and out of the model / escape lemma). *Let* $X' : \mathsf{Set}_i$*. Then*

1. *(In:) If* $\Delta \vdash e \searrow u \Downarrow X$ *and* $\Delta' \vdash e' \searrow u \Downarrow X'$ *then* $\Delta \vdash e : X \circledS \Delta' \vdash e' : X'$*.*
2. *(Out:) If* $\Delta \vdash d : X \circledS \Delta' \vdash d' : X'$ *then* $\Delta \vdash d \searrow v \Uparrow X$ *and* $\Delta' \vdash d' \searrow v \Uparrow X'$ *for some* v*.*
3. *(Out-Type:) If* $\Delta \vdash X : \mathsf{Set}_i \circledS \Delta' \vdash X' : \mathsf{Set}_i$ $\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow j$ *and* $\Delta' \vdash X' \searrow V \Uparrow \mathsf{Set} \rightsquigarrow j'$ *for some* V *and* $j, j' \leq i$*.*

Proof. Simultaneously by induction on $X' : \mathsf{Set}_i$.

5.3 The Kripke Model

We now define

$$\Delta \vdash d = d' : X \iff \Delta \vdash d : X \circledS \Delta \vdash d' : X.$$

We can view the relation $\Delta \vdash d = d' : X$ as inductively generated by the following rules:

$$\frac{}{\Delta \vdash \mathsf{Set}_i = \mathsf{Set}_i : \mathsf{Set}_j}\ i < j$$

$$\frac{\Delta \vdash E \searrow u \Downarrow \mathsf{Set}_i \qquad \Delta \vdash E' \searrow u \Downarrow \mathsf{Set}_i}{\Delta \vdash E = E' : \mathsf{Set}_j}\ i \leq j$$

$$\frac{\Delta \vdash e \searrow u \Downarrow E_1 \qquad \Delta \vdash e' \searrow u \Downarrow E_2 \qquad \Delta \vdash E_0, E_1, E_2 \searrow u' \Downarrow \mathsf{Set}_i}{\Delta \vdash e = e' : E_0}$$

$$\frac{\begin{array}{c}\Delta \vdash X = X' : \mathsf{Set}_i \\ \Delta' \vdash F \cdot d = F' \cdot d' : \mathsf{Set}_i \text{ for all } \Delta' \leq \Delta \text{ and } \Delta' \vdash d = d' : X\end{array}}{\Delta \vdash \mathsf{Fun}\, X F = \mathsf{Fun}\, X' F' : \mathsf{Set}_i}$$

$$\frac{\Delta' \vdash f \cdot d = f' \cdot d' : F \cdot d \text{ for all } \Delta' \leq \Delta \text{ and } \Delta' \vdash d = d' : X}{\Delta \vdash f = f' : \mathsf{Fun}\, X F}$$

We let $\Delta \vdash X = X'$ iff there exists an i such that $\Delta \vdash X = X' : \mathsf{Set}_i$. We write $\Delta \vdash d : X$ for $\Delta \vdash d = d : X$.

5.4 Validity of Syntactic Typing

Now we can show that all the rules for our typing and equality judgements are valid in the model. As a byproduct, we get completeness of NbE.

Let

$$\Delta \vdash \rho = \rho' : \Gamma \iff \forall x \in \mathsf{dom}(\Gamma).\, \Delta \vdash \rho(x) = \rho'(x) : \Gamma(x)\rho$$

Define $\Gamma \Vdash J$, meaning that $\Gamma \vdash J$ is valid in the Kripke model, as follows:

$$\begin{array}{lll}
\diamond \Vdash & :\iff & \text{true} \\
\Gamma, x{:}A \Vdash & :\iff & \Gamma \Vdash A \\[4pt]
\Gamma \Vdash A & :\iff & \text{either } A = \mathsf{Set}_i \text{ and } \Gamma \Vdash \text{ or } \Gamma \Vdash A : \mathsf{Set}_i \text{ for some } i \\[4pt]
\Gamma \Vdash t : A & :\iff & \Gamma \Vdash t = t : A \\
\Gamma \Vdash t = t' : A & :\iff & \Gamma \Vdash A \text{ and } \forall \Delta \vdash \rho = \rho' : \Gamma.\ \Delta \vdash t\rho = t'\rho' : A\rho \\[4pt]
\Gamma \Vdash \sigma : \Gamma' & :\iff & \Gamma \Vdash \sigma = \sigma : \Gamma' \\
\Gamma \Vdash \sigma = \sigma' : \Gamma' & :\iff & \Gamma \Vdash \text{ and } \Gamma' \Vdash \text{ and } \forall \Delta \vdash \rho = \rho' : \Gamma.\ \Delta \vdash \sigma\rho = \sigma'\rho' : \Gamma'
\end{array}$$

Theorem 1 (Soundness of the inference rules). *If $\Gamma \vdash J$ then $\Gamma \Vdash J$.*

Proof. By induction on $\Gamma \vdash J$. (Tedious, but easy.)

Lemma 4. $\Gamma\rho_{\mathsf{id}} \vdash \rho_{\mathsf{id}} = \rho_{\mathsf{id}} : \Gamma$.

Proof. For each $x \in \mathsf{dom}(\Gamma)$, we have $\Gamma\rho_{\mathsf{id}} \vdash x\rho_{\mathsf{id}} = x\rho_{\mathsf{id}} : \Gamma(x)\rho_{\mathsf{id}}$.

Corollary 1 (Completeness of NbE). *If $\Gamma \vdash t = t' : A$ then $\Gamma\rho_{\mathsf{id}} \vdash t\rho_{\mathsf{id}} \searrow v \Uparrow A\rho_{\mathsf{id}}$ and $\Gamma\rho_{\mathsf{id}} \vdash t'\rho_{\mathsf{id}} \searrow v \Uparrow A\rho_{\mathsf{id}}$ for some v.*

6 Kripke Logical Relation and Soundness of NbE

In the previous section we have defined a logical relation $\Delta \vdash d : X \circledS \Delta' \vdash d' : X'$ between two semantic objects in their typing environments. Now we will define a logical relation between a well-typed expression $\Gamma \vdash t : A$ and a value d in its typing environment $\Delta \vdash X$. The relation shall imply that d at type X reifies in Δ to a normal form v which is judgmentally equal to t at type A in context Γ (Lemma 5). The construction is similar to [ACD07] but has Δ as additional parameter.

We write $\Gamma' \leq \Gamma$ if Γ' is a well-formed extension of Γ. By induction on $X : \mathsf{Set}_i$ we define the relation

$$\Gamma \vdash t : C \circledR \Delta \vdash d : X$$

between well-typed terms $\Gamma \vdash t : C$ and semantic objects $\Delta \vdash d : X$.

$$\begin{array}{l}
\Gamma \vdash r : C \circledR \Delta \vdash f : \mathsf{Fun}\, X F \iff \\
\qquad \Gamma \vdash C = \mathsf{Fun}\, A\, \lambda x B : \mathsf{Set}_i \text{ for some } A, B \text{ and} \\
\qquad \Gamma \vdash A : \mathsf{Set}_i \circledR \Delta \vdash X : \mathsf{Set}_i \text{ and} \\
\qquad \Gamma' \vdash r\, s : B(\sigma_{\mathsf{id}}, x{=}s) \circledR \Delta' \vdash f\, d : F\, d \text{ for all } \Gamma' \leq \Gamma, \Delta' \leq \Delta \\
\qquad\qquad \text{and } \Gamma' \vdash s : A \circledR \Delta' \vdash d : X
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash r : C \circledR \Delta \vdash \mathsf{Fun}\, X F : \mathsf{Set}_i \iff \\
\qquad \Gamma \vdash C = \mathsf{Set}_i : \mathsf{Set}_{i+1} \\
\qquad \Gamma \vdash r = \mathsf{Fun}\, A\, \lambda x B : \mathsf{Set}_i \text{ for some } A, B \text{ and} \\
\qquad \Gamma \vdash A : \mathsf{Set}_i \circledR \Delta \vdash X : \mathsf{Set}_i \text{ and} \\
\qquad \Gamma' \vdash B(\sigma_{\mathsf{id}}, x{=}s) : \mathsf{Set}_i \circledR \Delta' \vdash F\, d : \mathsf{Set}_i \text{ for all } \Gamma' \leq \Gamma, \Delta' \leq \Delta \\
\qquad\qquad \text{and } \Gamma \vdash s : A \circledR \Delta \vdash d : X
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash r : C \circledR \Delta \vdash d : X \iff \Delta \vdash d \searrow v \Uparrow X \text{ and } \Gamma \vdash r = v : C \\
\qquad \text{for } X \text{ neutral or } X = \mathsf{Set}_i \text{ and } d \neq \mathsf{Fun}\, Y F
\end{array}$$

The logical relation is closed under weakening of contexts (both the syntactic, Γ, and the semantic one, Δ) and under judgmental and Kripke model equality, meaning one can always trade expressions and values for equals without violating the relation.

Lemma 5 (Into and out of the logical relation / escape lemma). *Let $\Gamma \vdash C : \mathsf{Set}_i \circledR \Delta \vdash X : \mathsf{Set}_i$.*

1. *(In:) If $\Gamma \vdash u : C$ and $\Delta \vdash e \searrow u \Downarrow X$ then $\Gamma \vdash u : C \circledR \Delta \vdash e : X$.*
2. *(Out:) If $\Gamma \vdash r : C \circledR \Delta \vdash d : X$ then $\Delta \vdash d \searrow v \Uparrow X$ for some v with $\Gamma \vdash r = v : C$.*
3. *(Out-type:) $\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow j$ for $j \leq i$ and $\Gamma \vdash C = V : \mathsf{Set}_i$.*

Proof. By induction on $X : \mathsf{Set}_i$.

Fundamental theorem. We relate substitutions $\Gamma' \vdash \sigma : \Gamma$ to environments $\Delta \vdash \rho : \Gamma$ by the following definition:

$$\begin{aligned} \Gamma' \vdash \sigma \circledR \Delta \vdash \rho :: \Gamma \iff & \text{ for all } x \in \mathsf{dom}(\Gamma), \\ & \Gamma' \vdash x\sigma : \Gamma(x)\sigma \circledR \Delta \vdash \rho(x) : \Gamma(x)\rho \end{aligned}$$

By induction on the length of Γ, we define the propositions $\Gamma \Vvdash J$ as follows:

$$\begin{array}{lll} \Vvdash & \iff & \text{true} \\ \Gamma, x\!:\!A \Vvdash & \iff & \Gamma \Vvdash A \\ \Gamma \Vvdash A & \iff & \text{either } A = \mathsf{Set}_i \text{ and } \Gamma \Vvdash \text{ or } \Gamma \Vvdash A : \mathsf{Set}_i \text{ for some } i \\ \Gamma \Vvdash t : A & \iff & \Gamma \Vvdash t = t : A \\ \Gamma \Vvdash t = t' : A & \iff & \Gamma \Vvdash \text{ and} \\ \multicolumn{3}{l}{\quad \Gamma' \vdash t\sigma : A\sigma \circledR \Delta \vdash t'\rho : A\rho \text{ for all } \Gamma' \vdash \sigma \circledR \Delta \vdash \rho :: \Gamma.} \end{array}$$

$$\begin{array}{lll} \Gamma \Vvdash \tau : \Gamma_0 & \iff & \Gamma \Vvdash \tau = \tau : \Gamma_0 \\ \Gamma \Vvdash \tau = \tau' : \Gamma_0 & \iff & \Gamma \Vvdash \text{ and } \Gamma_0 \Vvdash \text{ and} \\ \multicolumn{3}{l}{\quad \Gamma' \vdash \tau \circ \sigma \circledR \Delta \vdash \tau'\rho :: \Gamma_0 \text{ for all } \Gamma' \vdash \sigma \circledR \Delta \vdash \rho :: \Gamma.} \end{array}$$

Theorem 2 (Fundamental theorem of logical relations). *If $\Gamma \vdash J$ then $\Gamma \Vvdash J$.*

Proof. By induction on $\Gamma \vdash J$.

Lemma 6. $\Gamma \vdash \sigma_{\mathsf{id}} \circledR (\Gamma\rho_{\mathsf{id}}) \vdash \rho_{\mathsf{id}} :: \Gamma$.

Proof. We have to show $\Gamma \vdash x : \Gamma(x) \circledR \Gamma\rho_{\mathsf{id}} \vdash x : \Gamma(x)\rho_{\mathsf{id}}$ for all $x \in \mathsf{dom}(\Gamma)$. This holds by Lemma 5 since $\Gamma\rho_{\mathsf{id}} \vdash x \searrow x \Downarrow (\Gamma\rho_{\mathsf{id}})(x)$.

Corollary 2 (Soundness of NbE). *If $\Gamma \vdash t : A$ then there is some v such that $\Gamma\rho_{\mathsf{id}} \vdash t\rho_{\mathsf{id}} \searrow v \Uparrow A\rho_{\mathsf{id}}$ and $\Gamma \vdash t = v : A$.*

7 Completeness of Algorithmic Equality

In this section, we conclude the completeness of the equality algorithm from the completeness of NbE. In particular, if two values reify to the same normal form, they are algorithmically equal. Some care has to be paid to the case of functional values. It could be that f reifies to λxv since $f \cdot x$ reifies to v and f' reifies to $\lambda x'v' =_\alpha \lambda xv$ since $f' \cdot x'$ reifies to v'. Now we want to conclude that f and f' are algorithmically equal, which requires $f \cdot \mathsf{x}_\Delta$ equal to $f' \cdot \mathsf{x}_\Delta$. But the induction hypothesis is not applicable since x, x', x_Δ might be different variables, hence, $f \cdot x$ and $f \cdot \mathsf{x}_\Delta$ are different objects. If our values are actually normal *expressions*, we could use plain old α-conversion and rename the variables. However, we are dealing with values in an arbitrary syntactical applicative structure D!

We restrict the class of possible models to those that admit renaming of free variables. This is a sensible restriction since values should be *parametric* in the names of their free variables—case distinction on the name of a identifier is not a desirable program behavior.

Renamings. Let π be a bijective map from variables to variables. We assume a renaming operations $d\pi$ on values $d \in \mathsf{D}$ with the following properties:

$$\begin{array}{lll} \textsc{ren-c} & c\pi & = c \\ \textsc{ren-var} & x\pi & = \pi(x) \\ \textsc{ren-app} & (f \cdot d)\pi & = f\pi \cdot d\pi \\ \textsc{ren-eval} & (t\rho)\pi & = t(\rho\pi) \end{array}$$

Renaming can be defined for many syntactical applicative structures D: term models, explicit substitutions, closures, even Scott models which evaluate an abstraction $(\lambda x t)\rho$ to an actual function $h : \mathsf{D} \to \mathsf{D}$ with $h(d) = t(\rho, x = d)$. Setting $(h\pi)(d) = (h(d\pi^{-1}))\pi$ we have $(h(d))\pi = (h\pi)(d\pi)$ [Pit06].

We define renaming of contexts $\pi(\Delta)$ by

$$\begin{array}{lll} \textsc{ren-cxt-empty} & \pi(\diamond) & = \diamond \\ \textsc{ren-cxt-ext} & \pi(\Delta, x\!:\!X) & = \pi(\Delta), \pi(x)\!:\!X\pi \end{array}$$

Remark 1. This is not to be confused with the operation $\Delta\pi$ which is composition defined by $(\Delta\pi)(x) = \Delta(x)\pi$. We have $\pi(\Delta)(x\pi) = \Delta(x)\pi$. Thus, $\pi(\Delta) = \pi^{-1}\Delta\pi$ is a conjugation.

Lemma 7 (Completeness of algorithmic equality w. r. t. reification). *Let $\Delta = x_1\!:\!X_1, \dots x_n\!:\!X_n$ be a semantic context and π a permutation of names assigning the ith de Bruijn level to variable x_i ($\pi(x_i) = \mathsf{x}_i$ for $i = 1..n$).*

1. *If $\Delta \vdash e \searrow u \Downarrow X$ and $\Delta' \vdash e' \searrow u \Downarrow X'$ then $\pi(\Delta) \vdash e\pi = e'\pi \Downarrow X\pi$.*
2. *If $\Delta \vdash d \searrow v \Uparrow X$ and $\Delta' \vdash d' \searrow v \Uparrow X'$ then $\pi(\Delta) \vdash d\pi = d'\pi \Uparrow X\pi$.*
3. *If $\Delta \vdash X \searrow V \Uparrow \mathsf{Set} \rightsquigarrow i$ and $\Delta' \vdash X' \searrow V \Uparrow \mathsf{Set} \rightsquigarrow i'$ then $\pi(\Delta) \vdash X\pi = X'\pi \Uparrow \mathsf{Set} \rightsquigarrow \max(i, i')$.*

Proof. Simultaneously by induction on the first derivation.

A name permutation π can be viewed as an environment, taking a variable x to $\pi(x)$. This explains the notation $t\pi$, and $\pi(\Gamma)$ which satisfies $\pi(\Gamma)(x\pi) = \Gamma(x)\pi$. Let $\pi_{(x_1:A_1,\dots x_n:A_n)}(x_i) = \mathsf{x}_i$ for $i = 1..n$.

Corollary 3 (Completeness of algorithmic equality). *If $\Gamma \vdash t = t' : A$ then $\pi_\Gamma(\Gamma) \vdash t\pi_\Gamma = t'\pi_\Gamma \Uparrow A\pi_\Gamma$.*

Proof. We have $\Gamma\rho_{\mathsf{id}} \vdash t\rho_{\mathsf{id}}, t'\rho_{\mathsf{id}} \searrow \_ \Uparrow A\rho_{\mathsf{id}}$ by Cor. 1, thus, the corollary follows from Lemma 7 with $\rho_{\mathsf{id}}\pi_\Gamma = \pi_\Gamma$.

8 Strong Semantic Typing and Soundness of Algorithmic Equality

In this section, we tackle the second problem: soundness of the equality algorithm. We show that if two values are algorithmically equal, then they reify to the same normal form. Rule AQ-FUN gives us a hard time:

$$\textsc{aq-fun}\ \frac{\Delta \vdash e = e' \Downarrow \mathsf{Fun}\, X F \qquad \Delta \vdash d = d' \Uparrow X}{\Delta \vdash e\, d = e'\, d' \Downarrow F \cdot d}$$

Using the induction hypothesis, we can show $\Delta \vdash e'\,d' \searrow u\,v \Downarrow F \cdot d'$ but we need $F \cdot d$! The fact that d and d' reify to the same normal form does not help us here, we need a stronger induction hypothesis.

Again, soundness of an algorithmic equality on *syntax* is usually trivial: one simply shows that each algorithmic rule is an instance of an inference rule. When trying to apply this intuition to our semantic typing and equality, the Kripke model, one realizes that it lacks a substitution principle: *Judgements remain valid when substituting a term of the right type for a variable.* In the following we will equip our semantics with a substitution principle by brute force! We define *strong* semantic judgements (a *super* Kripke model) to hold if the weak semantic judgements (Kripke model) hold for all well-typed valuations of free variables (cf. hypothetical judgements [CPT05]). As precondition, we need a model in which we can reevaluate values in a new environment.

Reevaluation. For the following, impose more conditions on the model D. It must be equipped with a reevaluation function $d\theta$ of value d in environment θ, satisfying the following laws:

REEVAL-ID	$d\,\rho_{\mathsf{id}}$	$= d$
REEVAL-C	$c\theta$	$= c$
REEVAL-VAR	$x\theta$	$= \theta(x)$
REEVAL-FUN-E	$(f \cdot d)\theta$	$= f\theta \cdot d\theta$
REEVAL-ESUBST	$(t\rho)\theta$	$= t(\rho\theta)$

Reevaluation is easy to define on the syntactical applicative structure of closures; simply add $([\lambda xt]\rho)\theta = [\lambda xt](\rho\theta)$ and $(e\,d)\theta = e\theta \cdot d\theta$ to the above laws.

For Scott models, it is a bit more problematic. If $f \in \mathsf{D} \to \mathsf{D}$, we can set $(f\theta)(d) = (f(x))((\theta, x\!=\!d)$ for a fresh variable x. Freshness can be defined if we construct D as a nominal set [Shi05]. However, not all continuous functions f will fulfill REEVAL-FUN-E: for $(f(d))\theta = (f(x))(\theta, x\!=\!d\theta)$ to hold f must treat variables parametrically. For example, the function $f(d) = d$ if d not a variable and $f(x) = \mathsf{Set}_0$ for x a variable is not parametric and, thus, not compatible with reevaluation. The formulation of suitable parametricity conditions and the proof of parametricity for all values in the Kripke model remains open.

8.1 Super Kripke Model

Let Θ range over semantic contexts and let

$$\Delta \vdash \rho = \rho' : \Theta \iff \forall x \in \mathsf{dom}(\Theta).\ \Delta \vdash x\rho = x\rho' : \Theta(x)\rho$$

We write $\Delta \vdash \rho : \Theta$ iff $\Delta \vdash \rho = \rho : \Theta$. We define strong semantic equality by

$$\Theta \models d = d' : X \iff \forall \Delta \vdash \rho = \rho' : \Theta.\ \Delta \vdash d\rho = d'\rho' : X\rho$$

Let $\Theta \models X = X'$ iff $\Theta \models X = X' : \mathsf{Set}_i$ for some i. We write $\Theta \models X$ for $\Theta \models X = X$ and $\Theta \models d : X$ for $\Theta \models d = d : X$, and say d is semantically strongly typed of type X in context Θ.

Since $\Delta \vdash \rho_{\mathsf{id}} = \rho_{\mathsf{id}} : \Delta$, strong semantic equality $\Delta \models d = d' : X$ implies weak semantic equality $\Delta \vdash d = d' : X$.

Lemma 8 (Admissible rules). *The following implications, written as rules, hold for strong semantic equality.*

$$\frac{}{\Theta \models x = x : \Theta(x)}\; x \in \mathsf{dom}(\Theta)$$

$$\frac{\Theta \models f = f' : \mathsf{Fun}\, X F \qquad \Theta \models d = d' : X}{\Theta \models f \cdot d = f' \cdot d' : F \cdot d}$$

$$\frac{\begin{array}{c}\Theta \models \mathsf{Fun}\, X F \qquad \Theta \models f : \mathsf{Fun}\, X F \qquad \Theta \models f' : \mathsf{Fun}\, X F \\ \Theta, x{:}X \models f \cdot x = f' \cdot x : F \cdot x\end{array}}{\Theta \models f = f' : \mathsf{Fun}\, X F}$$

$$\frac{\begin{array}{c}\Theta \models \mathsf{Fun}\, X F,\ \mathsf{Fun}\, X' F' : \mathsf{Set}_i \\ \Theta \models X = X' : \mathsf{Set}_i \qquad \Theta, x{:}X \models F \cdot x = F' \cdot x : \mathsf{Set}_i\end{array}}{\Theta \models \mathsf{Fun}\, X F = \mathsf{Fun}\, X' F' : \mathsf{Set}_i}$$

$$\frac{\Theta \models d = d' : X \qquad \Theta \models X = X'}{\Theta \models d = d' : X'}$$

Proof. The soundness of the application rule relies on distributivity of valuation over application, $(f \cdot d)\rho = f\rho \cdot d\rho$.

Semantic context equality $\models \Theta = \Theta'$ is given inductively by

$$\frac{}{\models \diamond = \diamond} \qquad \frac{\models \Theta = \Theta' \qquad \Theta \models X = X' : \mathsf{Set}_i}{\models \Theta, x{:}X = \Theta', x{:}X'}\, x \notin \mathsf{dom}(\Theta)$$

We write $\models \Theta$ for $\models \Theta = \Theta$.

Lemma 9 (Soundness of algorithmic equality). *Let* $\models \Delta$.

1. *If* $\Delta \vdash e = e' \Downarrow X$ *then* $\Delta \models X$ *and* $\Delta \models e = e' : X$.
2. *If* $\Delta \vdash d = d' \Uparrow X$ *and* $\Delta \models X$ *and* $\Delta \models d, d' : X$ *then* $\Delta \models d = d' : X$.
3. *If* $\Delta \vdash X = X' \Uparrow \mathsf{Set} \rightsquigarrow i$ *and* $\Delta \models X, X' : \mathsf{Set}_i$ *then* $\Delta \vdash X = X' : \mathsf{Set}_i$.

Proof. Simultaneously by induction on the derivation of algorithmic equality using the admissible rules.

8.2 Strong Validity of Syntactic Typing

In the following, we establish that our inference rules are also valid in the super Kripke model. However, no new inductive proof is needed. We have already shown that the rules are valid in the weak semantics (Kripke model) under all weakly typed environments. This is actually equivalent to being valid in the strong semantics (super Kripke model) under all strongly typed environments.

We define strongly typed environments by

$$\Delta \models \rho = \rho' : \Gamma \iff \forall x \in \mathsf{dom}(\Gamma).\, \Delta \models \rho(x) = \rho'(x) : \Gamma(x)\rho.$$

Lemma 10 (Composing strongly and weakly typed environments). *If* $\Theta \models \rho = \rho' : \Gamma$ *and* $\Delta \vdash \theta = \theta' : \Theta$ *then* $\Delta \vdash \rho\theta = \rho'\theta' : \Gamma$.

Define $\Gamma \mathrel{|\!\models} J$, meaning that $\Gamma \vdash J$ is valid in the super Kripke model, as follows:

$$\begin{array}{lcl}
\diamond \mathrel{|\!\models} & :\Longleftrightarrow & \text{true} \\
\Gamma, x{:}A \mathrel{|\!\models} & :\Longleftrightarrow & \Gamma \mathrel{|\!\models} A \\[1ex]
\Gamma \mathrel{|\!\models} A & :\Longleftrightarrow & \text{either } A = \mathsf{Set}_i \text{ and } \Gamma \mathrel{|\!\models} \text{ or } \Gamma \mathrel{|\!\models} A : \mathsf{Set}_i \text{ for some } i \\[1ex]
\Gamma \mathrel{|\!\models} t : A & :\Longleftrightarrow & \Gamma \mathrel{|\!\models} t = t : A \\
\Gamma \mathrel{|\!\models} t = t' : A & :\Longleftrightarrow & \Gamma \mathrel{|\!\models} A \text{ and } \forall \Delta \models \rho = \rho' : \Gamma.\ \Delta \models t\rho = t'\rho' : A\rho \\[1ex]
\Gamma \mathrel{|\!\models} \sigma : \Gamma_0 & :\Longleftrightarrow & \Gamma \mathrel{|\!\models} \sigma = \sigma : \Gamma_0 \\
\Gamma \mathrel{|\!\models} \sigma = \sigma' : \Gamma_0 & :\Longleftrightarrow & \Gamma \mathrel{|\!\models} \text{ and } \Gamma_0 \mathrel{|\!\models} \text{ and} \\
& & \qquad \forall \Delta \models \rho = \rho' : \Gamma.\ \Delta \models \sigma\rho = \sigma'\rho' : \Gamma_0
\end{array}$$

Lemma 11 (Validity: weak implies strong). *If* $\Gamma \Vdash J$ *then* $\Gamma \mathrel{|\!\models} J$.

Proof. The hypothesis is $\forall \Delta \vdash \rho = \rho' : \Gamma.\ \Delta \vdash t\rho = t'\rho' : A\rho$. Assume $\Theta \models \rho = \rho' : \Gamma$ and $\Delta \vdash \theta = \theta' : \Theta$ and show $\Delta \vdash t\rho\theta = t'\rho'\theta' : A\rho\theta$. By Lemma 10 $\Delta \vdash \rho\theta = \rho'\theta' : \Gamma$, hence our goal follows by assumption.

Corollary 4. *If* $\Gamma \vdash J$ *then* $\Gamma \mathrel{|\!\models} J$.

Lemma 12. $\Gamma\rho_{\mathsf{id}} \models \rho_{\mathsf{id}} = \rho_{\mathsf{id}} : \Gamma$.

Corollary 5. *If* $\Gamma \vdash t : A$ *then* $\Gamma\rho_{\mathsf{id}} \models t\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$.

Putting things together:

Theorem 3 (Soundness of algorithmic equality). *If* $\Gamma \vdash t, t' : A$ *and* $\Gamma\rho_{\mathsf{id}} \vdash t\rho_{\mathsf{id}} = t'\rho_{\mathsf{id}} \Uparrow A\rho_{\mathsf{id}}$ *then* $\Gamma \vdash t = t' : A$.

Proof. We have $\Gamma \vdash t : A \circledR \Gamma\rho_{\mathsf{id}} \vdash t\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$ and $\Gamma \vdash t' : A \circledR \Gamma\rho_{\mathsf{id}} \vdash t'\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$. Also $\models \Gamma\rho_{\mathsf{id}}$, $\Gamma\rho_{\mathsf{id}} \models A\rho_{\mathsf{id}}$, $\Gamma\rho_{\mathsf{id}} \models t\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$, and $\Gamma\rho_{\mathsf{id}} \models t'\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$. By semantic soundness of the algorithm, $\Gamma\rho_{\mathsf{id}} \models t\rho_{\mathsf{id}} = t'\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$ which implies $\Gamma\rho_{\mathsf{id}} \vdash t\rho_{\mathsf{id}} = t'\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$, hence $\Gamma \vdash t : A \circledR \Gamma\rho_{\mathsf{id}} \vdash t'\rho_{\mathsf{id}} : A\rho_{\mathsf{id}}$. This entails $\Gamma \vdash t = t' : A$.

9 On Termination

What remains to show is termination of the algorithmic equality: Given two terms of the same type, the algorithm terminates on their values. We have already seen that the values of well-typed terms are reifiable, hence the NbE-algorithm, which compares the results of reification, is terminating. We would like to extend this result to algorithmic equality, which performs reification incrementally. One would expect a statement similar to Lemma 7:

> If $\Delta \vdash d \searrow_{\_} \Uparrow X$ and $\Delta \vdash d' \searrow_{\_} \Uparrow X$ then the query $\Delta \vdash d = d' \Uparrow X$ terminates.

Generalizing this statement to the mutually defined notions of reification and algorithmic equality ($\Downarrow$ and $\Uparrow \mathsf{Set}$), we see that the proof fails since during reification of d', context and type diverge from Δ and X. (Similar considerations lead us to the definition of $\circledS$ in Section 5.2.)

The skeleton of the algorithmic derivation is already determined by the derivation of $\Delta \vdash d \searrow \_ \Uparrow X$. Yet we have to show that the application $f' \cdot \mathsf{x}_\Delta$ is terminating in AQ-EXT. Somehow we have to exploit that d' originates from a well-typed term, thus, $\Delta \vdash d' : X$ and even $\Delta \models d : X$ both hold. But it is unclear how to make use of these facts in a termination proof.

10 Conclusion and Related Work

We have presented a bidirectional incremental $\beta\eta$-equality algorithm for a dependent type theory with predicative universes and verified it using NbE-techniques. The algorithm is formulated with respect to an abstract representation of values which supports several implementations. In Sec. 8 we had to exclude the representation via higher order abstract syntax, which was used in an early version of Agdalight, a predecessor of Agda 2. In the future we want to explore how to generalize our proof so that we also cover this implementation technique. Furthermore, we want to close the gap and prove termination of the algorithm as well as soundness and completeness of type checking.

Some complications in the verification vanish if we restrict to a concrete, term-like implementation of values, for instance, closures [Coq96, AC08]. Closures are a special case of explicit substitutions, hence, soundness of algorithmic equality is trivial: each algorithmic rule can be replaced by a sequence of declarative rules. Termination also becomes apparent: algorithmic equality works only on well-typed terms, which are normalizing.

Related work. The current proof uses similar techniques to those in our previous work on NbE [ACD07]. However, there are several differences, in order to deal with contextual reification we here need a Kripke model instead of a plain PER model.

Goguen [Gog94] proves decidability of UTT using typed operational semantics. He treats η, universes, and even inductive types and a impredicative universe of propositions. Showing soundness and completeness of his syntactic Kripke model he establishes subject reduction, confluence, and strong normalization, which imply decidability. However, he is not concerned about particular algorithms. Since his approach is based on η-reduction instead of η-expansion, it is not clear whether it scales to a unit type with extensional equality.

Harper and Pfenning [HP05] present an incremental bidirectional $\beta\eta$-equality algorithm for LF using erasure of dependencies; this does not extend to large eliminations. Chapman, Altenkirch, and McBride [CAM07] share their algorithm with us. They describe an implementation, but no verification.

Grégoire and Leroy [GL02] have implemented an incremental β-conversion test for the Calculus of Inductive Constructions based on "normalization by execution". Values are computed by compiling (open!) expression to Caml byte code. The result of executing the code is then *read back* to a β-normal expression. The soundness of this efficient form of evaluation has been formally verified in Coq. We expect that our and their work can be combined to obtain an efficient $\beta\eta$-conversion test. To this end, one needs to instantiate the syntactical applicative structure D to machine code and define contextual reification of code, similar to Grégoire and Leroy's *read back* function.

Acknowledgments. Thanks to the anonymous referees who gave insightful comments and pointed to the problem of termination.

References

[AAD07] Abel, A., Aehlig, K., Dybjer, P.: Normalization by evaluation for Martin-Löf type theory with one universe. In: Fiore, M. (ed.) Proc. of the 23rd Conf. on the Mathematical Foundations of Programming Semantics (MFPS XXIII). Electr. Notes in Theor. Comp. Sci, vol. 173, pp. 17–39. Elsevier, Amsterdam (2007)

[AC07] Abel, A., Coquand, T.: Untyped algorithmic equality for Martin-Löf's logical framework with surjective pairs. Fundam. Inform. 77(4), 345–395 (2005); TLCA 2005 special issue

[AC08] Altenkirch, T., Chapman, J.: Big step normalisation. Draft, available on the authors' homepages (2008)

[ACD07] Abel, A., Coquand, T., Dybjer, P.: Normalization by evaluation for Martin-Löf Type Theory with typed equality judgements. In: Proc. of the 22nd IEEE Symp. on Logic in Computer Science (LICS 2007), pp. 3–12. IEEE Computer Soc. Press, Los Alamitos (2007)

[ACD08] Abel, A., Coquand, T., Dybjer, P.: A semantic $\beta\eta$-equality algorithm for Martin-Löf Type Theory (extended version). Technical report, Ludwig-Maximilians-University Munich (2008), `http://www.tcs.ifi.lmu.de/~abel/semEqTR.pdf`

[Bar84] Barendregt, H.: The Lambda Calculus: Its Syntax and Semantics. North Holland, Amsterdam (1984)

[BL84] Bruce, K.B., Longo, G.: On combinatory algebras and their expansions. Theor. Comput. Sci. 31, 31–40 (1984)

[BS91] Berger, U., Schwichtenberg, H.: An inverse to the evaluation functional for typed λ-calculus. In: Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science, Amsterdam, pp. 203–211 (July 1991)

[CAM07] Chapman, J., Altenkirch, T., McBride, C.: Epigram reloaded: a standalone typechecker for ETT. In: van Eekelen, M.C.J.D. (ed.) Revised Selected Papers from the 6th Symp. on Trends in Functional Programming, TFP 2005, Trends in Functional Programming, Intellect, vol. 6, pp. 79–94 (2007)

[Coq94] Coquand, C.: From semantics to rules: A machine assisted analysis. In: Meinke, K., Börger, E., Gurevich, Y. (eds.) Proc. of the 7th Wksh. on Computer Science Logic, CSL 1993. LNCS, vol. 832, pp. 91–105. Springer, Heidelberg (1994)

[Coq96] Coquand, T.: An algorithm for type-checking dependent types. In: Mathematics of Program Construction. Selected Papers from the Third International Conference on the Mathematics of Program Construction, Kloster Irsee, Germany, July 17–21, 1995. Science of Computer Programming, vol. 26, pp. 167–177. Elsevier Science, Amsterdam (1996)

[CPT05] Coquand, T., Pollack, R., Takeyama, M.: A logical framework with dependently typed records. Fundam. Inform. 65(1-2), 113–134 (2005)

[Dan99] Danvy, O.: Type-directed partial evaluation. In: Hatcliff, J., Mogensen, T.Æ., Thiemann, P. (eds.) Partial Evaluation – Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998. LNCS, vol. 1706, pp. 367–411. Springer, Heidelberg (1999)

[Dyb96] Dybjer, P.: Internal type theory. In: Berardi, S., Coppo, M. (eds.) TYPES 1995. LNCS, vol. 1158, pp. 120–134. Springer, Heidelberg (1996)

[Dyb00] Dybjer, P.: A general formulation of simultaneous inductive-recursive definitions in type theory. The Journal of Symbolic Logic 65(2), 525–549 (2000)

[GL02] Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: Proc. of the 7th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2002). SIGPLAN Notices, vol. 37, pp. 235–246. ACM Press, New York (2002)

[Gog94] H. Goguen.: *A Typed Operational Semantics for Type Theory.* PhD thesis, University of Edinburgh. Available as LFCS Report ECS-LFCS-94-304 (August 1994)

[HP05] Harper, R., Pfenning, F.: On equivalence and canonical forms in the LF type theory. ACM Transactions on Computational Logic 6(1), 61–101 (2005)

[INR07] INRIA. The Coq Proof Assistant, Version 8.1. INRIA (2007), `http://coq.inria.fr/`

[Ler06] Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Morrisett, J.G., Jones, S.L.P. (eds.) Proc. of the 33rd ACM Symp. on Principles of Programming Languages, POPL 2006, pp. 42–54. ACM Press, New York (2006)

[ML75] Martin-Löf, P.: About models for intuitionistic type theories and the notion of definitional equality. In: Kanger, S. (ed.) Proceedings of the 3rd Scandinavian Logic Symposium, pp. 81–109 (1975)

[ML92] P. Martin-Löf.: Substitution calculus. Lecture in Göteborg (November 1992) (unpublished)

[Nor07] Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-41296 Göteborg, Sweden (September 2007)

[Pit06] Pitts, A.M.: Alpha-structural recursion and induction. Journal of the ACM 53, 459–506 (2006)

[Pol94] Pollack, R.: Closure under alpha-conversion. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 313–332. Springer, Heidelberg (1994)

[PS99] Pfenning, F., Schürmann, C.: System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)

[PT98] Pierce, B.C., Turner, D.N.: Local type inference. In: POPL 98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California (1998)

[Shi05] Shinwell, M.: The Fresh Approach: Functional Programming with Names and Binders. PhD thesis, University of Cambridge (2005)

[SS08] Schürmann, C., Sarnat, J.: Structural logical relations. In: Pfenning, F. (ed.) Proc. of the 23nd IEEE Symp. on Logic in Computer Science (LICS 2008) (2008)

The Capacity-C Torch Problem

Roland Backhouse

School of Computer Science University of Nottingham, Nottingham NG8 1BB, England
rcb@cs.nott.ac.uk

Abstract. The torch problem (also known as the bridge problem or the flashlight problem) is about getting a number of people across a bridge as quickly as possible under certain constraints. Although a very simply stated problem, the solution is surprisingly non-trivial. The case in which there are just four people and the capacity of the bridge is two is a well-known puzzle, widely publicised on the internet. We consider the general problem where the number of people, their individual crossing times and the capacity of the bridge are all input parameters. We present an algorithm that determines the shortest total crossing time; the number of primitive computations executed by the algorithm (i.e. the worst-case time complexity of the algorithm) is proportional to the square of the number of people.

Keywords: algorithm derivation, shortest path, dynamic programming, algorithmic problem solving.

The (capacity-C) torch problem is as follows.

> N people wish to cross a bridge. It is dark, and it is necessary to use a torch when crossing the bridge, but they only have one torch between them. The bridge is narrow and at most C people can be on it at any one time. The people are numbered from 1 thru N. Person i takes time $t.i$ to cross the bridge; when a group of people cross together they must all proceed at the speed of the slowest.
>
> Construct an algorithm that will get all N people across in the shortest time. Provide a clear justification that the algorithm does indeed find the shortest time.

The torch problem is an abstraction from a problem involving four people wishing to cross a bridge of capacity two and with specific concrete times. In this form, the problem is believed to have first appeared in 1981. Rote [3] gives a comprehensive bibliography.

The main interest in the torch problem is that what is "obvious" or "intuitive" is often wrong. For example, the "obvious" solution of letting the fastest person repeatedly accompany $C-1$ people across the bridge is wrong. (If $N=4$, $C=2$ and the travel times are 1, 1, 2 and 2, this solution takes time 7 whereas the

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 57–78, 2008.

shortest crossing time is 6[1].) Also, the "obvious" property that the shortest time is achieved when the number of crossings is minimised is incorrect. (If $N = 5$, $C = 3$ and the travel times are 1, 1, 4, 4 and 4, the shortest time is 8, which is achieved using 5 crossings. The shortest time using 3 crossings is 9.) It is not difficult to determine an *upper bound* on the crossing time, even in the general case. Nor is it difficult to provide counterexamples to incorrect solutions. The difficulty is to establish an irrefutable *lower bound* on the crossing time. A proper solution to the problem poses a severe test of our standards of proof.

In our solution, we assume that the people are ordered so that $t.i < t.j$ if $i < j$. If the given times are such that $t.i = t.j$ for some i and j, where $i < j$, we can always consider pairs $(t.i\,,\, i)$, where i ranges over people, ordered lexicographically. Renaming the crossing "times" to be such pairs, we obtain a total ordering on times with the desired property[2]. We also assume that N is at least $C+1$. (When N is at most C, it is obvious that exactly one crossing gives the optimal solution. When N is at least $C+1$, more than one crossing is required.)

For brevity, some of the more straightforward proofs at the beginning of the paper. A full version of the paper, which includes the details of all proofs, is available from the author's website.

1 Outline Strategy

An outline of our solution is as follows.

We call a sequence of crossings that gets everyone across in accordance with the rules a *putative sequence*. We will say that one putative sequence *subsumes* another putative sequence if the time taken by the first is at most the time taken for the second. Note that the subsumes relation is reflexive (every putative sequence subsumes itself) and transitive (if putative sequence a subsumes putative sequence b and putative sequence b subsumes putative sequence c then putative sequence a subsumes putative sequence c). An *optimal* sequence is a putative sequence that subsumes all putative sequences. A putative sequence is *suboptimal* if it is not optimal. The problem is to find an optimal sequence.

Recall that, when crossing the bridge, the torch must always be carried. This means that crossings, both of groups of people and of each individual person, alternate between "forward"and "return" trips, where a *forward trip* is a crossing in the desired direction, and a *return trip* is a crossing in the opposite direction.

A *regular forward trip* means a crossing in the desired direction *made by at least two people*, and a *regular return trip* means a trip in the opposite direction *made by exactly one person*. A *regular sequence* is a putative sequence that consists entirely of regular forward and return trips.

[1] Our examples are chosen so that it is easy for the reader to discover the fastest crossing time. Of course, the examples in puzzle books are deliberately chosen to make it difficult.

[2] Strictly, we also need to extend addition to pairs. Defining $(t, i)+(u, j)$ to be $(t+u\,,\, i{\downarrow}j)$ guarantees the appropriate algebraic structure, in particular distributivity of addition over minimum [2].

The first step (lemma 1) is to show that every optimal putative sequence is regular. The significance of this is threefold. First, it means that the search space for an optimal solution is finite. (This is because a forward trip followed by a return trip reduces the number of people at the start; hence there are at most $N-1$ forward trips in any regular sequence.) Second, the time taken by a regular putative sequence can be evaluated knowing only which forward trips are made. (Knowing the bag of forward trips, it is easy to determine how many times each person makes a return trip. This is because each person makes one fewer return trips than forward trips. In this way, the time taken for the return trips can be calculated.) Finally, and most importantly, knowing just the bag of forward trips in a regular putative sequence is sufficient to reconstruct a regular putative sequence. This is proved in theorem 1. Since all such sequences take the same total time, we can replace the problem of finding an optimal sequence of forward and return trips by the problem of finding an optimal bag of forward trips.

Finding an optimal bag of forward trips begins by establishing a number of lemmas with the goal of reducing the size of the search space. Subsequently, we can formulate the problem as, essentially, a shortest-path problem on an acyclic graph. More precisely, we present a collection of equations each of which corresponds to a component of an algorithm for non-deterministically constructing a bag of forward trips. By calculating the (unique) solution to these equations, we can resolve the non-determinacy in the construction and so obtain an optimal bag of forward trips. Then theorem 1 is applied to obtain a regular sequence that optimises the total travel time. The number of terms in the collection of equations is quadratic in the number of people and cubic in the capacity of the bridge, from which we can deduce the worst-case solution time.

2 Terminology

Let us suppose a putative sequence is given. By extracting just the forward trips in the sequence and ignoring the order in which they are made, we obtain a bag (multiset) of non-empty sets. We use F to denote such a bag. Note that a bag is a set with multiplicities. By a slight abuse of notation, we write $T{\in}F$ and call T an *element* of F if T is an element of the set underlying bag F; we also write $\#_F T$ for the multiplicity of T in the bag F. The bag is completely defined by listing its elements together with their multiplicities.

Since everyone must cross at some time, the bag F satisfies the property that

$$\langle \forall i \,:\, 1 \le i \le N \,:\, \langle \exists T : T{\in}F : i{\in}T \rangle \rangle \,. \tag{1}$$

Also, since each forward trip is non-empty and the capacity of the bridge is C,

$$\langle \forall T \,:\, T{\in}F \,:\, 1 \le |T| \le C \rangle \,. \tag{2}$$

From the bag F, we can determine the number of times each individual makes a forward trip. This is given by the function f which is defined by

$$f_F.i \;=\; \langle \Sigma T : i{\in}T : \#_F T \rangle \,. \tag{3}$$

The number of times that each person returns is given by the function r; since each person makes one more forward trip than return trip, we have

$$r_F.i \;=\; f_F.i - 1. \tag{4}$$

We distinguish two types of person:

(a) Someone who never makes a return trip is called a *settler*.

$$settler_F.i \;\equiv\; r_F.i = 0.$$

(b) Someone who does make a return trip is called a *nomad*.

$$nomad_F.i \;\equiv\; r_F.i > 0.$$

(Note that $settler_F.i \not\equiv nomad_F.i$.) We further subdivide the settlers into "pure" and "mixed" settlers.

(a) A *pure* settler is a settler who crosses with (only) other settlers.

$$pure_F.i \;\equiv\; \langle \forall T : T{\in}F \wedge i{\in}T : \langle \forall j : j{\in}T : settler_F.j \rangle\rangle\,.$$

(b) A *mixed* settler is a settler who crosses with at least one nomad.

$$mixed_F.i \;\equiv\; settler_F.i \;\wedge\; \langle \exists T{,}j : T{\in}F \wedge i{\in}T \wedge j{\in}T : nomad_F.j \rangle\,.$$

Correspondingly, we divide the forward trips into "pure", "mixed" and "nomadic".

(a) A *pure trip* is a forward trip in which everyone involved is a settler.

$$pure_F.T \;\equiv\; \langle \forall j : j{\in}T : settler_F.j \rangle\,.$$

(b) A *mixed trip* is a forward trip involving both settlers and nomads.

$$mixed_F.T \;\equiv\; \langle \exists i{,}j : i{\in}T \wedge j{\in}T : settler_F.i \wedge nomad_F.j \rangle\,.$$

(c) A *nomadic trip* is a forward trip in which everyone involved is a nomad.

$$nomadic_F.T \;\equiv\; \langle \forall i : i{\in}T : nomad_F.i \rangle\,.$$

A *full* trip is a forward trip in which C people cross. That is, the trip has no spare capacity.

$$full.T \;\equiv\; |T| = C.$$

The *leader* of a trip is the slowest person in the trip[3]:

$$lead.T \;=\; \langle \Uparrow i : i{\in}T : i \rangle\,.$$

Mixed and pure trips have multiplicity 1 in the bag F, and each settler is an element of exactly one element of F. It is therefore possible to define a function from settlers to people which identifies the slowest person in the trip made by the settler. Let us call this function $boss_F$. Then the defining property of $boss_F$ is

$$\langle \forall\, i{,}T : settler_F.i \wedge T{\in}F \wedge i{\in}T : lead.T = boss_F.i \rangle\,.$$

For nomads, the function $boss_F$ is undefined.

[3] The symbols $\Uparrow$ and $\Downarrow$ denote the maximum and minimum quantifiers, respectively; the symbols $\uparrow$ and $\downarrow$ denote the binary maximum and minimum operators.

3 Regular Sequences

Recall that a "regular" sequence is a sequence in which each forward trip involves at least two people and each return trip involves exactly one person. The following lemma restricts attention to just the regular sequences. The proof is omitted.

Lemma 1. Every putative sequence containing irregular trips is suboptimal. □

3.1 Scheduling Forward Trips

In view of lemma 1, we now consider bags of forward trips that correspond to regular putative sequences. Suppose F is such a bag. Then, with the function r defined by (4), the total time taken by the sequence is

$$\langle \Sigma T \;:\; T{\in}F \;:\; \langle \Uparrow i : i{\in}T : t.i\rangle \times \#_F T\rangle \;\;+\;\; \langle \Sigma i :: t.i \times r_F.i\rangle\,. \tag{5}$$

(Forward trip T takes time $\langle \Uparrow i : i{\in}T : t.i\rangle$ and has multiplicity $\#_F T$, and person i makes $r_F.i$ return trips each of which takes time $t.i$ because the sequence is regular.) Note that the total time is independent of the order in which the trips are scheduled.

Also, since the number of forward trips is $|F|$ and each return trip is undertaken by exactly one person,

$$|F| \;=\; \langle \Sigma i :: r_F.i\rangle + 1. \tag{6}$$

In a regular sequence, each forward trip involves at least 2 and at most C people, thus sharpening property (2):

$$\langle \forall T \;:\; T{\in}F \;:\; 2 \le |T| \le C\rangle\,. \tag{7}$$

Finally, as before, each person must cross at least once:

$$\langle \forall i \;:\; 1 \le i \le N \;:\; 1 \le f_F.i\rangle\,. \tag{8}$$

Crucially, given a bag of sets, F, such that properties (6), (7) and (8) hold of F, it is always possible to construct a regular putative sequence S such that the bag of forward trips in S is F. To establish this theorem, we first prove several properties relating the number of pure trips, the number of nomads and the number of non-pure trips in F.

To this end, we define the functions n ("number of nomads"), nc ("nomad count"), rc ("return count"), sc ("settler count"), np ("the number of non-pure trips") and pc ("pure-trip count") as follows. In the definitions, G is an arbitrary bag of sets, and T ranges over elements of G. The multiplicity of T in G is denoted by $\#_G T$.

$$n_G = \langle \Sigma i : nomad_G.i : 1\rangle \tag{9}$$

$$nc_G.T = \langle \Sigma i \;:\; nomad_G.i \wedge i{\in}T \;:\; 1\rangle \tag{10}$$

$$rc_G = \langle \Sigma i :: r_G.i\rangle \tag{11}$$

$$sc_G.T = \langle \Sigma i \;:\; settler_G.i \wedge i{\in}T \;:\; 1\rangle \tag{12}$$

$$np_G = \langle \Sigma T : \neg(pure_G.T) : \#_G T\rangle \tag{13}$$

$$pc_G = \langle \Sigma T : pure_G.T : 1\rangle \tag{14}$$

(Note that pure trips always have a multiplicity of 1.)

The following lemma and its corollary identify some straightforward relations between the various counts. Note that lemma 2 is true of all bags, whereas corollary 1 exploits a relation between the size of the bag and its return count.

Lemma 2. Suppose G is a bag of sets. Then

$$n_G = 0 \;\equiv\; rc_G = 0, \tag{15}$$

$$n_G \leq rc_G, \tag{16}$$

$$rc_G \;=\; \langle \Sigma T :: nc_G.T \times \#_G T \rangle \;-\; n_G, \tag{17}$$

$$np_G \neq 1. \tag{18}$$

□

Corollary 1. If G is a bag of sets such that $|G| = rc_G + 1$ then

$$n_G = 0 \;\equiv\; |G| = 1. \tag{19}$$

$$n_G = 1 \;\Rightarrow\; \langle \forall T :: \neg(pure_G.T) \rangle\,. \tag{20}$$

□

In general, the implication in (20) cannot be strengthened to equivales. For example, the bag G equal to $\{\{1,3\}\,,\{1,2,4\}\,,\{2,5\}\}$ satisfies the property that $|G| = rc_G + 1$ and every trip in G is non-pure. However, the set of nomads in G is $\{1,2\}$. That is, $n_G \neq 1$. The converse implication does hold for the bag of forward trips corresponding to an optimal putative sequence.

Theorem 1. Suppose F is a bag of sets satisfying (6), (7) and (8). Then there is a regular putative sequence of which the bag of forward trips equals F.[4]

Proof. Consider the algorithm below. It constructs a sequence S of forward and return trips. On termination, the bag of forward trips defined by S (denoted by $ForwardBag.S$ in the algorithm) equals F.

The symbol ε denotes the empty sequence and $S +\!\!+ S'$ denotes a sequence obtained by appending a sequence S' to the end of S. The sequence S' begins with the trip T and has total length $2 \times nc_G.T$. The trip T is followed in S' by a sequence of alternating return and forward trips, beginning and ending with a return trip. The return trips are made by the $nc_G.T$ nomads in T, the order being arbitrary; the forward trips are all pure, their choice is also arbitrary. The choice of T at each iteration (indicated by the "$[\!]T$" quantification[5]) is a non-pure trip with the property that $nc_G.T \leq pc_G + 1$. Note that whether or not a trip is pure is evaluated with respect to the bag G and not the bag F. The guard on the choice of T guarantees that S' can be constructed from the elements of G. The removal of one occurrence of T and the pure trips in S' from the bag G results in the bag denoted by $G \ominus S'$. The symbol "$\dot{\cup}$" in the invariant denotes bag union.

[4] Thanks to Arjan Mooij for providing the key insight in the proof of this theorem.

[5] Choice quantifiers are used frequently in this paper. Formally, $\langle [\!]k : R : S \rangle$ introduces a local variable k with scope delimited by the angle brackets; k is non-deterministically initialised to a value satisfying R, following which statement S is executed. The type of k is implicit. Here, T is a trip.

The invariant is truthified by the initialisation because F satisfies (6). It is also maintained by the loop body because $|G|$ is decreased by $nc_G.T$ and, simultaneously, $r_G.i$ is decreased by 1 for $nc_G.T$ instances of i; also, the forward trips added to the sequence S are precisely the trips removed from the bag G.

On termination of the loop, we claim that G has size 1; the sequence S is concluded by the one trip remaining in G.

```
$S,G \;:=\; \varepsilon,F$ ;
{ **Invariant**:   $|G| \;=\; rc_G + 1 \;\;\wedge\;\; F = G \,\dot{\cup}\, ForwardBag.S$ }
do ⟨[]$T$
      :    $T \in G \,\wedge\, \neg(pure_G.T) \,\wedge\, nc_G.T \leq pc_G + 1$
      :    { See text above for the definition of $S'$ }
           $S,G \;:=\; S \mathbin{+\!\!+} S' \,,\, G \ominus S'$
   ⟩
od
{ $|G| = 1$ } ;
$\langle []T : G = \{T\} : S \;:=\; S \mathbin{+\!\!+} [T] \rangle$
{ $F = ForwardBag.S$ }
```

The key to the correctness of this algorithm is the claim that the assertion "$|G| = 1$" is implied by the condition for terminating the loop:

$$\langle \forall T \;:\; T \in G \wedge \neg(pure_G.T) \;:\; \neg(nc_G.T \leq pc_G + 1) \rangle \quad .$$

The contrapositive of this claim is that, when $|G| \neq 1$, there is a non-pure trip available to extend the sequence S. We prove this as follows. Assume that $|G| \neq 1$. Then, by (19), $\langle \exists T : T \in G : \neg(pure_G.T) \rangle$. So,

$$\langle \exists T \;:\; T \in G \wedge \neg(pure_G.T) \;:\; nc_G.T \leq pc_G + 1 \rangle$$

$=$ { property of minimum }

$$\langle \Downarrow T : T \in G \wedge \neg(pure_G.T) : nc_G.T \rangle \;\leq pc_G + 1$$

$\Leftarrow$ { pigeon-hole principle (the minimum of a non-empty bag of integers is at most the average), (13) and integer inequalities }

$$\langle \Sigma T : \neg(pure_G.T) : nc_G.T \times \#_G T \rangle \;<\; np_G \times (pc_G + 2)$$

$=$ { (17) }

$$rc_G + n_G \;<\; np_G \times (pc_G + 2)$$

$\Leftarrow$ { (16) }

$$2 \times rc_G \;<\; np_G \times (pc_G + 2)$$

$$
\begin{array}{ll}
= & \{ \quad \text{by range splitting, } |G| = pc_G + np_G \; ; \\
 & \quad\; \text{also, by invariant, } |G| = rc_G + 1 \quad \} \\
 & 2 \times (pc_G + np_G - 1) \; < \; np_G \times (pc_G + 2) \\
\Leftarrow & \{ \quad \text{arithmetic} \quad \} \\
 & 2 \; \leq \; np_G \\
= & \{ \quad (18) \quad \} \\
 & 0 \neq np_G \\
= & \{ \quad (19) \text{ and assumption: } |G| \neq 1 \quad \} \\
 & \mathsf{true}.
\end{array}
$$

□

4 The Optimisation Problem

The optimisation problem we now focus on is to determine a bag of sets F such that properties (6), (7) and (8) hold of F which minimises the total travel time as given by (5). A bag, F, with the properties (6), (7) and (8) will be called a *regular* bag.

In our analysis, we refer to the subterm $\langle \Sigma T : T{\in}F : \langle \Uparrow i : i{\in}T : t.i \rangle \rangle$ in (5) as F's *forward time*, and $\langle \Sigma i :: t.i \times r_F.i \rangle$ as F's *return time*. We also refer to the subterm $\langle \Uparrow i : i{\in}T : t.i \rangle$ as T's *trip time* and $t.i \times r_F.i$ as person i's *return time*. It is important to note that we also use this terminology for bags of trips, F, that are not necessarily regular.

We continue to use the notion of "subsumption" but now applied to (regular) bags rather than sequences. So regular bag F subsumes regular bag G if F's total travel time is at most that of G. A bag is optimal if it is regular and subsumes all other bags, and is suboptimal if it is regular but not optimal.

Our solution is based on the following theorem.

Theorem 2. A bag of trips, F, that does *not* satisfy the following properties is *suboptimal.*

(a) For each T in F, the nomads in T are persons 1 thru $nc_F.T$. That is,

$$\langle \forall i{,}T \; : \; T{\in}F \wedge i{\in}T \; : \; nomad_F.i \equiv 1 \leq i \leq nc_F.T \rangle \, .$$

(b) The function $boss_F$ is monotonically increasing. That is, for all settlers i and j,

$$boss_F.i \leq boss_F.j \Leftarrow i \leq j.$$

(c) All pure trips in F are full. There is at most one non-full mixed trip in F and, if there is one, it is the fastest mixed trip and it has n_F nomads.

(d) For all non-nomadic trips, the function nc is a decreasing function of the leader of the trip. That is, for all non-nomadic trips T and U in F,

$$nc_F.T \geq nc_F.U \Leftarrow lead.T \leq lead.U.$$

□

In words, 2(a) expresses the property that, in an optimal bag, the nomads are the fastest, and always make forward trips in a contiguous group which includes person 1. 2(b) expresses the property that the trips divide the settlers into contiguous groups. 2(d) has the corollary that the pure settlers are the slowest. So, in summary, theorem 2 establishes the "intuitively obvious" property that the search for an optimal solution can be restricted to bags of trips in which, in order of increasing travel times, the groups of people are: the nomads, the settlers in a non-full mixed trip, the mixed settlers in full trips and the pure settlers.

To prove theorem 2 we use *proof-by-contradiction*. We prove a property P by contradiction by showing that every regular bag, F, that does not satisfy P can be transformed to a regular bag, F', that does satisfy P and has a strictly smaller total travel time. To establish a succession of properties, P and Q say, we first prove P and then assume P when proving Q.

Note that non-nomadic trips have multiplicity 1 in F. Thus, for non-nomadic trips T, there is no confusion between the trip T and the individual occurrences of T in F. On the other hand, nomadic trips may have multiplicity greater than 1 in F. For such trips, we are careful to make clear whether the transformation is applied to all occurrences of the trip or just one.

4.1 Choosing Nomads

We begin by proving part (a) of theorem 2. We first establish that the nomads are persons 1 thru n, for some n.

Lemma 3. Every regular bag of forward trips is subsumed by a regular bag in which all settlers are slower than all nomads.

Proof. Suppose that, within regular bag F, p is the fastest settler and q is the slowest nomad. Suppose p is faster than q.

Interchange p and q everywhere in F. We get a regular bag, F'. The return time is clearly reduced by at least $t.q - t.p$.

The times for the forward trips in F involving q are not increased in F' (because $t.p < t.q$). The time for the *one* forward trip in F involving p is increased in F' by an amount that is at most $t.q - t.p$. This is verified by considering two cases. The first case is when q is an element of p's forward trip. In this case, swapping p and q has no effect on the trip, and the increase in time taken is 0. In the second case, q is not an element of p's forward trip. In this case, it suffices to observe that, for any x (representing the maximum time taken by the other participants in p's forward trip),

$$
\begin{array}{ll}
 & t.p \uparrow x \ + \ (t.q - t.p) \\
= & \qquad \{ \qquad \text{distributivity of sum over max, arithmetic} \quad \} \\
 & t.q \uparrow (x + (t.q - t.p)) \\
\geq & \qquad \{ \qquad t.p \leq t.q, \text{ monotonicity of max} \quad \} \\
 & t.q \uparrow x .
\end{array}
$$

Finally, the times for all other forward trips are unchanged.

The net effect is that the total time taken does not increase. That is, F' subsumes F. Also, the total forward-trip time of the settlers is strictly increased. Thus, repeating the process of swapping the fastest settler with the slowest nomad whilst the former is faster than the latter is guaranteed to terminate with a bag that subsumes the given bag and in which all settlers are slower than all nomads. □

Lemma 4. Every regular bag of forward trips is subsumed by a bag, F, that satisfies 2(a).

Proof. Suppose a regular bag F of forward trips is given. By lemma 3, F is subsumed by a bag G in which the nomads are persons 1 thru n_G. (Bags F and G may be the same, but that is not significant.)

For each trip T in G, consider the set of nomads in T. Specifically, define $nom.T$ to be

$$T \cap \{i \,|\, nomad_G.i\}.$$

Recall that $nc_G.T$ is the number of nomads in set T. That is, $nc_G.T = |nom.T|$.

Replace T in the bag G by

$$(T \cap \{i \,|\, settler_G.i\}) \;\cup\; \{i \mid 1 \leq i \leq nc_G.T\}.$$

This replaces G by a bag F'. To see that F' is regular, we observe that the replacement of T increases the number of forward trips by person i only when $i \leq nc_G.T$. But $nc_G.T \leq n_G$; so, settlers in G are also settlers in F'. Hence, the size of the bag T is unchanged by the replacement. That is, property (7) is an invariant of the replacement. The number of forward trips made by nomads i in G such that $nc_G.T < i \leq n_G$ decreases by at most 1. So, such nomads may not be nomads in F'. However, the number of forward trips each makes remains strictly positive (since a nomad makes at least 2 forward trips, by definition), and each decrease in the number of forward trips made by such a nomad is compensated by an increase in the number of forward trips made by some nomad i, where $1 \leq i \leq nc_G.T$. That is, properties (8) and (6) are invariant under the replacement. Finally, the replacement decreases the total trip time because the times of the return trips are decreased, and the times of the forward trips are not increased. That is, F' subsumes G; by the transitivity of the subsumes relation, F' also subsumes F. □

Corollary 2. Every regular bag is subsumed by a bag, F, in which the number of nomads, n_F, is at most C.

Proof. Every regular bag is subsumed by a bag, F, satisfying 2(a), and

$$\begin{array}{ll} & n_F \leq C \\ = & \qquad \{ \quad 2(a) \quad \} \\ & \langle \Uparrow T : T \in F : nc_F.T \rangle \;\leq C \end{array}$$

$=$ { definition of maximum ($\Uparrow$) }

$\langle \forall T : T \in F : nc_F.T \leq C \rangle$

$=$ { definition of nc_F, (7) }

true. □

4.2 Permuting Settlers

In this section, we prove part (b) of theorem 2. We begin, however, with a similar lemma which is used later in the proof of part (d).

Lemma 5. Suppose bag F satisfies 2(a). Then either the settler count in each trip is a monotonic function of the leader of the trip (i.e.

$$sc_F.T \leq sc_F.U \Leftarrow lead.T \leq lead.U \quad)$$

or F is suboptimal.

Proof. Take trips T and U in F such that $sc_F.T > sc_F.U$ and $lead.T \leq lead.U$. It follows that $T \neq U$. Because $sc_F.T > 0$ and F satisfies 2(a), $lead.T$ is a settler. Now, because $lead.T \leq lead.U$ and F satisfies 2(a), it follows that $lead.U$ is also a settler. But settlers are elements of exactly one trip. We conclude that $sc_F.T > sc_F.U > 0$, both T and U have multiplicity 1 in F, and $lead.T < lead.U$.

Rearrange the settlers in T and U so that $sc_F.T$ and $sc_F.U$ are unchanged (thus guaranteeing a regular bag) and the slowest settlers are in T and the fastest settlers are in U. Using primes to denote the new values of T and U, we have

$t.(lead.T') + t.(lead.U')$

$=$ { $lead.U$ is the slowest settler, so $lead.T' = lead.U$ }

$t.(lead.U) + t.(lead.U')$

$<$ { $sc_F.U' = sc_F.U < sc_F.T$ and U' contains the fastest settlers; so $lead.U' < lead.T$ }

$t.(lead.U) + t.(lead.T)$.

Other trips are unchanged, so the effect is to strictly decrease the total travel time. □

Lemma 6. A bag F that does not satisfy 2(b) is suboptimal.

Proof. Take any two settlers i and j such that $boss_F.i > boss_F.j$ and $i \leq j$. It follows that $i \neq j$ and they must be in different trips, T and U say. Swap $boss_F.j$ (the slowest person in trip U) with the fastest settler in trip T. Then, using primes to denote the new trips,

$t.(lead.T') + t.(lead.U')$

$=$ { $i < j \leq boss_F.j < boss_F.i$; so $i \neq boss_F.i$

hence $lead.T' = lead.T = boss_F.i$ $\}$

$t.(lead.T) + t.(lead.U')$

$<$ $\{$ $lead.U = boss_F.j$,

$boss_F.j$ has been replaced by k where $k \le i < j \le boss_F.j$ $\}$

$t.(lead.T) + t.(lead.U)$.

Other trips are unchanged, so the effect is to strictly decrease the total travel time. □

4.3 Filling Non-nomadic Trips

Part (c) of theorem 2 is about filling non-nomadic trips as far as possible with nomads. The proof is split into several lemmas. The proofs themselves are omitted because they add no new techniques.

Lemma 7. A bag F that satisfies 2(a) but has a non-full pure trip is suboptimal. □

Lemma 8. A bag F that satisfies 2(a) but has a non-full mixed trip that is not the fastest mixed trip is suboptimal. □

Lemma 9. Suppose bag F satisfies 2(a). Suppose F has a non-full mixed trip that is the fastest mixed trip but all mixed trips in F have fewer than n_F nomads. Then F is suboptimal. □

Lemma 10. Suppose bag F satisfies 2(a). Suppose there is a mixed trip, T say, with n_F nomads and a non-full mixed trip, U say, with n nomads where $n < n_F$. Suppose U is the fastest mixed trip. Then F is suboptimal. □

Corollary 3. A bag F that satisfies 2(a) but does not satisfy theorem 2(c) is suboptimal. A bag F that satisfies 2(a) but does not satisfy theorem 2(d) is suboptimal. □

5 Constructing an Optimal Bag of Forward Trips

In this section, we use theorem 2 to give a lower bound on the time taken to cross. In the process of calculating the lower bound, an optimal bag of forward trips can be constructed. Then, using the construction given in section 4, an optimal putative sequence can be constructed from the bag.

Our algorithm for constructing an optimal bag constructs in stages an "ordered" bag of sets where "ordered" is defined below.

Definition 1 (Ordered). We say that a bag of sets, F, is *ordered* if

$$\langle \forall T : T \in F : 2 \le |T| \le C \rangle \tag{21}$$

and it satisfies the four properties stated in theorem 2. □

The algorithm constructs a bag of trips, F, starting with the slowest trips. The measure of progress is a pair (m,p) ordered lexicographically, where m is a measure of the number of people not yet included in a trip and p measures the "excess" of pure trips over return trips. Formally, we exploit the following theorem.

Theorem 3. A bag of sets, F, is optimal if it is ordered and

$$\langle \forall i \ : \ 1 \leq i \leq N \ : \ 1 \leq f_F.i \rangle \, , \tag{22}$$

and

$$pc_F \ = \ \langle \Sigma i : 2 \leq i \leq n_F : r_F.i \rangle \, . \tag{23}$$

Proof. Comparing the definition of regular bags (properties (6), (7) and (8)) with the definition of ordered bags, we see that (21) and (7) are identical, as are (22) and (8). Thus, any ordered bag is a solution if it also satisfies (6). We now show that (6) and (23) are equivalent when the bag F satisfies the properties stated in theorem 2. That is, we prove that

$$|F| \ = \ \langle \Sigma i :: r_F.i \rangle + 1 \quad \equiv \quad pc_F \ = \ \langle \Sigma i : 2 \leq i \leq n_F : r_F.i \rangle \, .$$

We have:

$$\begin{array}{ll} & |F| \\ = & \{ \quad \text{definition of } |F|\text{, range splitting} \quad \} \\ & pc_F \ + \ np_F \\ = & \{ \quad \text{by 2(a), } 1 {\in} T \equiv \neg(pure_F.T)\text{, definition of } f_F.1 \quad \} \\ & pc_F \ + \ f_F.1 \ \ . \end{array}$$

Hence,

$$\begin{array}{ll} & pc_F \ = \ \langle \Sigma i : 2 \leq i \leq n_F : r_F.i \rangle \\ = & \{ \quad \text{above, arithmetic} \quad \} \\ & |F| - f_F.1 \ = \ \langle \Sigma i : 2 \leq i \leq n_F : r_F.i \rangle \\ = & \{ \quad f_F.1 - 1 = r_F.1\text{, arithmetic} \quad \} \\ & |F| \ = \ \langle \Sigma i : 1 \leq i \leq n_F : r_F.i \rangle + 1 \\ = & \{ \quad \text{by 2(a), } \quad r_F.i \neq 0 \ \equiv \ 1 \leq i \leq n_F \quad \} \\ & |F| \ = \ \langle \Sigma i :: r_F.i \rangle + 1. \end{array}$$

□

Henceforth, we call $pc_F - \langle \Sigma i : 2 \leq i \leq n_F : r_F.i \rangle$ the *excess* of the bag F. Theorem 3 states that an optimal bag is obtained by constructing an ordered bag in which everyone makes a trip (property (22)) and the number of pure trips equals the number of return trips made by nomads other than person 1 (property (23)).

Theorem 3 offers no way of determining the number of pure trips in an optimal bag except by considering all the different possibilities. The basic structure of our solution is thus to evaluate the minimum over all p of the total travel time of a regular, ordered bag of forward trips in which the number of pure trips is p. So, the problem becomes one of determining a lower bound on the travel time incurred by a bag of mixed and nomadic forward trips with an "excess" p. This problem is solved by identifying a collection of (acyclic) equations on the travel times; by determining the (unique) solution of the equations we obtain the desired lower bound on the total travel time; simultaneously, the bag of forward trips can be constructed in the standard way.

For convenience, we define rt by

$$rt.n \quad = \quad \langle \Sigma i : 1 \leq i \leq n : t.i \rangle \, . \tag{24}$$

In the equations below, occurrences of the function rt record the return time for a given number of nomads; occurrences of the function t record the forward time of some trip.

5.1 Outline Algorithm

The basic structure of our solution is to design a non-deterministic algorithm that constructs regular, ordered bags. The algorithm is designed so that every such bag is constructed by some resolution of the non-determinism.

An outline of the algorithm is shown below.

```
$\{\ 2 \leq C < N\ \}$
$\langle [\!]\, p$
  :    $0 \leq p \leq \left\lfloor \frac{N-2}{C} \right\rfloor$
  :    $AddPureTrips$
       $\{\ |F| = p = pc_F\ \}$ ;
       $\langle [\!]\, n,n'$
         :    $n = n' = 0$
         :    $\langle [\!]\, m$
                :    $m = N - p \times C$
                :    $AddFullMixedTrips$
                     $\{\ 2 \leq m \leq C\ \}$ ;
                     $IncludeRest$
                     $\{\ \langle \forall i\ :\ 1 \leq i \leq N\ :\ 1 \leq f_F.i \rangle\ \}$ ;
                     if  $0 = p \ \rightarrow$  skip
                   □ $0 < p \ \rightarrow\ FinaliseNumberOfNomads$
```

```
                              fi
                          ⟩ ;
                          $AddNomadicTrips$
                  ⟩
          ⟩
          { $(ordered.F \wedge \langle \forall i : 1 \leq i \leq N : 1 \leq f_F.i \rangle)$
            $\wedge \; pc_F = \langle \Sigma i : 2 \leq i \leq n_F : r_F.i \rangle$ }
```

The algorithm constructs a bag, F, of trips. At all stages, F is ordered. The algorithm begins by introducing a variable p which is non-deterministically initialised to a natural number at most $\left\lfloor \frac{N-2}{C} \right\rfloor$. The step $AddPureTrips$ initialises the variable F to a bag of p pure trips. Informally, p is the excess of the bag F. (This isn't quite true as explained below.) Subsequent stages reduce p to zero.

Next, variables n and n' are introduced, both with initial value zero; an invariant of n is that persons 1 thru n are nomads in F. The value of n' is always at least n. Persons $n{+}1$ thru n' have a forward count of one (and so are settlers); these persons will, however, eventually become nomads in F.

The variable m is introduced next. An invariant of m is that persons $n'{+}1$ thru m are the ones with a forward count of zero in F; since all pure trips are full, the initial value of m is thus $N - p{\times}C$. The step $AddFullMixedTrips$ adds full mixed trips to F while C is less than m. Then $IncludeRest$ adds one additional trip to F in order to guarantee that every person is included in at least one trip. This additional trip may be full or non-full.

The final step is to add nomadic trips to F in order to reduce the excess p to 0, if this is not already the case. The number of nomads in these trips is at least $n'{\uparrow}2$ and at most $m{\downarrow}(p{+}1)$.

The total travel time is simply the minimum over all possible choices of the travel times of the constructed bags. The calculation of the optimal travel time is equivalent to a shortest-path problem. Formally, each non-deterministic choice is interpreted as a minimum and the addition of a set to F adds the return-trip time of the nomads in the added trip (where the predicate *nomad* is evaluated once the trip is added) and the forward-trip time to the total trip time. We exploit the fact that addition distributes over minimum —the formal equivalent of the "principle of optimality" of dynamic programming— in order to obtain a polynomial-time algorithm.

Let us now give the details of the individual steps.

5.2 Adding Pure Trips

The first step (after the non-deterministic initialisation of p) is $AddPureTrips$ which initialises F to a set of p pure trips. In order to guarantee that the nomad count nc_F is a decreasing function of the leader, $boss_F$ is increasing, and all pure trips are full, the assignment to F is simply

$$F \;:=\; \langle \cup k \;:\; 1 \leq k \leq p \;:\; \{\{i \mid N - k \times C < i \leq N - (k-1) \times C\}\}\rangle\,.$$

All trips added to F are full, and the leader of the kth trip is person $N-(k-1)\times C$.

On completion of this assignment, F is ordered and $|F| = p = pc_F$.

The forward-trip time for the pure trips is constant. It is given by the function PT:

$$PT.p \;=\; \langle \Sigma i : 0 \leq i < p : t.(N - C \times i)\rangle\,. \tag{25}$$

Thus

$$TOT \;=\; \left\langle \Downarrow p : 0 \leq p \leq \left\lfloor \frac{N-2}{C} \right\rfloor : PT.p \;+\; NP.p \right\rangle . \tag{26}$$

The value of the function NP is determined by the mixed and nomadic trips added to F in the later stages.

(Formally, (26) is a consequence of the fact that addition distributes over minimum. In other words, the time for the pure trips can be "factored out" of the calculation of the total travel time.)

5.3 Adding Full Mixed Trips

In the second stage, variables n, n' and m are initialised to 0, 0 and $N - p \times C$, respectively, and a set of full mixed trips is added to F as follows.

```
{ Invariant:      $ordered.F$
             ∧    $0 \leq n \leq n' \leq C-1 \;\wedge\; n'\uparrow 1 < m$
             ∧    $\langle \forall i : 1 \leq i \leq n : 1 \leq r_F.i \rangle$
             ∧    $\langle \forall i : n < i \leq n' \vee m < i \leq N : 1 = f_F.i \rangle$
             ∧    $pc_F \;=\; \langle \Sigma i : 2 \leq i \leq n : r_F.i\rangle + p + [0 = n < n']$
             ∧    $d.m.n.n' \leq p$ }
do $C < m$ →
        $\langle [\,]\, i$
          :  $n'\uparrow 1 \leq i \leq C-1 \;\wedge\; d.(m-C+i).n'.i \leq p - n'\uparrow 1 + 1$
          :  $F \;:=\; F \cup \{\{j \mid 1 \leq j \leq i\} \cup \{j \mid m-C+i < j \leq m\}\}$ ;
             $m, n, n', p \;:=\; m-C+i\,,\, n'\,,\, i\,,\, p - n'\uparrow 1 + 1$
        $\rangle$
od
```

The second, third and fourth clauses of the invariant of this loop express precisely the functions of m, n and n'. Refer back to section 5.1 for an informal account of their function.

The clause

$$pc_F \;=\; \langle \Sigma i : 2 \leq i \leq n : r_F.i\rangle + p + [0 = n < n']$$

in the invariant expresses precisely the relation between p and the pure count of F. Note that when $0 = n$ all trips in F are pure; when, in addition, $n < n'$ there is one trip in F which includes persons 1 thru n'. At a later stage, this trip will become a mixed trip and, so, is not counted in p. The term $[0 = n < n']$ evaluates to 1 if $0 = n < n'$ and to 0 otherwise. The inclusion of this term compensates for not counting the trip in the excess p.

The clause

$$d.m.n.n' \leq p$$

requires some explanation. Its function is to guarantee that the non-deterministic choices do not abort; equivalently, the value of p is always at least zero. Specifically,

$$d.m.n.n' \;=\; \left(\left\lceil \frac{m{-}C}{C - n'{\uparrow}1} \right\rceil + 1\right) \times (n'{\uparrow}1 - 1). \tag{27}$$

In order to guarantee 2(d) —the number of nomads in non-nomadic trips is a decreasing function of the leader of the trip— , the value of n' is increasing; each mixed trip that is added to F has at least $n'{\uparrow}1$ nomads and at most $C - n'{\uparrow}1$ settlers. Thus, in this stage, at least $\left\lceil \frac{m{-}C}{C - n'{\uparrow}1} \right\rceil$ additional trips are added to F, each of which causes p to be reduced by at least $n'{\uparrow}1 - 1$. The third stage is executed when $m \leq C$; this stage guarantees that all people make at least one trip by adding to F a single trip consisting of persons 1 thru m. This also causes p to be reduced by at least $n'{\uparrow}1 - 1$. Thus $d.m.n.n'$ is a lower bound on the amount that p will be reduced by the later addition of trips to F.

We leave the verification of the invariant to the reader.

From the algorithm, we can determine the minimum total travel time for the mixed trips. We have, for all p such that $0 \leq p \leq \left\lfloor \frac{N{-}2}{C} \right\rfloor$,

$$NP.p \;=\; MX.(N - p{\times}C).0.0.p.$$

For the moment, we define $MX.m.n.n'.p$ only for the case that $C < m$. Specifically, for all n and n' such that $0 \leq n \leq n' \leq C{-}1$, all m such that $C < m$, and all p such that $d.m.n.n' \leq p$, we have

$$\begin{aligned} MX.m.n.n'.p = \;& \langle{\Downarrow} i \\ & \quad : \; n'{\uparrow}1 \leq i \leq C{-}1 \;\wedge\; d.(m{-}C{+}i).n'.i \leq p - n'{\uparrow}1 + 1 \\ & \quad : \; MX.(m{-}C{+}i).n'.i.(p - n'{\uparrow}1 + 1) \\ & \rangle \;+\; t.m \;+\; rt.n'. \end{aligned}$$

The justification of this equation is that, for a given choice of i, a trip is added to F with leader m. The forward time for the trip is thus $t.m$. The trip adds 1 to the forward count of each person from 1 thru i; after the addition of the trip, persons 1 thru n' are the nomads in F and the addition of the trip to F adds $rt.n'$ to F's return-trip time. As before, the distributivity of addition over minimum is used to "factor out" the contribution of each trip to the total travel time.

5.4 Completing the Mixed Trips

The third stage, *IncludeRest*, ensures that everyone makes at least one forward trip. The assignment is simply

$$F\,,n\,,p \quad := \quad F\cup\{\{j \,|\, 1\le j\le m\}\}\;,\;n'\;,\;p-n'\uparrow 1+1$$

From the invariant of the second stage, we determine that the assignment is executed when $n'\uparrow 1<m\le C$; this guarantees that the added trip is regular. Also, from the invariant, $d.m.n.n' \le p$. That is, $n'\uparrow 1-1 \;\le\; p$. After the assignment, it is thus that case that $0\le p$. In more detail, the postcondition established by the assignment is

$$\begin{array}{ll} & ordered.F \\ \wedge & \langle \forall i : 1\le i\le N : 1\le f_F.i\rangle \\ \wedge & pc_F \;=\; \langle \Sigma i : 2\le i\le n : r_F.i\rangle + p + [0=n<n'] \\ \wedge & (0<p \,\vee\, 0<n). \end{array}$$

The final conjunct is a consequence of the assumption $C<N$. At the conclusion of the second stage, F is non-empty; so, $|F|$ is at least 2 after the above assignment. By inspection of the assignments in the two stages, we conclude that either $0<p$ or $0<n'$. The conjunct follows because n is assigned the value of n'.

The trip that is added to F may be full (when $m=C$) or non-full (when $m<C$); it may also be (or become) a mixed trip or it may be a nomadic trip. Immediately following the assignment, the statement

```
if  $0=p$ → skip
□  $0<p$ →   $\langle [\,]\,i$
                  :   $n'\uparrow 2 \le i \le m\downarrow(p{+}1)$
                  :   $n' \;:=\; i$
              $\rangle$
fi
```

is executed. This chooses the final value of the number of nomads. (The choice of i cannot abort because, as remarked above, $n'\uparrow 1<m\le C$; as a consequence, if $0<p$ then $n'\uparrow 2 \le m\downarrow(p{+}1)$.)

The addition of this trip extends our definition of the function MX. Anticipating the fact that when $0=p$ no further trip is added to F, we have: for all m, n and n' such that $m\le C$,

$$MX.m.n.n'.(n'\uparrow 1-1) \;=\; t.m+rt.n'.$$

Also, for all m, n and n' such that $2 \leq m \leq C$, and all p such that $0 < p$,

$$\begin{aligned} & MX.m.n.n'.(p + n'{\uparrow}1 - 1) \\ = \; & \langle \Downarrow i : n'{\uparrow}2 \leq i \leq m{\downarrow}(p{+}1) : NT.n'.i.p \rangle + t.m + rt.n' . \end{aligned}$$

The function NT gives the time taken by the nomadic trips. See below.

5.5 Adding Nomadic Trips

The final stage in the construction of F is the possible addition of a number of nomadic trips. When this stage is executed, we have the following properties of F, n, n' and p:

$$\begin{aligned} & 0 \leq n \leq n' \leq C \;\wedge\; (0 = p \;\vee\; 2 \leq n' \leq p{+}1) \\ \wedge \;\; & ordered.F \\ \wedge \;\; & pc_F \;=\; \langle \Sigma i : 2 \leq i \leq n : r_F.i \rangle + p + [0 = n < n'] \\ \wedge \;\; & \langle \forall i : 1 \leq i \leq n : 1 \leq r_F.i \rangle \;\wedge\; \langle \forall i : n < i \leq N : 1 = f_F.i \rangle \end{aligned}$$

The goal is to add nomadic trips to F in such a way that F remains ordered, the number of nomads in F becomes n' (if it is not n' already) and p is reduced to zero.

We present a non-deterministic algorithm to achieve this task. The algorithm is designed so that every bag that satisfies the specification corresponds to one way of resolving the non-determinism.

The algorithm begins by ensuring that F has n' nomads, and then repeatedly adds nomadic trips to F whilst $0{<}p$. By choosing the trips in order of decreasing size, 2(a) is guaranteed without loss of choice. The symbl "$\dot{\cup}$" denotes bag union. (It is only at this point in the construction that trips in F may have multiplicity greater than 1.)

```
{ Precondition given above }
if $0{=}p \rightarrow$ skip
□ $0{<}p \wedge n{<}n' \;\rightarrow\; F, p, n \;:=\; F \cup \{\{j \mid 1 \leq j \leq n'\}\}, p{-}n'{+}1, n'$
fi
{ $\langle \forall i : 1 \leq i \leq n' : 1 \leq r_F.i \rangle \;\wedge\; \langle \forall i : n' < i \leq N : 1 = f_F.i \rangle$ } ;
do $0{<}p \rightarrow$  $\langle [\,]\, i$
                    :    $2 \leq i \leq (p{+}1){\downarrow}n$
                    :    $F, p, n \;:=\; F \;\dot{\cup}\; \{\{j \mid 1 \leq j \leq i\}\}, p{-}i{+}1, i$
                 $\rangle$
od
{ $ordered.F \;\wedge\; n_F = n' \;\wedge\; pc_F \;=\; \langle \Sigma i : 2 \leq i \leq n_F : r_F.i \rangle$ }
```

Now we determine the additional travel time incurred by adding these trips to F. Letting $NT.n.n'.p$ denote the additional time, we have:

$$NT.n.n'.0 \;=\; 0.$$

Also, for all p, n and n' such that $0{<}p$, $0 \leq n < n'$ and $2 \leq n' \leq (p{+}1){\downarrow}C$,

$$NT.n.n'.p \;=\; t.n' + rt.n \;+\; TNT.n'.(p - n{\uparrow}1 + 1).$$

Finally, for all p and n such that $0{<}p$, and $0 \leq n$,

$$NT.n.n.p \;=\; TNT.n.p.$$

The function TNT is given by:

$$TNT.n.0 \;=\; 0,$$

and, for all p such that $0{<}p$ and all n such that $2 \leq n \leq (p{+}2){\downarrow}C$,

$$TNT.n.p \;=\; \langle \Downarrow i : 2 \leq i \leq (p{+}1){\downarrow}n : t.i \;+\; rt.i \;+\; TNT.i.(p{-}i{+}1) \rangle \,.$$

5.6 Solving the Equations

Finding the total travel time incurred by an optimal bag of forward trips is achieved by determining the greatest solution to the above equations. This can be done in worst-case time $O(N^2 {\times} C^3)$ by first tabulating PT and TNT, then NT followed by MX, and finally TOT. Each set of equations is evaluated by imposing an appropriate lexicographic ordering on the parameters; for example, $MX.m.n.n'.p$ is evaluated in lexicographic order on p, m, $C{-}n'$ and $C{-}n$. (As remarked earlier, the process is equivalent to solving a shortest-path problem, so other shortest-path algorithms could be used. These are likely to be more effective in practice; but they may have poorer worst-case complexities.) An optimal bag (rather than just the travel time) can be determined in the standard way by recording the terms that realise the minimum when evaluating any such quantification.

6 Conclusion

That the torch problem can be solved at the level of generality in this paper in time quadratic in the number of people appears to be new. The case of 4 people and a bridge capacity of 2 is very widely discussed on the internet (although its origin appears to be unknown). The case of N people and a bridge capacity of 2 has been solved by Rote [3]. (The reader is referred to Rote's paper for other publications and web links.) I believe that the capacity-C problem has been attempted, but presumably never solved (in polynomial time), because I was told before embarking on it that it was believed to be "hard" [Tom Verhoeff, private communication]. The problem is indeed difficult, but it is not "hard" in any technical sense of the word (as, for example, in "NP-hard"). As we have shown, the problem can be solved in time quadratic in the number of people.

I was motivated to tackle the problem because I use the capacity-2 problem in a course on algorithmic problem solving [1] to entry-level Computer Science students at the University of Nottingham. The capacity-2 problem is interesting because it demonstrates that "obvious" solutions may be incorrect; also, obtaining a correct solution demands particular attention to the avoidance of unnecessary detail. (Like the capacity-C problem, the solution is obtained by focusing on just the forward trips.) Initially, I thought that the capacity-C problem would not be much more difficult than the capacity-2 problem. It turned out to be far harder to solve than I anticipated.

Of course, the solution presented here specialises in the case that the capacity is 2. In this case, the solution is essentially the same as that presented by Rote. (In the case that the capacity is 2, all the complications concerning the possibility of non-full bags disappear; for this reason, the construction simplifies considerably.) Rote describes the solution in terms of "multigraphs" rather than bags. For capacity 2, the difference is superficial. Each edge of a "multigraph" connects two people and, hence, is just a set of two people. However, Rote's "multigraphs" do not appear to generalise to capacity N, whereas the use of bags does.

For capacity 2, the solution can be determined in logarithmic time. For full details see [1, chapter 8]. Rote describes the more obvious "greedy", linear-time algorithm.

The current solution leaves much to be desired. The underlying calculations have not been done to the level of rigour and detail that I would nowadays demand. The fact that the capacity-2 problem can be solved by a greedy algorithm suggests that there is much scope for improving the worst-case complexity of the solution presented here. The reason I suspect that improvements can be made is that the "edges" in the underlying path problem connect vertices labelled with a four-tuple one of whose components is p but the "length" of an edge does not depend on p. This suggests that, at the very least, a linear-time algorithm should be possible.

Acknowledgements. Thanks go to a number of people who have helped me at various stages.

I thank Diethard Michaelis for stressing the importance of regularity and for his suggestions on naming (which I have adopted). Thanks also for his reading and commenting on drafts and for his efforts to improve on notation (which I have yet to do anything about!).

Thanks to Arjan Mooij for helping me prove that any regular bag of forward trips can be transformed to a putative sequence. Section 3.1 is essentially due to him — but the responsibility for errors is mine. Thanks also to João Ferreira and Arjan Mooij for suggesting improvements to some calculations.

Finally, thanks to Tom Verhoeff and Günter Rote for providing me with bibliographic information on the problem.

References

1. Backhouse, R.: Algorithmic problem solving. Lecture notes, School of Computer Science, University of Nottingham. Updated at least annually and widely available on the internet, but see author's website for latest version
2. Backhouse, R.: Regular algebra applied to language problems. Journal of Logic and Algebraic Programming (66), 71–111 (2006)
3. Rote, G.: Crossing the bridge at night. Bulletin of the European Association for Theoretical Computer Science 78, 241–246 (2002)

Recounting the Rationals: Twice!

Roland Backhouse and João F. Ferreira*

School of Computer Science
University of Nottingham
Nottingham NG8 1BB, England
{rcb,jff}@cs.nott.ac.uk

Abstract. We derive an algorithm that enables the rationals to be efficiently enumerated in two different ways. One way is known and is credited to Moshe Newman; it corresponds to a deforestation of the so-called Calkin-Wilf tree of rationals. The second is new and corresponds to a deforestation of the Stern-Brocot tree of rationals. We show that both enumerations stem from the same simple algorithm. In this way, we construct a Stern-Brocot enumeration algorithm with the same time and space complexity as Newman's algorithm.

Keywords: Calkin-Wilf tree, Stern-Brocot tree, algorithm derivation, enumeration algorithm, rational numbers.

Recently, there has been a spate of interest in the construction of bijections between the natural numbers and the (positive) rationals (see [4,6,2] and [1, pages 94–97]). Gibbons *et al* [4] describe as "startling" the observation that the rationals can be efficiently enumerated[1] by "deforesting" the Calkin-Wilf [2] tree of rationals. However, they claim that it is "not at all obvious" how to "deforest" the Stern-Brocot tree of rationals. (For information on the Stern-Brocot tree, see [5, pages 116–118].)

In this paper, we derive an efficient algorithm for enumerating the rationals both in Calkin-Wilf and Stern-Brocot order. The algorithm is based on a bijection between the rationals and invertible 2×2 matrices. The key to the algorithm's derivation is the reformulation of Euclid's algorithm in terms of matrices. The enumeration is efficient in the sense that it has the same time and space complexity as the algorithm credited to Moshe Newman in [6], albeit with a constant-fold increase in the number of variables and number of arithmetic operations needed at each iteration.

Section 1 reviews Euclid's algorithm, whilst section 2 discusses the enumeration algorithms. Section 3 discusses the method used to derive the algorithm. The appendix documents a Haskell implementation of the algorithm.

* Funded by Fundação para a Ciência e a Tecnologia (Portugal) under grant SFRH/BD/24269/2005.

[1] By an *efficient enumeration* we mean a method of generating each rational without duplication with constant cost per rational in terms of arbitrary-precision simple arithmetic operations.

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 79–91, 2008.

1 Euclid's Algorithm

A positive rational in so-called "lowest form" is an ordered pair of positive, co-prime integers. Every rational $\frac{m}{n}$ has unique lowest-form representation $\frac{m/(m\triangledown n)}{n/(m\triangledown n)}$. (We use "$\triangledown$" to denote "greatest common divisor". We prefer to use an infix notation whenever —as in this case— the operator is symmetric and associative. As we see below, the exploitation of symmetry and associativity is extremely important to effective reasoning.)

Because computing the lowest-form representation involves computing greatest common divisors, it seems sensible to investigate Euclid's algorithm to see whether it gives insight into how to enumerate the rationals. Indeed it does.

Below we present Euclid's algorithm as it might be presented in a modern textbook. (We use Dijkstra's Guarded Command Language [3] to express the algorithm because it allows us to fully express the symmetry between m and n. The "do-od" statement is executed repeatedly. Termination occurs when both of the two guards $y<x$ and $x<y$ are false (i.e. when x and y are equal). When $y<x$ evaluates to true, the assignment $x := x-y$ is executed, and then the do-od is executed again. Similarly, when $x<y$ the assignment $y := y-x$ is executed before repeated execution of the do-od statement.)

```
{ $0<m \wedge 0<n$ }
$x,y := m,n$;
{ **Invariant**: $0<x \wedge 0<y \wedge x\triangledown y = m\triangledown n$
  **Bound function**: $x+y$ }
do $y<x \rightarrow x := x-y$
□ $x<y \rightarrow y := y-x$
od
{ $x = y = x\triangledown y = m\triangledown n$ }
```

The algorithm below is a somewhat unusual, but very effective, way of rewriting Euclid's algorithm when the goal is to establish the theorem that the greatest common divisor of two numbers is a linear combination of the numbers.

The algorithm is expressed in matrix terms. The input to the algorithm is a vector $(m\,n)$ of strictly positive integers. The vector $(x\,y)$ is initialised to $(m\,n)$ and, on termination, its value is the vector $(m\triangledown n\; m\triangledown n)$. In addition to computing the greatest common divisor, it also computes a matrix $\mathbf{C}$. An invariant of the algorithm is that the vector $(x\,y)$ equals $(m\,n)\times\mathbf{C}$. In words, $(x\,y)$ is a "linear combination" of $(m\,n)$. Specifically, $\mathbf{I}$, $\mathbf{A}$ and $\mathbf{B}$ are 2×2 matrices; $\mathbf{I}$ is the identity matrix $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $\mathbf{A}$ is the matrix $\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$ and $\mathbf{B}$ is the matrix $\begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$. The assignment $(x\,y) := (x\,y)\times\mathbf{A}$ is equivalent to $x,y := x-y,y$, as can be easily checked.

```
$\{\ 0<m \wedge 0<n\ \}$
$(x\,y),\mathbf{C} := (m\,n),\mathbf{I}\,;$
$\{\ \textbf{Invariant}:\quad (x\,y) = (m\,n)\times\mathbf{C}\ \}$
do $y<x \rightarrow (x\,y),\mathbf{C} := (x\,y)\times\mathbf{A}\,,\mathbf{C}\times\mathbf{A}$
$\Box\ x<y \rightarrow (x\,y),\mathbf{C} := (x\,y)\times\mathbf{B}\,,\mathbf{C}\times\mathbf{B}$
od
$\{\ (x\,y) = (m\triangledown n\ m\triangledown n) = (m\,n)\times\mathbf{C}\ \}$
```

The verification of the supplied invariant is a simple consequence of the associativity of matrix multiplication. It is this form of the algorithm that is the starting point for our enumeration of the rationals.

2 Enumerating the Rationals

Beginning with an arbitrary pair of positive integers m and n, the above algorithm calculates an invertible matrix $\mathbf{C}$ such that

$$(m\triangledown n\ m\triangledown n) = (m\,n)\times\mathbf{C}.$$

It follows that

$$(1\,1)\times\mathbf{C}^{-1} = \left({}^{m}/_{(m\triangledown n)}\ {}^{n}/_{(m\triangledown n)}\right). \tag{1}$$

Because the algorithm is deterministic, positive integers m and n uniquely define the matrix $\mathbf{C}$. That is, there is a function from pairs of positive integers to finite products of the matrices $\mathbf{A}$ and $\mathbf{B}$.

Also, because the matrices $\mathbf{A}$ and $\mathbf{B}$ are constant and invertible, $\mathbf{C}^{-1}$ is a finite product of the matrices $\mathbf{A}^{-1}$ and $\mathbf{B}^{-1}$ and (1) uniquely defines a rational $\frac{m}{n}$. We may therefore conclude that there is a bijection between the rationals and the finite products of the matrices $\mathbf{A}^{-1}$ and $\mathbf{B}^{-1}$ provided that we can show that all such products are different.

The finite products of matrices $\mathbf{A}^{-1}$ and $\mathbf{B}^{-1}$ form a binary tree with root the identity matrix (the empty product). Renaming $\mathbf{A}^{-1}$ as $\mathbf{L}$ and $\mathbf{B}^{-1}$ as $\mathbf{R}$, the tree can be displayed with "L" indicating a left branch and "R" indicating a right branch. Fig. 1 displays the first few levels of the tree.

That all matrices in the tree are different is proved by showing that the tree is a binary search tree (as formalised shortly). The key element of the proof[2] is that the determinants of $\mathbf{A}$ and $\mathbf{B}$ are both equal to 1 and, hence, the determinant of any finite product of $\mathbf{L}$s and $\mathbf{R}$s is also 1.

[2] The proof is an adaptation of the proof in [5, page 117] that the rationals in the Stern-Brocot tree are all different. Our use of determinants corresponds to their use of "the fundamental fact" (4.31). Note that the definitions of $\mathbf{L}$ and $\mathbf{R}$ are swapped around in [5]).

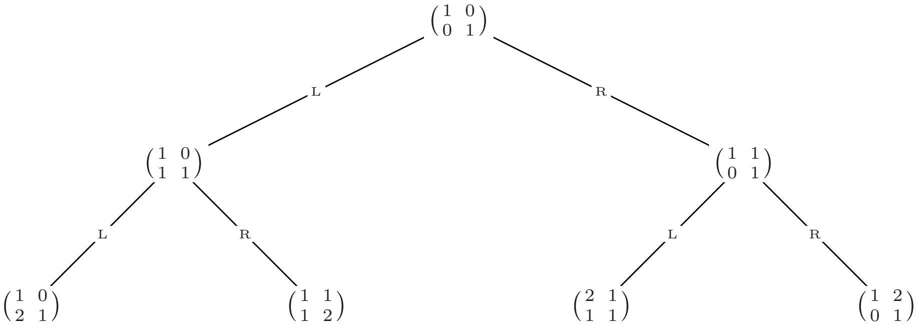

$\begin{pmatrix}1&0\\0&1\end{pmatrix}$
L
R
$\begin{pmatrix}1&0\\1&1\end{pmatrix}$
$\begin{pmatrix}1&1\\0&1\end{pmatrix}$
L
R
L
R
$\begin{pmatrix}1&0\\2&1\end{pmatrix}$
$\begin{pmatrix}1&1\\1&2\end{pmatrix}$
$\begin{pmatrix}2&1\\1&1\end{pmatrix}$
$\begin{pmatrix}1&2\\0&1\end{pmatrix}$

Fig. 1. Tree of Products of **L** and **R**

Formally, we define the relation $\prec$ on matrices that are finite products of **L**s and **R**s by

$$\begin{pmatrix}a&c\\b&d\end{pmatrix} \prec \begin{pmatrix}a'&c'\\b'&d'\end{pmatrix} \quad\equiv\quad \frac{a+c}{b+d} < \frac{a'+c'}{b'+d'} .$$

(Note that the denominator in these fractions is strictly positive; this fact is easily proved by induction.) We prove that, for all such matrices **X**, **Y** and **Z**,

$$\mathbf{X}{\times}\mathbf{L}{\times}\mathbf{Y} \prec \mathbf{X} \prec \mathbf{X}{\times}\mathbf{R}{\times}\mathbf{Z} \quad . \tag{2}$$

It immediately follows that there are no duplicates in the tree of matrices because the relation $\prec$ is clearly transitive and a subset of the inequality relation. (Property (2) formalises precisely what we mean by the tree of matrices forming a binary search tree: the entries are properly ordered by the relation $\prec$, with matrices in the left branch being “less than” the root matrix which is “less than” matrices in the right branch.)

In order to show that

$$\mathbf{X}{\times}\mathbf{L}{\times}\mathbf{Y} \prec \mathbf{X}, \tag{3}$$

suppose $\mathbf{X}{=}\begin{pmatrix}a&c\\b&d\end{pmatrix}$ and $\mathbf{Y}{=}\begin{pmatrix}a'&c'\\b'&d'\end{pmatrix}$. Then, since $\mathbf{L}{=}\begin{pmatrix}1&0\\1&1\end{pmatrix}$, (3) is easily calculated to be

$$\frac{(a+c){\times}a' + (c{\times}b') + (a+c){\times}c' + (c{\times}d')}{(b+d){\times}a' + (d{\times}b') + (b+d){\times}c' + (d{\times}d')} < \frac{a+c}{b+d} \quad .$$

That this is true is also a simple, albeit longer, calculation (which exploits the monotonicity properties of multiplication and addition); as observed earlier, the key property is that the determinant of **X** is 1, i.e. $a{\times}d - b{\times}c = 1$. The proof that $\mathbf{X}{\prec}\mathbf{X}{\times}\mathbf{R}{\times}\mathbf{Z}$ is similar.

Of course, we can also express Euclid’s algorithm in terms of transpose matrices. Instead of writing assignments to the vector $(x\;y)$, we can write assignments to its transpose $\begin{pmatrix}x\\y\end{pmatrix}$. Noting that **A** and **B** are each other’s transposition, the assignment

$$(x\;y),\mathbf{C} \;:=\; (x\;y){\times}\mathbf{A}\,,\mathbf{C}{\times}\mathbf{A}$$

in the body of Euclid's algorithm becomes

$$\begin{pmatrix} x \\ y \end{pmatrix} , \mathbf{C} \;\; := \;\; \mathbf{B}\times \begin{pmatrix} x \\ y \end{pmatrix} \; , \mathbf{B}\times\mathbf{C} \;\; .$$

Similarly, the assignment

$$(x\,y) , \mathbf{C} \;:=\; (x\,y)\times\mathbf{B} \,, \mathbf{C}\times\mathbf{B}$$

becomes

$$\begin{pmatrix} x \\ y \end{pmatrix} , \mathbf{C} \;\; := \;\; \mathbf{A}\times \begin{pmatrix} x \\ y \end{pmatrix} \; , \mathbf{A}\times\mathbf{C} \;\; .$$

On termination, the matrix $\mathbf{C}$ computed by the revised algorithm will of course be different; the pair $\begin{pmatrix} {}^{m}\!/_{(m\triangledown n)} \\ {}^{n}\!/_{(m\triangledown n)} \end{pmatrix}$ is recovered from it by the identity

$$\mathbf{C}^{-1}\times \begin{pmatrix} 1 \\ 1 \end{pmatrix} \;=\; \begin{pmatrix} {}^{m}\!/_{(m\triangledown n)} \\ {}^{n}\!/_{(m\triangledown n)} \end{pmatrix} .$$

In this way, we get a second bijection between the rationals and the finite products of the matrices $\mathbf{A}^{-1}$ and $\mathbf{B}^{-1}$. This is the basis for our second method of enumerating the rationals.

In summary, we have:

Theorem 1. Define the matrices $\mathbf{L}$ and $\mathbf{R}$ by

$$\mathbf{L} \;=\; \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \qquad \text{and} \qquad \mathbf{R} \;=\; \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} .$$

Then the following algorithm computes a bijection between the (positive) rationals and the finite products of $\mathbf{L}$ and $\mathbf{R}$. Specifically, the bijection is given by the function that maps the rational $\frac{m}{n}$ to the matrix $\mathbf{D}$ constructed by the algorithm together with the function from a finite product, $\mathbf{D}$, of $\mathbf{L}$s and $\mathbf{R}$s to $(1\,1)\times\mathbf{D}$. (The comments added to the algorithm supply the information needed to verify this assertion.)

```
$\{\;\; 0<m \wedge 0<n \;\;\}$
$(x\,y) , \mathbf{D} \;:=\; (m\,n) , \mathbf{I}\,;$
$\{\;\; \textbf{Invariant}\!: \quad (m\,n) \;=\; (x\,y)\times\mathbf{D} \;\;\}$
do $y<x \rightarrow (x\,y) , \mathbf{D} \;:=\; (x\,y)\times\mathbf{L}^{-1} , \mathbf{L}\times\mathbf{D}$
$\Box\;\; x<y \rightarrow (x\,y) , \mathbf{D} \;:=\; (x\,y)\times\mathbf{R}^{-1} , \mathbf{R}\times\mathbf{D}$
od
$\{\;\; (x\,y) \,=\, (m\triangledown n \;\, m\triangledown n) \quad\wedge\quad ({}^{m}\!/_{(m\triangledown n)} \;{}^{n}\!/_{(m\triangledown n)}) \,=\, (1\,1)\times\mathbf{D} \;\;\}$
```

Similarly, by applying the rules of matrix transposition to all expressions in the above, Euclid's algorithm constructs a second bijection between the rationals and finite products of the matrices $\mathbf{L}$ and $\mathbf{R}$. Specifically, the bijection is given by the function that maps the rational $\frac{m}{n}$ to the matrix $\mathbf{D}$ constructed by the revised algorithm together with the function from finite products, $\mathbf{D}$, of $\mathbf{L}$s and $\mathbf{R}$s to $\mathbf{D}\times \binom{1}{1}$. □

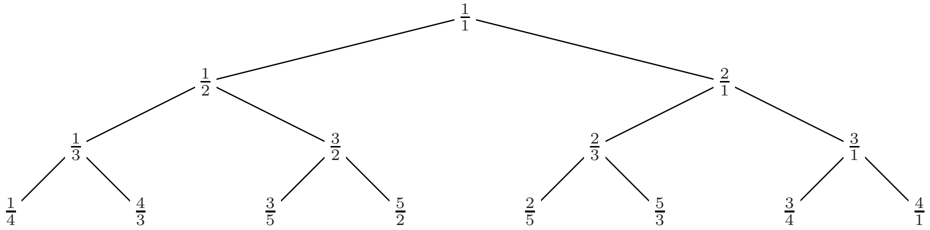

$\frac{1}{1}$
$\frac{1}{2}$ $\frac{2}{1}$
$\frac{1}{3}$ $\frac{3}{2}$ $\frac{2}{3}$ $\frac{3}{1}$
$\frac{1}{4}$ $\frac{4}{3}$ $\frac{3}{5}$ $\frac{5}{2}$ $\frac{2}{5}$ $\frac{5}{3}$ $\frac{3}{4}$ $\frac{4}{1}$

Fig. 2. Calkin-Wilf Tree of Rationals

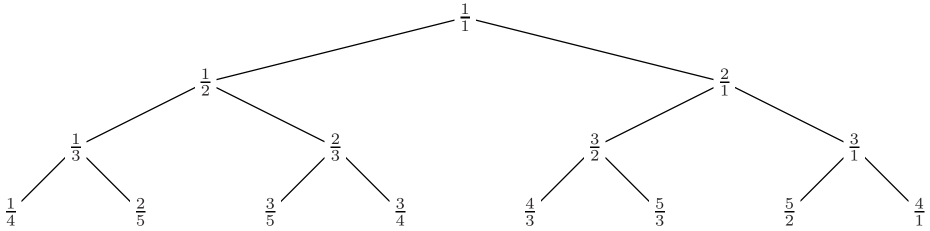

$\frac{1}{1}$
$\frac{1}{2}$ $\frac{2}{1}$
$\frac{1}{3}$ $\frac{2}{3}$ $\frac{3}{2}$ $\frac{3}{1}$
$\frac{1}{4}$ $\frac{2}{5}$ $\frac{3}{5}$ $\frac{3}{4}$ $\frac{4}{3}$ $\frac{5}{3}$ $\frac{5}{2}$ $\frac{4}{1}$

Fig. 3. Stern-Brocot Tree of Rationals

2.1 Enumerating Products of L and R

The problem of enumerating the rationals has been transformed to the problem of enumerating all finite products of the matrices **L** and **R**. As observed earlier, the matrices are naturally visualised as a tree —recall fig. 1— with left branching corresponding to multiplying (on the right) by **L** and right branching to multiplying (on the right) by **R**.

By premultiplying each matrix in the tree by $(1\,1)$, we get a tree of rationals. (Premultiplying by $(1\,1)$ is accomplished by adding the elements in each column.) This tree is called the Calkin-Wilf tree [4,1,2]. The first four levels of the tree are shown in fig. 2. In this figure, the vector $(x\,y)$ has been displayed as $\frac{y}{x}$. (Note the order of x and y. This is to aid comparison with existing literature.)

By postmultiplying each matrix in the tree by $\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$, we also get a tree of rationals. (Postmultiplying by $\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$ is accomplished by adding the elements in each row.) This tree is called the Stern-Brocot tree [5, pages 116–118]. See fig. 3. In this figure, the vector $\begin{pmatrix}x\\y\end{pmatrix}$ has been displayed as $\frac{x}{y}$.

Of course, if we can find an efficient way of enumerating the matrices in fig. 1, we immediately get an enumeration of the rationals as displayed in the Calkin-Wilf tree and as displayed in the Stern-Brocot tree — as each matrix is enumerated, simply premultiply by $(1\,1)$ or postmultiply by $\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$. Formally, the matrices are enumerated by enumerating all strings of Ls and Rs in lexicographic order, beginning with the empty string; each string is mapped to a matrix by the homomorphism that maps "L" to **L**, "R" to **R**, and string concatenation to matrix product. It is easy to enumerate all such strings; as we see shortly, converting strings to matrices is also not difficult, for the simple reason that **L** and **R** are invertible.

The enumeration proceeds level-by-level. Beginning with the unit matrix (level 0), the matrices on each level are enumerated from left to right. There are 2^k matrices on level k, the first of which is $\mathbf{L}^k$. The problem is to determine for a given matrix, which is the matrix "adjacent" to it. That is, given a matrix **D**, which is a finite product of **L** and **R**, and is different from $\mathbf{R}^k$ for all k, what is the matrix that is to the immediate right of **D** in fig. 1?

Consider the lexicographic ordering on strings of Ls and Rs of the same length. The string immediately following a string s (that is not the last) is found by identifying the rightmost L in s. Supposing s is the string tLR^j, where R^j is a string of j Rs, its successor is tRL^j.

It's now easy to see how to transform the matrix identified by s to its successor matrix. Simply postmultiply by $\mathbf{R}^{-j}\times\mathbf{L}^{-1}\times\mathbf{R}\times\mathbf{L}^j$. This is because, for all **T** and j,

$$(\mathbf{T}\times\mathbf{L}\times\mathbf{R}^j)\times(\mathbf{R}^{-j}\times\mathbf{L}^{-1}\times\mathbf{R}\times\mathbf{L}^j) \;=\; \mathbf{T}\times\mathbf{R}\times\mathbf{L}^j.$$

Also, it is easy to calculate $\mathbf{R}^{-j}\times\mathbf{L}^{-1}\times\mathbf{R}\times\mathbf{L}^j$. Specifically,

$$\mathbf{R}^{-j}\times\mathbf{L}^{-1}\times\mathbf{R}\times\mathbf{L}^j \;=\; \begin{pmatrix} 2j+1 & 1 \\ -1 & 0 \end{pmatrix}.$$

(We omit the details. Briefly, by induction, $\mathbf{L}^j$ equals $\left(\begin{smallmatrix} 1 & 0 \\ j & 1 \end{smallmatrix}\right)$. Also, **R** is the transpose of **L**.)

The final task is to determine, given a matrix **D**, which is a finite product of **L**s and **R**s, and is different from $\mathbf{R}^k$ for all k, the unique value j such that $\mathbf{D}=\mathbf{T}\times\mathbf{L}\times\mathbf{R}^j$ for some **T**. This can be determined by examining Euclid's algorithm once more.

The matrix form of Euclid's algorithm discussed in theorem 1 computes a matrix **D** given a pair of positive numbers m and n; it maintains the invariant

$$(m\,n) \;=\; (x\,y)\times\mathbf{D}.$$

D is initially the identity matrix and x and y are initialised to m and n, respectively; immediately following the initialisation process, **D** is repeatedly premultiplied by **R** so long as x is less than y. Simultaneously, y is reduced by x. The number of times that **D** is premultiplied by **R** is thus the greatest number j such that $j{\times}m$ is less than n, which is $\lfloor\frac{n-1}{m}\rfloor$. Now suppose the input values m and n are coprime. Then, on termination of the algorithm, $(1\,1)\times\mathbf{D}$ equals $(m\,n)$. That is, if

$$\mathbf{D} \;=\; \begin{pmatrix} \mathbf{D}_{00} & \mathbf{D}_{01} \\ \mathbf{D}_{10} & \mathbf{D}_{11} \end{pmatrix},$$

then,

$$\left\lfloor\frac{n-1}{m}\right\rfloor \;=\; \left\lfloor\frac{\mathbf{D}_{01}+\mathbf{D}_{11}-1}{\mathbf{D}_{00}+\mathbf{D}_{10}}\right\rfloor.$$

It remains to decide how to keep track of the levels in the tree. For this purpose, it is not necessary to maintain a counter. It suffices to observe that $\mathbf{D}$ is a power of $\mathbf{R}$ exactly when the rationals in the Calkin-Wilf, or Stern-Brocot, tree are integers, and this integer is the number of the next level in the tree (where the root is on level 0). So, it is easy to test whether the last matrix on the current level has been reached. Equally, the first matrix on the next level is easily calculated. For reasons we discuss in the next section, we choose to test whether the rational in the Calkin-Wilf tree is an integer; that is, we evaluate the boolean $\mathbf{D}_{00} + \mathbf{D}_{10} = 1$. In this way, we get the following (non-terminating) program which computes the successive values of $\mathbf{D}$.

```
$\mathbf{D} := \mathbf{I}$;
do $\mathbf{D}_{00} + \mathbf{D}_{10} = 1 \rightarrow \mathbf{D} := \begin{pmatrix} 1 & 0 \\ \mathbf{D}_{01} + \mathbf{D}_{11} & 1 \end{pmatrix}$
□ $\mathbf{D}_{00} + \mathbf{D}_{10} \neq 1 \rightarrow j := \left\lfloor \frac{\mathbf{D}_{01} + \mathbf{D}_{11} - 1}{\mathbf{D}_{00} + \mathbf{D}_{10}} \right\rfloor$ ; $\mathbf{D} := \mathbf{D} \times \begin{pmatrix} 2j+1 & 1 \\ -1 & 0 \end{pmatrix}$
od
```

A minor simplification of this algorithm is that the "$-\ 1$" in the assignment to j can be omitted. This is because $\left\lfloor \frac{n-1}{m} \right\rfloor$ and $\left\lfloor \frac{n}{m} \right\rfloor$ are equal when m and n are coprime and m is different from 1. We return to this shortly.

2.2 The Enumerations

As remarked earlier, we immediately get an enumeration of the rationals as displayed in the Calkin-Wilf tree and as displayed in the Stern-Brocot tree — as each matrix is enumerated, simply premultiply by $(1\,1)$ or postmultiply by $\binom{1}{1}$, respectively.

In the case of enumerating the Calkin-Wilf tree, several optimisations are possible. First, it is immediate from our derivation that the value assigned to the local variable j is a function of $(1\,1) \times \mathbf{D}$. In turn, the matrix $\begin{pmatrix} 2j+1 & 1 \\ -1 & 0 \end{pmatrix}$ is also a function of $(1\,1) \times \mathbf{D}$. Let us name the function J, so that the assignment becomes

$$\mathbf{D} \ := \ \mathbf{D} \times J.((1\,1) \times \mathbf{D}).$$

Then, the Calkin-Wilf enumeration iteratively evaluates

$$(1\,1) \times (\mathbf{D} \times J.((1\,1) \times \mathbf{D})).$$

Matrix multiplication is associative; so this is

$$((1\,1) \times \mathbf{D}) \times J.((1\,1) \times \mathbf{D}),$$

which is also a function of $(1\,1) \times \mathbf{D}$. Moreover —in anticipation of the current discussion— we have been careful to ensure that the test for a change in the level in the tree is also a function of $(1\,1)\times\mathbf{D}$. Combined together, this means that, in order to enumerate the rationals in Calkin-Wilf order, it is not necessary to compute $\mathbf{D}$ at each iteration, but only $(1\,1)\times\mathbf{D}$. Naming the two components of this vector m and n, and simplifying the matrix multiplications, we get[3]

```
$m{,}n \;:=\; 1{,}1\;;$
do $m{=}1 \;\rightarrow\; m{,}n \;:=\; n+1{,}m$
□  $m{\neq}1 \;\rightarrow\; m{,}n \;:=\; (2\left\lfloor\frac{n-1}{m}\right\rfloor+1)\times m-n\;,\;m$
od
```

At this point, a further simplification is also possible. We remarked earlier that $\lfloor\frac{n-1}{m}\rfloor$ equals $\lfloor\frac{n}{m}\rfloor$ when m and n are coprime and m is different from 1. By good fortune, it is also the case that $(2\lfloor\frac{n}{m}\rfloor+1)\times m-n$ simplifies to $n+1$ when m is equal to 1. That is, the elimination of "$-\ 1$" in the evaluation of the floor function leads to the elimination of the entire case analysis! This is the algorithm attributed to Newman in [6].

```
$m{,}n \;:=\; 1{,}1\;;$
do  $m{,}n \;:=\; (2\left\lfloor\frac{n}{m}\right\rfloor+1)\times m-n\;,\;m$
od
```

3 Discussion

This paper was motivated by reading two publications, [5, pages 116–118] and [4]. Gibbons, Lester and Bird [4] show how to enumerate the elements of the Calkin-Wilf tree, but claim that "it is not at all obvious how to do this for the Stern-Brocot tree". Specifically, they say:

> However, there is an even better compensation for the loss of the ordering property in moving from the Stern-Brocot to the Calkin-Wilf tree: it becomes possible to deforest the tree altogether, and generate the rationals directly, maintaining no additional state beyond the 'current' rational. This startling observation is due to Moshe Newman (Newman, 2003). In contrast, it is not at all obvious how to do this for the Stern-Brocot tree; the best we can do seems to be to deforest the tree as far as its levels, but this still entails additional state of increasing size.

In this paper, we have shown that it is possible to enumerate the rationals in Stern-Brocot order without incurring "additional state of increasing size". More importantly, we have presented *one* enumeration algorithm with *two* specialisations, one being the Calkin-Wilf enumeration they present, and the other being the Stern-Brocot enumeration that they described as being "not at all obvious".

[3] Recall that, to comply with existing literature, the enumerated rational is $\frac{n}{m}$ and not $\frac{m}{n}$.

The optimisation of Calkin-Wilf enumeration which leads to Newman's algorithm is not possible for Stern-Brocot enumeration. Nevertheless, the complexity of Stern-Brocot enumeration is the same as the complexity of Newman's algorithm, both in time and space. The only disadvantage of Stern-Brocot enumeration is that four variables are needed in place of two; the advantage is the (well-known) advantage of the Stern-Brocot tree over the Calkin-Wilf tree — the rationals on a given level are in ascending order.

Gibbons, Lester and Bird's goal seems to have been to show how the functional programming language Haskell implements the various constructions – the construction of the tree structures and Newman's algorithm. In doing so, they repeat the existing mathematical presentations of the algorithms as given in [2,5,6]. The ingredients for an efficient enumeration of the Stern-Brocot tree are all present in these publications, but the recipe is missing!

The fact that expressing the rationals in "lowest form" is essential to the avoidance of duplication in any enumeration immediately suggests the relevance of Euclid's algorithm. The key to our exposition is that Euclid's algorithm can be expressed in terms of matrix multiplications, where —significantly— the underlying matrices are invertible. Transposition and inversion of the matrices capture the symmetry properties in a precise, calculational framework. As a result, the bijection between the rationals and the tree elements is immediate and we do not need to give separate, inductive proofs for both tree structures. Also, the determination of the next element in an enumeration of the tree elements has been reduced to one unifying construction.

Acknowledgements. Thanks go to Jeremy Gibbons for his comments on earlier drafts of this paper, and for help with TEX commands. Thanks also to our colleagues in the Nottingham Tuesday Morning Club for helping iron out omissions and ambiguities.

This paper was submitted in April 2007 to the American Mathematical Monthly; it was rejected in November 2007 on the grounds that it was not of sufficient interest to readers of the Monthly. One (of two referees) did, however, recommend publication. The referee made the following general comment.

> Each of the two trees of rationals—the Stern-Brocot tree and the Calkin-Wilf tree—has some history. Since this paper now gives the definitive link between these trees, I encourage the authors, perhaps in their Discussion section, to also give the definitive histories of these trees, something in the same spirit as the Remarks at the end of the Calkin and Wilf paper.

We thank the referee for the detailed comments; unfortunately, we have not been able to obtain copies of the original papers by Stern and Brocot —we would have had difficulty reading the German and French in any case— and are not in a position to fulfill the referee's request. It would, indeed, be interesting for a mathematical historian to pursue this suggestion.

References

1. Aigner, M., Ziegler, G.: Proofs From The Book, 3rd edn. Springer, Heidelberg (2004)
2. Calkin, N., Wilf, H.S.: Recounting the rationals. The American Mathematical Monthly 107(4), 360–363 (2000)
3. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453–457 (1975)
4. Gibbons, J., Lester, D., Bird, R.: Enumerating the rationals. Journal of Functional Programming 16(3), 281–291 (2006)
5. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics: a Foundation for Computer Science, 2nd edn. Addison-Wesley Publishing Company, Reading (1994)
6. Knuth, D.E., Rupert, C.P., Smith, A., Stong, R.: Recounting the rationals, continued. American Mathematical Monthly 110(7), 642–643 (2003)

A Appendix: Haskell Implementation

This appendix contains an encoding of the enumeration algorithms in Haskell. The file from which this printed version was compiled is a so-called “lhs2TEX” file[4] which can be used directly as input to a Haskell compiler; this safeguards against typographical errors in the printed paper.

The implementation encodes a matrix as a list of *columns*.

```
type Entry  = Integer
type Column = [Entry]
type Matrix = [Column]
```

We define a type of non-empty trees, with associated map, fold and unfold functions.

```
data Tree a              = Node (a, Tree a, Tree a)
mapt f (Node (a, l, r))  = Node (f a, mapt f l, mapt f r)
foldt f (Node (a, l, r)) = f (a, foldt f l, foldt f r)
unfoldt f x              = let (a, y, z) = f x
                           in Node (a, unfoldt f y, unfoldt f z)
```

With matrices *matId*, *matL* and *matR* defined to be the identity matrix, the matrix **L** and the matrix **R**, respectively, the tree of matrices is generated as follows.

```
mTree :: Tree Matrix
mTree = unfoldt level matId
  where level m = (m, m × matL, m × matR)
```

The Calkin-Wilf tree can be obtained by pre-multiplying the matrices by the vector $(1\ \ 1)$.

[4] The lhs2TEX system has been implemented by Ralf Hinze and Andres Löh.

```
cwTree :: Tree Rational
cwTree = mapt (mkCWRat ∘ ([[1], [1]]×)) mTree


mkCWRat :: Matrix → Rational
mkCWRat [[m], [n]] = n/m
```

Similarly, the Stern-Brocot tree can be obtained by post-multiplying the matrices by the transpose of the vector $(1\ \ 1)$, i.e., $\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$.

```
sbTree :: Tree Rational
sbTree = mapt (mkSBRat ∘ (×[[1, 1]])) mTree


mkSBRat :: Matrix → Rational
mkSBRat [[m, n]] = m/n
```

We enumerate the matrices using the *iterate* function, computing each matrix from the previous one.

```
nextM :: Matrix → Matrix
nextM [[1, 0], [n, 1]] = [[1, n + 1], [0, 1]]
nextM [c0, c1] = let j  = ⌊((sum c1) − 1)/(sum c0)⌋
                     k  = 2×j + 1
                     ck = map (k×) c0
                 in  [zipWith (−) ck c1, c0]


mats :: [Matrix]
mats = iterate nextM matId
```

The (non-optimised) implementation of the Calkin-Wilf enumeration is then a matter of premultiplying by $(1\ \ 1)$.

```
cwEnum :: [Rational]
cwEnum = map mkCW mats
  where mkCW = mkCWRat ∘ ([[1], [1]]×)
```

The Stern-Brocot enumeration can be defined in a similar way, but instead of premultiplying, we postmultiply by $\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$:

```
sbEnum :: [Rational]
sbEnum = map mkSB mats
  where mkSB = mkSBRat ∘ (×[[1, 1]])
```

Incorporating the optimisations discussed above, the Calkin-Wilf enumeration is transformed to the algorithm attributed to Newman.

```
cwnEnum :: [Rational]
cwnEnum = iterate nextCW 1/1
```

```
nextCW :: Rational → Rational
nextCW r = let (n, m) = (numerator r, denominator r)
               j      = ⌊n/m⌋
           in  m/((2×j + 1)×m − n)
```

Zippy Tabulations of Recursive Functions

Richard S. Bird

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
bird@comlab.ox.ac.uk

> **zippy**: (adjective) 1. bright,
> fresh or lively. 2. speedy
> *Oxford Compact Dictionary*

Abstract. This paper is devoted to the statement and proof of a theorem showing how recursive definitions whose associated call graphs satisfy certain shape conditions can be converted systematically into efficient bottom-up tabulation schemes. The increase in efficiency can be dramatic, typically transforming an exponential time algorithm into one that takes only quadratic time. The proof of the theorem relies heavily on the theory of zips developed by Roland Backhouse and Paul Hoogendijk.

1 Introduction

From one point of view all recursive functions look alike: if the input is simple the function value is computed directly; if the input is not simple, it is decomposed into simpler instances on which the function is computed recursively, and the various results are then combined to give the final value. In a divide and conquer algorithm, such as mergesort, the recursive instances do not overlap so the top-down computation is efficient. By contrast, in a dynamic programming algorithm the recursive instances do overlap, and the top-down computation is potentially very inefficient because the same subproblem may be solved many times.

The two main methods for improving matters are *memoization* and *tabulation*. In memoization the top-down structure of the computation is preserved but computed results are remembered and stored in a memo table for subsequent retrieval. In tabulation the computation switches to a bottom-up scheme in which the simple problems are solved first and then solutions to larger problems are placed in a table level by level.

The problem with tabulation is that the programmer has to think carefully about the structure of the table being built and the best way to store the entries to make them easily accessible for the next level. This is particularly true of the pure functional programmer who eschews arrays and relies on lists and trees instead. Wouldn't it be pleasant if the bottom-up tabulation algorithm could be derived automatically, or at least systematically from the top-down recursion?

The wish is not grantable in general. For example, [8] shows that choosing a good table design for a given decomposition is an NP-hard problem. However, it turns out that for one particular class of recursive functions, namely

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 92–109, 2008.

those involving decompositions which satisfy a rather severe shape constraint, a bottom-up tabulation scheme can indeed be derived automatically. The proof that everything works is the main contribution of the present paper. The proof is interesting because it depends heavily on the theory of zips developed by Backhouse and Hoogendijk, see [1,2,6,7]. Our main theorem can also be viewed as an extension and formalisation of some of bottom-up schemes described in [3].

2 Top-Down Computation

Let us begin by assembling the ingredients for describing a generic top-down recursion. The first thing to say is that although we use mostly Haskell notation to describe recursive functions, we do not assume a lazy semantics. Indeed for the theorem and proof to come our setting is a relational semantics based on allegories with certain properties, see [4]. But the allegorical undergrowth need not impede our progress.

The first assumption is that the input is an element of some instance of a *polymorphic* data type $L\,a$ with no empty structures. For concreteness think of $L\,a$ as being nonempty lists of elements of type a, and the input as an element of $L\,\mathit{Integer}$, a list of integers. For technical reasons, explained later on, we disallow data types that contain empty elements. We use single capital letters for the names of polymorphic types, and the same letter for the map function over the data type. So if $f :: a \to b$, then $L f :: L\,a \to L\,b$. In a word, L is a functor.

The next ingredient is a total function $sg :: L\,a \to \mathit{Bool}$ that determines whether the input is sufficiently simple for the result to be returned directly. The identifier sg connotes 'singleton', which is often the base case of a recursion though other interpretations are possible.

The companion to sg is a partial function $ex :: L\,a \to a$, total on arguments that satisfy sg, that extracts the singleton element. Both sg and ex are polymorphic, which means they satisfy the properties that $sg \cdot L f = sg$ and $ex \cdot L f = f \cdot ex$ for any total function f. In two words, sg and ex are natural transformations. In general, a polymorphic function $\alpha :: F\,a \to G\,a$ satisfies $\alpha \cdot F f = G f \cdot \alpha$ for any total function f. In particular, $ex :: L\,a \to I\,a$, where I is the identity functor, and $sg :: L\,a \to K\,a$, where $K\,a = \mathit{Bool}$ is the constant functor that returns Bool. Moreover, $K f$ is the identity function on Bool. We will exploit these naturality assumptions later on.

Next is a partial function $dc :: L\,a \to F\,(L\,a)$ which is total over non-singleton structures. In words, dc returns an F-structure of L-structures. The identifier dc connotes 'decompositions'. The function dc is, by assumption, another natural transformation so we have $F\,(L f) \cdot dc = dc \cdot L f$ for any total function f.

Everything is now in place for describing a top-down computation:

$$\begin{aligned} td\ \ \ \ &:: (a \to b) \to (F\,b \to b) \to L\,a \to b \\ td\,f\,g &= (sg \to f \cdot ex, g \cdot F\,(td\,f\,g) \cdot dc) \end{aligned}$$

The expression on the right is a McCarthy conditional $(p \to f, g)$. Applied to x the conditional returns $f\,x$ if $p\,x$ and $g\,x$ otherwise. The computation of td

reads: if the input is a singleton, extract the element and apply f; otherwise decompose the input into further instances using dc, apply $td\,f\,g$ recursively to each of these instances and collect the sub-results into one result by applying g.

Before going on to formulate a bottom-up tabulation scheme let us first consider some instructive examples of top-down computations.

3 Examples

Bear in mind that the main restriction on td is that the function dc should be polymorphic. That rules out quicksort as an example because dc has to inspect the elements in the argument list. There are many examples of recursive top-down computations with polymorphic decomposition functions, and we will look only at examples where the function $dc :: L\,a \rightarrow F\,(L\,a)$ is *layered* in the sense that dc produces an F-structure of L-structures in which all the L-structures have the same shape. This restriction turns out to be central in the results to come.

First, consider mergesort restricted to power lists. By definition a power list is a list whose length is a power of two. Here $L\,a$ is the type of power lists and $F = P$, where $P\,a = (a, a)$ so P is the data type of pairs. The function sg is a test for singletons and ex extracts a singleton value. The definition of dc is

$$dc\,xs = (take\,n\,xs, drop\,n\,xs) \quad \textbf{where}\ n = length\,xs\ \textbf{div}\ 2$$

Applied to a power list, dc returns a pair of power lists of the same length, so is layered. The function $td\,wrap\,merge$ sorts a power list of elements from any ordered type, where $wrap\,x = [x]$ and $merge$ merges a pair of ordered power lists. The call graph associated with dc is a perfect binary tree. The subproblems do not overlap and there is no sharing of subtrees.

Second, consider a recursion based on the decomposition function

$$\begin{aligned} &dc :: L\,a \rightarrow P\,(L\,a)\\ &dc = fork\,(init, tail)\end{aligned}$$

where *init* and *tail* respectively drop the last and first elements of a list of length at least two. Here L is the data type of nonempty lists and P is again the data type of pairs. The useful function *fork* is defined by

$$\begin{aligned} &fork && :: (a \rightarrow b, a \rightarrow c) \rightarrow a \rightarrow (b, c)\\ &fork\,(f, g)\,x && = (f\,x, g\,x)\end{aligned}$$

The functions sg and ex are as for mergesort. The function dc returns a pair of lists of the same length, so is layered. The call graph of dc is pictured for an input list of length 5 in Figure 1. Observe that this graph is also a perfect binary tree except that subtrees are shared. A tree with shared nodes was called a *nexus* in [3] and we will henceforth adopt this terminology. Evaluating $td\,f\,g$ takes exponential time assuming f and g take constant time.

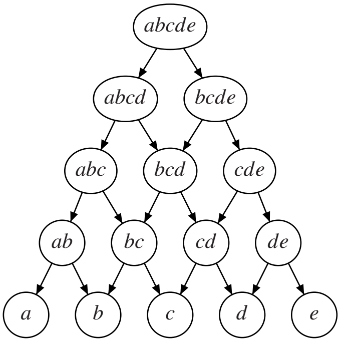


Fig. 1. A nexus

Third, consider a problem about matrices . Each $(n+1) \times (n+1)$ matrix has four $n \times n$ corner matrices. For example, the matrix

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix}$$

has four corner matrices

$$\begin{pmatrix} a & b & c \\ e & f & g \\ i & j & k \end{pmatrix} \begin{pmatrix} b & c & d \\ f & g & h \\ j & k & l \end{pmatrix} \begin{pmatrix} e & f & g \\ i & j & k \\ m & n & o \end{pmatrix} \begin{pmatrix} f & g & h \\ j & k & l \\ n & o & p \end{pmatrix}$$

Imagine a function defined on a matrix by splitting the matrix into its four corners, recursively computing the value of the function on these corners, and constructing a further value out of the four results. If the matrix is a singleton, then the value is computed directly. The type of a matrix is a list of lists, so $L\,a = [[a]]$. We can also take $F\,a = [a]$, though a special type of quadruples would serve equally well. We omit the definition of dc but it is clearly layered. The associated nexus is a perfect quaternary tree with sharing and the computation of $td\,f\,g$ takes 4^n steps on a $n \times n$ matrix, assuming f and g take constant time.

Finally, here is a problem in which the function dc is not polymorphic but can be converted into one that is. The value $comb\,(n, r)$ is the number of combinations of n objects taken r at a time. As a total function we can define

$$\begin{array}{lll} comb & :: P\ Integer \rightarrow Integer & \\ comb\,(n,r) & \mid\ n < r \lor r < 0 & = 0 \\ & \mid\ n = r \lor r = 0 & = 1 \\ & \mid\ n > r \land r > 0 & = comb\,(n{-}1, r) + comb\,(n{-}1, r{-}1) \end{array}$$

We set $comb\,(n, r) = 0$ if n and r do not satisfy $0 \leq r \leq n$. The associated decomposition, singleton test and extraction functions are not polymorphic but can be made so with a change of representation. Suppose (n, r) is represented by a binary string of length $n{+}1$ containing a single 1 at position r if $0 \leq r \leq n$, and all 0s otherwise. With this representation $dc = \mathit{fork}\,(\mathit{init}, \mathit{tail})$ because if xs represents (n, r), then $\mathit{init}\,xs$ represents $(n{-}1, r)$ and $\mathit{tail}\,xs$ represents $(n{-}1, r{-}1)$. Even better, at the expense of a little extra computation the functions sg and ex can be defined as for mergesort because a string of 0s has only singleton 0s below it, and the sum of such singletons is 0. In other words, we can base the computation of *comb* on the nexus of Figure 1 and define

$$\mathit{comb} = \mathit{td}\;\mathit{id}\;(\mathit{uncurry}\,(+)) \cdot \mathit{rep}$$

where $\mathit{rep} :: P\;\mathit{Int} \to L\;\mathit{Int}$ installs the representation. This version is somewhat less efficient than the direct definition because of the unnecessary computations of 0, but both still take exponential time. And tabulation can reduce this to quadratic time as we will now see.

4 Bottom-Up Tabulation

Look again at Figure 1. The idea behind bottom-up tabulation is simply to replace each label x of a node n with $\mathit{td}\,f\,g\,x$. This is achieved by applying g to appropriate combinations of the labels of the nodes below n. The nexus takes the form of a labelled tree

$$\textbf{data}\; N\;a = \mathit{Leaf}\;a \mid \mathit{Node}\;a\,(F\,(N\;a))$$

A node of a nexus consists of a label of type a and an F-structure of nexuses, where F is the same data type that appears in the type of dc.

The bottom-up algorithm takes the form

$$\begin{aligned} &\mathit{bu} && :: (a \to b) \to (F\;b \to b) \to L\,a \to b \\ &\mathit{bu}\,f\;g && = \mathit{label} \cdot \mathit{ex} \cdot \mathit{until}\;\mathit{sg}\,(L\,(\mathit{node}\;g) \cdot \mathit{cd}) \cdot L\,(\mathit{leaf}\;f) \end{aligned}$$

where the standard function *until* can be defined by

$$\begin{aligned} &\mathit{until} && :: (a \to \mathit{Bool}) \to (a \to a) \to a \to a \\ &\mathit{until}\,p\,f && = (p \to \mathit{id}, \mathit{until}\,p\,f \cdot f) \end{aligned}$$

The computation of $\mathit{bu}\,f\;g$ begins with some L-structure of values. The first step is to apply $L\,(\mathit{leaf}\;f)$ and the result is an L-structure of nexuses, all of which are leaves; this gives the first level of the table. Then the function $L\,(\mathit{node}\;g) \cdot \mathit{cd}$ is applied repeatedly until the result is a singleton L-structure. From this singleton a single nexus is extracted and its label is returned as the final result of the computation.

Here are the types and definitions of the ingredient functions. The function *leaf* is defined by

$$\begin{aligned} &\mathit{leaf} && :: (a \to b) \to a \to N\;b \\ &\mathit{leaf}\;f && = \mathit{Leaf} \cdot f \end{aligned}$$

The companion function *node* is defined by

$$\begin{aligned}
&\mathit{node} && :: (F\ b \to b) \to F\,(N\ b) \to N\ b \\
&\mathit{node}\ g && = \mathit{uncurry}\ \mathit{Node} \cdot \mathit{fork}\ (g \cdot F\ \mathit{label}, \mathit{id})
\end{aligned}$$

The function *label* extracts the label of a nexus:

$$\begin{aligned}
&\mathit{label} && :: N\ b \to b \\
&\mathit{label}\ (\mathit{Leaf}\ x) && = x \\
&\mathit{label}\ (\mathit{Node}\ (x, \mathit{fns})) && = x
\end{aligned}$$

Finally, the remaining function *cd* has type $cd :: L\,a \to L\,(F\ a)$. In particular, if $g :: F\ B \to B$ for some type B, then $L\,(\mathit{node}\ g) \cdot cd :: L\,(N\ B) \to L\,(N\ B)$. The functions $dc :: L\,a \to F\ (L\,a)$ and $cd :: L\,a \to L\,(F\ a)$ have dual types so we give them dual names.

One obvious definition of *cd*, though by no means the only one, is suggested by its type. The definition is

$$cd = \mathit{zip}(F, L) \cdot dc$$

where $\mathit{zip}(F, L) :: F\ (L\,a) \to L\,(F\ a)$ zips an F-structure of L-structures into an L-structure of F-structures. Zips are, in general, partial operations: $\mathit{zip}(F, L)$ is well-defined only when applied to an F-structure of L-structures in which all the L-structures have the same shape. For example, the transpose function $\mathit{zip}(L, L)$ on matrices, i.e. elements of $L\,(L\,a)$ where $L\,a = [a]$, is well-defined only if the matrix is a nonempty list of nonempty lists all of the same length. The restriction to nonempty lists is required because matrix transpose on an empty matrix is essentially a nondeterministic operation, returning an arbitrary list of empty lists (in Haskell this problem is resolved by having transpose return an infinite list of empty lists). When F and L contain no empty structures, both $\mathit{zip}(F, L)$ and $\mathit{zip}(L, F)$ are partial functions. This explains our restriction to data types without empty elements. Zips are required to have additional properties, described in Section 6.

Recall that *dc* is *layered* if *dc* returns an F-structure of L-structures all of the same shape; in such a case the domain of $\mathit{zip}(F, L) \cdot dc$ is that of *dc*.

Theorem 1. *bu f g = td f g provided that dc is layered and cd is any function satisfying the following three conditions:*

$$\begin{aligned}
sg \cdot cd &= sg \cdot \mathit{zip}(F, L) \cdot dc \\
ex \cdot cd &= F\ ex \cdot dc \\
dc \cdot cd &= F\ cd \cdot dc
\end{aligned}$$

The second condition of Theorem 1 is required to hold only if *cd* returns a result that satisfies *sg*, so $ex \cdot cd$ is well-defined. Similarly, the third condition is required to hold only if *cd* returns a result not satisfying *sg*, so $dc \cdot cd$ is well-defined.

Corollary 1. *With* $cd = \mathit{zip}(F, L) \cdot dc$ *we have bu f g = td f g provided that dc is layered and also satisfies the symmetry condition*

$$F\ dc \cdot dc = \mathit{zip}(F, F) \cdot F\ dc \cdot dc$$

Section 6 is devoted to the proofs of Theorem 1 and Corollary 1. First we revisit the examples in Section 3 to see what the symmetry condition entails.

5 Examples Revisited

Consider again the function dc arising in mergesort over power lists; as we have seen, dc is layered. Define $\mathit{cd} :: L\,a \rightarrow L\,(P\,a)$ by

$$\mathit{cd} = \mathit{zip}(P, L) \cdot \mathit{fork}\,(\mathit{even}, \mathit{odd})$$

where even applied to a power list returns the elements in even position; similarly for odd. The function $\mathit{zip}(P, L)$ zips a pair of power lists into a power list of pairs. For example,

$$\mathit{cd}\,[0\,..\,7] = [(0, 1), (2, 3), (4, 5), (6, 7)]$$

The three conditions of Theorem 1 are easy to check and we omit details. As a result the function $\mathit{bu}\;\mathit{wrap}\;\mathit{merge}$ defines an iterative version of mergesort. There is no asymptotic increase in time efficiency because there is no sharing of subtrees.

Next, consider the function $\mathit{dc} = \mathit{fork}\,(\mathit{init}, \mathit{tail})$. Here we aim for an application of Corollary 1, so we take $\mathit{cd} = \mathit{zip}(P, L) \cdot \mathit{dc}$. To appreciate that dc is symmetric, observe that the result of applying $P\,\mathit{dc} \cdot \mathit{dc}$ to the string "abcde" is

$$((\text{“abc”}, \text{“bcd”}), (\text{“bcd”}, \text{“cde”}))$$

Now, the definition of $\mathit{zip}(P, P)$, the zip function for pairs of pairs, is

$$\begin{array}{ll} \mathit{zip}(P, P) & :: P\,(P\,a) \rightarrow P\,(P\,a) \\ \mathit{zip}(P, P)\,((x, y), (u, v)) & = ((x, u), (y, v)) \end{array}$$

Applying $\mathit{zip}(P, P)$ to the above pair of pairs leaves it unchanged. So dc is indeed symmetric and Corollary 1 is applicable. The bottom-up algorithm $\mathit{bu}\,f\;g$ takes quadratic time assuming f and g take constant time.

The matrix example is similar. The function dc returns a list of matrices all of the same shape, and also satisfies the symmetry condition

$$\mathit{zip}(L, L) \cdot L\,\mathit{dc} \cdot \mathit{dc} = L\,\mathit{dc} \cdot \mathit{dc}$$

where $\mathit{zip}(L, L)$ is matrix transpose. For example, applying $L\,\mathit{dc} \cdot \mathit{dc}$ to the 4×4 matrix of Section 3 gives the symmetric matrix

$$\begin{pmatrix} \begin{pmatrix} a & b \\ e & f \end{pmatrix} & \begin{pmatrix} b & c \\ f & g \end{pmatrix} & \begin{pmatrix} e & f \\ i & j \end{pmatrix} & \begin{pmatrix} f & g \\ j & k \end{pmatrix} \\ \begin{pmatrix} b & c \\ f & g \end{pmatrix} & \begin{pmatrix} c & d \\ g & h \end{pmatrix} & \begin{pmatrix} f & g \\ j & k \end{pmatrix} & \begin{pmatrix} g & h \\ k & l \end{pmatrix} \\ \begin{pmatrix} e & f \\ i & j \end{pmatrix} & \begin{pmatrix} f & g \\ j & k \end{pmatrix} & \begin{pmatrix} i & j \\ m & n \end{pmatrix} & \begin{pmatrix} j & k \\ n & o \end{pmatrix} \\ \begin{pmatrix} f & g \\ j & k \end{pmatrix} & \begin{pmatrix} g & h \\ k & l \end{pmatrix} & \begin{pmatrix} j & k \\ n & o \end{pmatrix} & \begin{pmatrix} k & l \\ o & p \end{pmatrix} \end{pmatrix}$$

Indeed the term symmetric was chosen in analogy with the idea of a symmetric matrix. It can now be appreciated that the condition that dc be layered means that the associated nexus is structured into levels with connections only between one level and the next, and the symmetry condition in the case $F\ a = [a]$ means that all nodes have the same number, n say, of children, and adjacent nodes at one level are connected to adjacent n tuples of nodes at the next level. So the symmetry condition is quite severe.

6 Proofs

Let us first assume Theorem 1 and show how Corollary 1 follows. The proof of Corollary 1 depends on various properties of zips, first formulated by Backhouse and Hoogendijk and recorded in a sequence of papers [1,2,6,7]. These properties are described in a calculus of relations rather than functions because of the need to reason about partial and nondeterministic operations.

In particular, as we said earlier, $\mathit{zip}(F, G) :: F\,(G\,a) \rightarrow G\,(F\,a)$ is a partial operation which is only well-defined on F-structures of G-structures all of the same shape. In such a case the result is a G-structure of F-structures all of the same shape. As well as being partial, zips can also be nondeterministic. For example, take $F\ a = \mathit{Int}$. In this case $\mathit{zip}(F, G)$ takes an integer n to an arbitrary G-structure of copies of n and so is a nondeterministic operation. A similar example is matrix transpose when the empty matrix is allowed. But if we exclude empty F-structures, an assumption captured by the restriction $F\,0 = 0$ where 0 is the empty relation, then $\mathit{zip}(F, G)$ is a deterministic operation, though still partial.

We will need four properties of zips. The first is that $\mathit{zip}(F, G)$ is a natural transformation from FG to GF. The second property is that $\mathit{zip}(F, I)$ is the identity function on F. The third property is the law of composition:

$$\mathit{zip}(F, GH) = G\,(\mathit{zip}(F, H)) \cdot \mathit{zip}(F, G)$$

Finally, zips enjoy a *higher-order naturality* property. Restricted to data types F that do not contain empty elements, this property states that if $f :: G\,a \rightarrow H\,a$ is a polymorphic function, then

$$f \cdot \mathit{zip}(F, G) = \mathit{zip}(F, H) \cdot F\,f \cdot \mathit{zip}(F, G)\,\triangleright$$

Here $\mathit{zip}(F, G)\,\triangleright$ is a *coreflexive* relation, a sub-relation of the identity function, which holds only elements in the domain of $\mathit{zip}(F, G)$. Coreflexive relations are also known as partial skips. In particular, dc is a layered function just in the case that $dc = \mathit{zip}(F, L)\,\triangleright \cdot dc$. It is proved in [6] that zips with these properties can be defined for all the data types one normally encounters in functional programming.

Setting $cd = \mathit{zip}(F, L) \cdot dc$ we can now verify the three conditions of Theorem 1. The first condition is immediate from the definition of cd. The second condition, namely $ex \cdot cd = F\ ex \cdot dc$, is proved as follows:

$$\begin{array}{ll} & ex \cdot cd \\ = & \{\text{given definition of } cd\} \\ & ex \cdot zip(F, L) \cdot dc \\ = & \{\text{higher-order naturality of zips and } ex :: L\,a \to I\ a\} \\ & zip(F, I) \cdot F\ ex \cdot zip(F, L) \triangleright \cdot dc \\ = & \{\text{identity property of zips and assumption that } dc \text{ is layered}\} \\ & F\ ex \cdot dc \end{array}$$

Finally, to show that $dc \cdot cd = F\ cd \cdot dc$ we reason:

$$\begin{array}{ll} & dc \cdot zip(F, L) \cdot dc \\ = & \{\text{higher-order naturality of zips and } dc :: L\,a \to F\ (L\,a)\} \\ & zip(F, FL) \cdot F\ dc \cdot zip(F, L) \triangleright \cdot dc \\ = & \{\text{since } dc \text{ is layered}\} \\ & zip(F, FL) \cdot F\ dc \cdot dc \\ = & \{\text{composition property of zips}\} \\ & F\ zip(F, L) \cdot zip(F, F) \cdot F\ dc \cdot dc \\ = & \{\text{assumption that } dc \text{ is symmetric}\} \\ & F\ zip(F, L) \cdot F\ dc \cdot dc \\ = & \{\text{functor composition}\} \\ & F\ (zip(F, L) \cdot dc) \cdot dc \end{array}$$

This completes the proof of Corollary 1.

6.1 Proof of Theorem 1

Now we turn to the proof of Theorem 1. Proofs, like computations, can be top down or bottom up. We will proceed top down, collecting claims for subsequent subproofs. A number of steps involve the two basic laws of McCarthy conditionals, namely

$$\begin{array}{l} (p \to f, g) \cdot h = (p \cdot h \to f \cdot h, g \cdot h) \\ h \cdot (p \to f, g) = (p \to h \cdot f, h \cdot g) \end{array}$$

Use of these laws is signalled with the hint 'conditionals'. Observe that by applying the first law of conditionals twice, we have that

$$(p \to f, g) \cdot h = (p \to f', g') \cdot h$$

whenever $f \cdot h = f' \cdot h$ and $g \cdot h = g' \cdot h$. Finally, frequent use is made of the functor law $F(f \cdot g) = F f \cdot F\ g$ without explicit mention.

The proof of Theorem 1 is by induction. We show that $bu_n = td_n$ for all n, where bu_0 and td_0 are each the empty relation (i.e., the everywhere undefined function) and

$$\begin{aligned} bu_{n+1} &= \mathit{label} \cdot \mathit{ex} \cdot un_{n+1} \cdot L\,(\mathit{leaf}\,f) \\ un_{n+1} &= (sg \to id, un_n \cdot L\,(\mathit{node}\,g) \cdot cd) \\ td_{n+1} &= (sg \to f \cdot ex, g \cdot F\,td_n \cdot dc) \end{aligned}$$

The function un abbreviates $\mathit{until}\;sg\,(L\,(\mathit{node}\,g) \cdot cd)$. The base case is immediate and the induction step is:

$$\begin{aligned}
& bu_{n+1} \\
= \quad & \{\text{definition}\} \\
& \mathit{label} \cdot ex \cdot (sg \to id, un_n \cdot L\,(\mathit{node}\,g) \cdot cd) \cdot L\,(\mathit{leaf}\,f) \\
= \quad & \{\text{conditionals and } sg \cdot L\,(\mathit{leaf}\,f) = sg\} \\
& (sg \to \mathit{label} \cdot ex \cdot L\,(\mathit{leaf}\,f), \mathit{label} \cdot ex \cdot un_n \cdot L\,(\mathit{node}\,g) \cdot cd \cdot L\,(\mathit{leaf}\,f)) \\
= \quad & \{\text{claim: see (1) below}\} \\
& (sg \to \mathit{label} \cdot ex \cdot L\,(\mathit{leaf}\,f), \mathit{label} \cdot \mathit{node}\,g \cdot F\,(ex \cdot un_n) \cdot dc \cdot L\,(\mathit{leaf}\,f)) \\
= \quad & \{\text{since } \mathit{label} \cdot ex \cdot L\,(\mathit{leaf}\,f) = ex \cdot L f \text{ and} \\
& \qquad \mathit{label} \cdot \mathit{node}\,g = g \cdot F\,\mathit{label}\} \\
& (sg \to ex \cdot L f, g \cdot F\,(\mathit{label} \cdot ex \cdot un_n) \cdot dc \cdot L\,(\mathit{leaf}\,f)) \\
= \quad & \{\text{naturality of } ex \text{ and } dc\} \\
& (sg \to f \cdot ex, g \cdot F\,(\mathit{label} \cdot ex \cdot un_n \cdot L\,(\mathit{leaf}\,f)) \cdot dc) \\
= \quad & \{\text{definition of } bu_n\} \\
& (sg \to f \cdot ex, g \cdot F\,bu_n \cdot dc) \\
= \quad & \{\text{induction and definition of } td_{n+1}\} \\
& td_{n+1}
\end{aligned}$$

The identities

$$\begin{aligned} \mathit{label} \cdot ex \cdot L\,(\mathit{leaf}\,f) &= ex \cdot L f \\ \mathit{label} \cdot \mathit{node}\,g &= g \cdot F\,\mathit{label} \end{aligned}$$

are easy to prove from the definitions of *leaf* and *node*. The claim is that

$$ex \cdot un_n \cdot L\,(\mathit{node}\,g) \cdot cd = \mathit{node}\,g \cdot F\,(ex \cdot un_n) \cdot dc \tag{1}$$

The proof of (1) is again by induction. The base case follows from the assumption that $F\,0 = 0$. For the induction step we argue:

$$\begin{aligned}
& ex \cdot un_{n+1} \cdot L(\mathit{node}\,g) \cdot cd \\
= \quad & \{\text{definition of } un_{n+1} \text{ and conditionals}\} \\
& (sg \to ex, ex \cdot un_n \cdot L\,(\mathit{node}\,g) \cdot cd) \cdot L\,(\mathit{node}\,g) \cdot cd \\
= \quad & \{\text{induction}\} \\
& (sg \to ex, \mathit{node}\,g \cdot F\,(ex \cdot un_n) \cdot dc) \cdot L\,(\mathit{node}\,g) \cdot cd
\end{aligned}$$

$$
\begin{array}{ll}
= & \{\text{conditionals}\} \\
& (sg \cdot L\,(node\;g) \cdot cd \rightarrow ex \cdot L\,(node\;g) \cdot cd, \\
& \qquad\qquad\qquad\qquad node\;g \cdot F\,(ex \cdot un_n) \cdot dc \cdot L\,(node\;g) \cdot cd) \\
= & \{\text{naturality of } sg,\ ex \text{ and } dc\} \\
& (sg \cdot cd \rightarrow node\;g \cdot ex \cdot cd, \\
& \qquad\quad node\;g \cdot F\,(ex \cdot un_n \cdot L\,(node\;g)) \cdot dc \cdot cd) \\
= & \{\text{the three assumptions of Theorem 1}\} \\
& (sg \cdot zip(F,L) \cdot dc \rightarrow node\;g \cdot F\;ex \cdot dc, \\
& \qquad\qquad\qquad\qquad node\;g \cdot F\,(ex \cdot un_n \cdot L\,(node\;g) \cdot cd) \cdot dc) \\
= & \{\text{conditionals}\} \\
& node\;g \cdot F\;ex \cdot (sg \cdot zip(F,L) \rightarrow id, F\,(un_n \cdot L\,(node\;g) \cdot cd)) \cdot dc \\
= & \{\text{claim: see (2) below}\} \\
& node\;g \cdot F\;ex \cdot F\,(sg \rightarrow id, un_n \cdot L\,(node\;g) \cdot cd) \cdot dc \\
= & \{\text{definition of } un_{n+1}\} \\
& node\;g \cdot F\,(ex \cdot un_{n+1}) \cdot dc
\end{array}
$$

The final claim is that for any partial functions $h, k :: L\,A \rightarrow B$

$$(sg \cdot zip(F,L) \rightarrow F\,h, F\,k) = F\,(sg \rightarrow h,k) \cdot zip(F,L) \triangleright \tag{2}$$

Intuitively, (2) holds because *sg* is polymorphic and so depends only on the shape of its argument L-structure. Thus $sg \cdot zip(F,L)$ applied to an F-structure of L-structures all of the same shape returns *True* if *sg* returns *True* on all the elements of F, and *False* if *sg* returns *False* on all the elements. However, the formal proof of (2) seems surprisingly difficult and depends on a number of additional concepts related to zips that we don't have the space to go into. So we will omit the proof.

7 Non-layered Decompositions

In practice, most dynamic programming problems do not have layered decompositions. For example, consider an optimal bracketing problem such as chain matrix multiplication (see Chapter 16 of [5]). The decomposition function *dc* for this problem has type $dc :: L\,a \rightarrow L\,(P\,(L\,a))$ where L is the data type of nonempty lists. For example,

$$dc\ \text{“abcd”} = [(\text{“a”}, \text{“bcd”}), (\text{“ab”}, \text{“cd”}), (\text{“abc”}, \text{“d”})]$$

We can define *dc* by

$$dc = zip(P,L) \cdot cross\,(inits, tails) \cdot fork\,(init, tail)$$

where *inits* and *tails* return the nonempty initial and tail segments of a list respectively, and *cross* is defined by

$$
\begin{array}{ll}
cross & :: (a \rightarrow c, b \rightarrow d) \rightarrow (a,b) \rightarrow (c,d) \\
cross\,(f,g)\,(x,y) & = (f\,x, g\,y)
\end{array}
$$

Although dc is neither layered nor symmetric, the function $fork\,(init, tail)$ has both these properties and we can base a tabulation scheme on this function instead.

More generally, suppose $dc :: L\,a \to F\,(L\,a)$ satisfies the conditions of Corollary 1, and $dcg :: L\,a \to G\,(L\,a)$ is defined by $dcg = extendL \cdot dc$, where $extendL :: F\,(L\,a) \to G\,(L\,a)$. For example, for optimal bracketing we have $G\,a = L\,(P\,a)$ and

$$extendL = zip(P, L) \cdot cross\,(inits, tails)$$

The generalised top-down algorithm gtd reads:

$$\begin{aligned} &gtd && :: (a \to b) \to (G\,b \to b) \to L\,a \to b \\ &gtd\,f\ g && = (sg \to f \cdot ex, g \cdot G\,(gtd\,f\ g) \cdot extendL \cdot dc) \end{aligned}$$

The generalised bottom-up tabulation algorithm gbu is defined by

$$\begin{aligned} &gbu && :: (a \to b) \to (G\,b \to b) \to L\,a \to b \\ &gbu\,f\ g && = label \cdot ex \cdot until\,sg\,(L\,(node\ g) \cdot cd) \cdot L\,(leaf\,f) \end{aligned}$$

where the definition of $node$ is changed to read

$$\begin{aligned} &node && :: (G\,b \to b) \to F\,(N\,b) \to N\,b \\ &node\ g && = uncurry\,Node \cdot fork\,(g \cdot G\,label \cdot extendN, id) \end{aligned}$$

The new function $extendN$ has type $extendN :: F\,(N\,b) \to G\,(N\,b)$. To describe the necessary relationship between $extendL$ and $extendN$ we need a function with type $L\,a \to N\,b$. The following function does the job:

$$\begin{aligned} &nexus && :: (a \to b) \to (G\,b \to b) \to L\,a \to N\,b \\ &nexus\,f\ g && = (sg \to leaf\,f \cdot ex, node\ g \cdot F\,(nexus\,f\ g) \cdot dc) \end{aligned}$$

Equivalently, $nexus\,f\ g = td\,(leaf\,f)\,(node\ g)$, where td is as defined in Section 2. The function $nexus\,f\ g$ builds a nexus except that subtrees are not shared.

Theorem 2. *Suppose cd and dc satisfy the conditions of Theorem 1. Then* $gtd\,f\ g = gbu\,f\ g$ *provided*

$$extendN \cdot F\,(nexus\,f\ g) = G\,(nexus\,f\ g) \cdot extendL$$

Proof. First observe that (1) of Section 6 involved no properties of $node\,g$ so it remains valid. Using (1) we can prove that

$$gbu\,f\ g = label \cdot nexus\,f\ g \tag{3}$$

The proof of (3) is by induction. The induction step reads:

$$\begin{aligned} & gbu_{n+1}\,f\ g \\ = \quad & \{\text{definition of } gbu\} \end{aligned}$$

$$
\begin{array}{ll}
 & \mathit{label} \cdot \mathit{ex} \cdot (\mathit{sg} \rightarrow \mathit{id}, \mathit{un}_n \cdot \mathit{step}\ g) \cdot L\,(\mathit{leaf}\ f) \\
= & \quad \{\text{conditionals and } \mathit{sg} \cdot L\,(\mathit{leaf}\ f) = \mathit{sg}\} \\
 & \mathit{label} \cdot (\mathit{sg} \rightarrow \mathit{ex} \cdot L\,(\mathit{leaf}\ f), \mathit{ex} \cdot \mathit{un}_n \cdot \mathit{step}\ g \cdot L\,(\mathit{leaf}\ f)) \\
= & \quad \{(1)\} \\
 & \mathit{label} \cdot (\mathit{sg} \rightarrow \mathit{ex} \cdot L\,(\mathit{leaf}\ f), \mathit{node}\ g \cdot F\,(\mathit{ex} \cdot \mathit{un}_n) \cdot \mathit{dc} \cdot L\,(\mathit{leaf}\ f)) \\
= & \quad \{\text{naturality of } \mathit{ex} \text{ and } \mathit{dc}\} \\
 & \mathit{label} \cdot (\mathit{sg} \rightarrow \mathit{leaf}\ f \cdot \mathit{ex}, \mathit{node}\ g \cdot F\,(\mathit{ex} \cdot \mathit{un}_n \cdot L\,(\mathit{leaf}\ f)) \cdot \mathit{dc}) \\
= & \quad \{\text{induction}\} \\
 & \mathit{label} \cdot (\mathit{sg} \rightarrow \mathit{leaf}\ f \cdot \mathit{ex}, \mathit{node}\ g \cdot F\,(\mathit{nexus}_n\, f\ g) \cdot \mathit{dc}) \\
= & \quad \{\text{definition of } \mathit{nexus}\} \\
 & \mathit{label} \cdot \mathit{nexus}_{n+1}\, f\ g
\end{array}
$$

Now to complete the proof that $\mathit{gtd}\, f\ g = \mathit{gbu}\, f\ g$ we have to show that

$$
\mathit{gtd}\, f\ g = \mathit{label} \cdot \mathit{nexus}\, f\ g \tag{4}
$$

The proof of (4) is again by induction. The induction step is

$$
\begin{array}{ll}
 & \mathit{gtd}_{n+1}\, f\ g \\
= & \quad \{\text{definition}\} \\
 & (\mathit{sg} \rightarrow f \cdot \mathit{ex}, g \cdot G\,(\mathit{gtd}_n\, f\ g) \cdot \mathit{extendL} \cdot \mathit{dc}) \\
= & \quad \{(3)\} \\
 & (\mathit{sg} \rightarrow f \cdot \mathit{ex}, g \cdot G\,(\mathit{label} \cdot \mathit{nexus}_n\, f\ g) \cdot \mathit{extendL} \cdot \mathit{dc}) \\
= & \quad \{\text{assumption}\} \\
 & (\mathit{sg} \rightarrow f \cdot \mathit{ex}, g \cdot G\, \mathit{label} \cdot \mathit{extendN} \cdot F\,(\mathit{nexus}_n\, f\ g) \cdot \mathit{dc}) \\
= & \quad \{\text{since } g \cdot G\, \mathit{label} \cdot \mathit{extendN} = \mathit{label} \cdot \mathit{node}\ g\} \\
 & (\mathit{sg} \rightarrow f \cdot \mathit{ex}, \mathit{label} \cdot \mathit{node}\ g \cdot F\,(\mathit{nexus}_n\, f\ g) \cdot \mathit{dc}) \\
= & \quad \{\text{since } f \cdot \mathit{ex} = \mathit{label} \cdot \mathit{leaf}\ f \cdot \mathit{ex}\} \\
 & (\mathit{sg} \rightarrow \mathit{label} \cdot \mathit{leaf}\ f \cdot \mathit{ex}, \mathit{label} \cdot \mathit{node}\ g \cdot F\,(\mathit{nexus}_n\, f\ g) \cdot \mathit{dc}) \\
= & \quad \{\text{conditionals and induction}\} \\
 & \mathit{label} \cdot \mathit{nexus}_{n+1}\, f\ g
\end{array}
$$

For example, consider the optimal bracketing problem again, in which $F = P$ and $G\ a = L\,(P\ a)$. Suppose we define $\mathit{extendN}$ by

$$
\begin{array}{l}
\mathit{extendN} :: P\,(N\ a) \rightarrow L\,(P\,(N\ a)) \\
\mathit{extendN} = \mathit{zip}(P, L) \cdot \mathit{cross}\,(\mathit{lspine}, \mathit{rspine})
\end{array}
$$

where

$$
\begin{array}{ll}
\mathit{lspine} & :: N\ a \rightarrow L\,(N\ a) \\
\mathit{lspine}\,(\mathit{Leaf}\ x) & = [\mathit{Leaf}\ x] \\
\mathit{lspine}\,(\mathit{Node}\,(x, (\ell, r))) & = \mathit{lspine}\,\ell \mathbin{+\!\!+} [\mathit{Node}\,(x, (\ell, r))]
\end{array}
$$

and

$$\begin{array}{ll} \mathit{rspine} & :: N\ a \to L\,(N\ a) \\ \mathit{rspine}\,(\mathit{Leaf}\ x) & = [\mathit{Leaf}\ x] \\ \mathit{rspine}\,(\mathit{Node}\,(x,(\ell,r))) & = [\mathit{Node}\,(x,(\ell,r))] + \!\! + \ \mathit{rspine}\ r \end{array}$$

It is easy to show that

$$\begin{array}{l} \mathit{lspine} \cdot \mathit{nexus}\,f\ g\ = L\,(\mathit{nexus}\,f\ g) \cdot \mathit{inits} \\ \mathit{rspine} \cdot \mathit{nexus}\,f\ g = L\,(\mathit{nexus}\,f\ g) \cdot \mathit{tails} \end{array}$$

Now we can reason

$$\begin{array}{cl} & \mathit{extendN} \cdot P\,(\mathit{nexus}\,f\ g) \\ = & \quad \{\text{definition of } \mathit{extendN}\} \\ & \mathit{zip}(P,L) \cdot \mathit{cross}\,(\mathit{lspine},\mathit{rspine}) \cdot P\,(\mathit{nexus}\,f\ g) \\ = & \quad \{\text{property of } \mathit{cross}\} \\ & \mathit{zip}(P,L) \cdot \mathit{cross}\,(\mathit{lspine} \cdot \mathit{nexus}\,f\ g, \mathit{rspine} \cdot \mathit{nexus}\,f\ g) \\ = & \quad \{\text{above}\} \\ & \mathit{zip}(P,L) \cdot \mathit{cross}\,(L\,(\mathit{nexus}\,f\ g) \cdot \mathit{inits}, L\,(\mathit{nexus}\,f\ g) \cdot \mathit{tails}) \\ = & \quad \{\text{property of } \mathit{cross}\} \\ & \mathit{zip}(P,L) \cdot PL\,(\mathit{nexus}\,f\ g) \cdot \mathit{cross}\,(\mathit{inits},\mathit{tails}) \\ = & \quad \{\text{naturality of } \mathit{zip}(P,L)\} \\ & LP\,(\mathit{nexus}\,f\ g) \cdot \mathit{zip}(L,P) \cdot \mathit{cross}\,(\mathit{inits},\mathit{tails}) \\ = & \quad \{\text{definition of } \mathit{extendL}\} \\ & LP\,(\mathit{nexus}\,f\ g) \cdot \mathit{extendL} \end{array}$$

Hence Theorem 2 is applicable.

8 Non-symmetric Decompositions

For some problems the decomposition function is layered but not symmetric. For instance, consider the function *dc* that returns the immediate subsequences of a list. For example,

$$\mathit{dc}\ \text{“abcde”} = [\text{“abcd”}, \text{“abce”}, \text{“abde”}, \text{“acde”}, \text{“bcde”}]$$

Applied to a list of length n the function *dc* returns n lists each of length $n-1$ obtained by dropping a single element. The nexus associated with *dc* is essentially a Boolean lattice, see Figure 2. The nexus is layered but the connectivity varies from level to level. One way of constructing a Boolean lattice was given in [3], but the construction was not shown formally to meet the requirements of the associated tabulation scheme. Instead, it is possible to justify an alternative construction by appeal to Theorem 1 and we will just sketch the details.

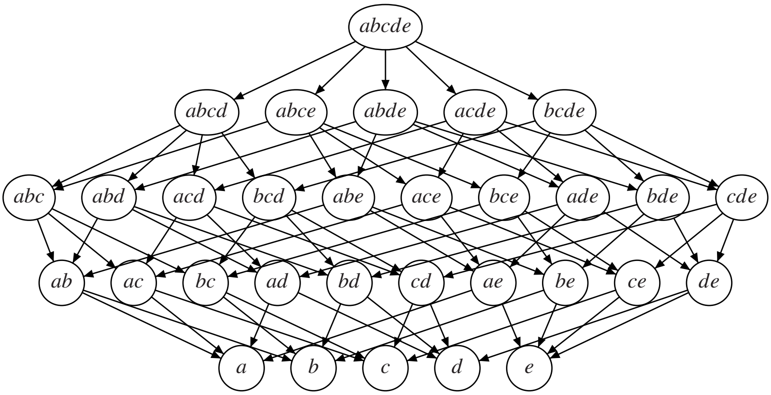


Fig. 2. A Boolean lattice

Firstly, we have $dc :: L\,a \rightarrow F\,(L\,a)$, where $F = L$ and L is the data type of nonempty lists. One definition of dc is

$$\begin{aligned} &dc\,[x, y] &&= [[x], [y]] \\ &dc\,(x : xs) &&= [[x] +\!\!+\, ys \mid ys \leftarrow dc\,xs] +\!\!+\, [xs] \end{aligned}$$

An equivalent definition, though not legal Haskell, is

$$\begin{aligned} &dc\,[x, y] &&= [[x], [y]] \\ &dc\,(xs +\!\!+\, [x]) &&= [xs] +\!\!+\, [ys +\!\!+\, [x] \mid ys \leftarrow dc\,xs] \end{aligned}$$

To satisfy Theorem 1 we have to invent an appropriate definition of cd meeting the three conditions. The essential trick is to group the elements at each level in a particular way in order to prepare for the next level. For example,

$$\begin{array}{l} a \quad b \quad c \quad d \quad e \\ ab \quad (ac\;bc) \quad (ad\;bd\;cd) \quad (ae\;be\;ce\;de) \\ abc \quad (abd\,(acd\;bcd)) \quad ((abe\,(ace\;bce))\,(ade\;bde\;cde)) \\ abcd \quad (abce\,(abde\,(acde\;bcde))) \\ abcde \end{array}$$

The first level is a list of singletons, the second level consists of groups containing, in order, 1, 2, 3 and 4 members, the third level has groups of 1, 3 and 6 elements, and the fourth level has groups of 1 and 4 elements. These numbers are the diagonals of Pascal's triangle. An element ab represents the pair $[a, b]$ and abc represents the triple $[ab, ac, bc]$, and so on. The group structure can be captured by representing lists as binary trees satisfying a shape constraint, essentially that of binomial trees. For example, the middle two lines above are pictured as binomial trees in Figure 3. Reading upwards, the left spine of the first tree has subtrees of sizes 1, 2, 3 and 4, and the left spine of the second tree has subtrees

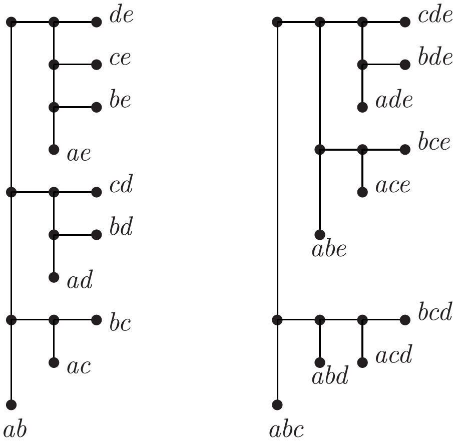


Fig. 3. Two binomial trees

of sizes 1, 3 and 6, in which the second subtree has subtrees of sizes 1, 2 and 3, and the third has subtree of sizes 1 and 2.

Given the data type declaration

$$\mathbf{data}\ B\ a = \mathit{Tip}\ a \mid \mathit{Bin}\,(B\ a)\,(B\ a)$$

the conversion function $\mathit{cvtLB} :: L\,a \rightarrow B\ a$ for converting a list into a binary tree is defined by

$$\mathit{cvtLB} = \mathit{foldl}1\ \mathit{Bin} \cdot \mathit{map}\ \mathit{Tip}$$

This function builds a binary tree all of whose right subtrees are tips, representing the first level of the nexus. With this binary tree representation of lists the function sg translates into a test for whether its argument is a tip, and ex extracts the tip value. The formal definition of cd is now given by

$$\begin{array}{ll}
\mathit{cd} & :: B\ a \rightarrow B\,(L\,a) \\
\mathit{cd}\,(\mathit{Bin}\,(\mathit{Tip}\ a)\,(\mathit{Tip}\ b)) & = \mathit{Tip}\,[a, b] \\
\mathit{cd}\,(\mathit{Bin}\ u\,(\mathit{Tip}\ b)) & = \mathit{Bin}\,(\mathit{cd}\ u)\,(B\,(:[b])\ u) \\
\mathit{cd}\,(\mathit{Bin}\,(\mathit{Tip}\ a)\ v) & = \mathit{Tip}\,(a : \mathit{as}) \quad \mathbf{where}\ \mathit{Tip}\ \mathit{as} = \mathit{cd}\ v \\
\mathit{cd}\,(\mathit{Bin}\ u\ v) & = \mathit{Bin}\,(\mathit{cd}\ u)\,(\mathit{zipBWith}\,(:)\ u\,(\mathit{cd}\ v))
\end{array}$$

The function $\mathit{zipBWith} :: (a \rightarrow b \rightarrow c) \rightarrow B\ a \rightarrow B\ b \rightarrow B\ c$ is analogous to the function $\mathit{zipWith}$ on lists. For example, applying cd to the first tree of figure 3 yields the second tree.

We now claim that $\mathit{td}\,f\ g = \mathit{bu}\,f\ g \cdot \mathit{cvtLB}$. In order to justify the claim we have to invent a definition of $\mathit{dc}' :: B\ a \rightarrow L\,(B\ a)$ with two properties: firstly, $\mathit{td}\,f\ g = \mathit{td}'\,f\ g \cdot \mathit{cvtLB}$ where td' is the same as td except that dc is replaced by

dc'; and, secondly, dc' and cd satisfy the conditions of Theorem 1. The necessary definition of dc' turns out to be

$$\begin{array}{ll}
dc' & ::\ B\ a \rightarrow L\,(B\ a) \\
dc'\,(Bin\,(Tip\ a)\,(Tip\ b)) & = [Tip\ a,\ Tip\ b] \\
dc'\,(Bin\ u\,(Tip\ b)) & = [u] \mathbin{+\!\!+} [Bin\ v\,(Tip\ b) \mid v \leftarrow dc'\ u] \\
dc'\,(Bin\,(Tip\ a)\ v) & = Tip\ a : dc'\ v \\
dc'\,(Bin\ u\ v) & = [u] \mathbin{+\!\!+} zipWith\ Bin\,(dc'\ u)\,(dc'\ v)
\end{array}$$

The first two clauses are obvious translations of the second list-based definition of dc given above. More precisely, we have

$$dc' \cdot cvtLB = F\ cvtLB \cdot dc$$

This equation is sufficient to prove $td\,f\ g = td'\,f\ g \cdot cvtLB$. The second two clauses deal with binary trees of higher rank and are needed for the three conditions of Theorem 1. But the proofs are long and involved, so they are omitted.

9 Summary

Let us recap. Theorem 1 captures the essential relationship between top-down and bottom-up computations for layered decomposition functions. In the restricted case of a symmetric decomposition function we can appeal to Corollary 1. When the decomposition function is not layered, but can be viewed as an extension of one that is, we can appeal to Theorem 2. Finally, in order to apply Theorem 1 some invention is required, both to find an alternative data type for representing L and the appropriate decomposition and composition functions. It remains future work to see whether the invention can be placed on a more systematic footing.

Acknowledgements

A special debt of gratitude is owed to Roland Backhouse for joint collaboration on the (omitted) proof of (2), which will be recorded elsewhere. Thanks are also due to Ralf Hinze and Shin-Chen Mu for many discussions on the subject of tabulations way back in 2003 when the ideas were first being formulated. Ralf Hinze also very kindly gave of his time to draw Figures 1 and 2 using Functional MetaPost. Finally, acknowledgement is owed to all the referees for their constructive remarks, and to one in particular for suggesting that it would be interesting to see whether the combinatorial function *comb* could be treated within the framework, and my pleasure in discovering that it could.

References

1. Backhouse, R.C., Doornbos, H., Hoogendijk, P.: A Class of Commuting Relators. In: STOP workshop, Ameland, The Netherlands (September 1992), `http://www.cs.nott.ac.uk/~rcb/MPC/papers/zips.ps.gz`

2. Backhouse, R.C., Hoogendijk, P.: Generic Properties of Datatypes. In: Backhouse, R., Gibbons, J. (eds.) Generic Programming. LNCS, vol. 2793, pp. 97–132. Springer, Heidelberg (2003)
3. Bird, R.S., Hinze, R.: Trouble shared is trouble halved. In: ACM SIGPLAN Haskell Workshop, Uppsala, Sweden, pp. 1–6 (August 2003)
4. Bird, R.S., de Moor, O.: The Algebra of Programming. Prentice Hall International Series in Computer Science (1997)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Algorithms. MIT Press, Cambridge Mass (1997)
6. Hoogendijk, P.: A Generic Theory of Data Types Ph.D Thesis, Eindhoven Technical University (1997)
7. Hoogendijk, P., Backhouse, R.C.: When do datatypes commute? In: Moggi, E., Rosolini, G. (eds.) CTCS 1997. LNCS, vol. 1290, pp. 242–260. Springer, Heidelberg (1997)
8. Steffen, P., Giegerich, R.: Table design in dynamic programming. Information and Computation 204(9) (September 2006)

Unfolding Abstract Datatypes

Jeremy Gibbons

Computing Laboratory, University of Oxford
http://www.comlab.ox.ac.uk/jeremy.gibbons/

Abstract. We argue that *abstract datatypes* — with public interfaces hiding private implementations — represent a form of *codata* rather than ordinary data, and hence that proof methods for *corecursive* programs are the appropriate techniques to use for reasoning with them. In particular, we show that the universal properties of unfold operators are perfectly suited for the task. We illustrate with the solution to a problem in the recent literature.

1 Introduction

Dijkstra [10] argued that the single most important contribution computing science has made to the world is the emphasis on designing abstractions in order to manage complexity. *Abstract datatypes* [30] — with public interfaces hiding private implementations — have a pivotal role to play in that contribution. Nevertheless, the use of abstract datatypes is not as common among functional programmers (particularly those using languages like Haskell, which does not have first-class modules) as one might expect from history's lesson. One reason for this phenomenon might be the seductive attractions of pattern matching over algebraic datatypes [55], which seem to rely on making visible the representation of data and hence breaking the encapsulation; we return to this point in Section 5. But another reason for the underuse of abstract datatypes, we feel, is that they are not subject to the familiar proof methods of equational reasoning and induction to which functional programming so readily lends itself [2].

The essential reason why standard proof techniques are inapplicable to abstract datatypes is that they are a form of *codata* rather than a form of *data*, with an emphasis on observation rather than construction, process rather than value, and the indefinite rather than the finite. In this paper, we argue that the appropriate proof methods for reasoning about abstract datatypes are those associated with *corecursive programs* [15]. In particular, building on established work on final coalgebra semantics for object-oriented programs, we show that the universal properties of unfold operators [16] fit the bill very nicely.

Our use here of unfold operators, and hence of possibly-infinite data structures, pushes us towards lazy rather than eager functional programming. That works out nicely, because it is the lazy functional programmer who tends to place greater emphasis on equational reasoning. On the other hand, aficionados of ML at least have a powerful module facility at their disposal, and so might be expected to make greater use of abstract datatypes than do adherents of Haskell.

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 110–133, 2008.

We illustrate our case with the solution to a problem of reasoning with abstract datatypes from the recent literature, concerning the elimination of redundant conversions to and from lists in a stream-based reimplementation of the Haskell standard list library [9].

The remainder of this paper is structured as follows. Section 2 explains the modelling of abstract datatypes using existential type quantification, and Section 3 discusses the corecursive proof methods appropriate for reasoning about such constructions. Section 4 presents the case study on Coutts *et al.*'s stream fusion. Section 5 concludes and discusses related work.

2 Abstract Types Have Existential Type

Abstract datatypes are "a kind of data abstraction where a type's internal form is hidden behind a set of access functions; values of the type are created and inspected only by calls to the access functions" [22]. Hiding of the internal form is achieved by existential quantification over the representation type: an abstract data structure consists of operations operating on an internal state, whose type is hidden from everything except those operations. Mitchell and Plotkin [32] expressed this view in the slogan "abstract types have existential type".

2.1 An Example: Complex Numbers

For example, consider (a simplification of) the Haskell datatype *Complex*:

$$\textbf{data}\ \mathit{Complex} = \mathit{MkComplex}\ \mathit{Double}\ \mathit{Double}$$

This introduces a constructor $\mathit{MkComplex} :: \mathit{Double} \rightarrow \mathit{Double} \rightarrow \mathit{Complex}$. The outcome is not an abstract datatype, because the representation as a pair of *Double*s (as Cartesian coordinates, as it happens) is visible. Instead, it is a *concrete datatype*. This provides other advantages — such as pattern matching — but loses the benefit of information hiding. For example, to determine whether a complex number is real, we can use the following function:

$$\begin{array}{l}\mathit{isReal} :: \mathit{Complex} \rightarrow \mathit{Bool}\\ \mathit{isReal}\ (\mathit{MkComplex}\ x\ y) = (y == 0.0)\end{array}$$

But if we decide to change the representation to polar coordinates, all such definitions will need modification.

An abstract datatype of complex numbers should hide the representation, as follows [43].

$$\begin{array}{lll}\textbf{data}\ \mathit{Complex} = \exists s.\ C & (s \rightarrow (\mathit{Double}, \mathit{Double}) \rightarrow s) & \text{-- create}\\ & (s \rightarrow \mathit{Complex} \rightarrow s) & \text{-- add}\\ & (s \rightarrow \mathit{Double}) & \text{-- real}\\ & (s \rightarrow \mathit{Double}) & \text{-- imaginary}\\ & s & \text{-- self}\end{array}$$

As above, this introduces a new constructor C; this takes four functions and an internal representation or 'self', and yields a *Complex*. Here, the self is of type s, for some s; the four functions each take an argument of type s. Note that the type variable s does not appear on the left-hand side of the datatype declaration, so it has to be quantified somehow. A universal quantification would be inappropriate, because the representation is of *some* type, not *any* type; existential quantification is what is required. (In common extensions to Standard Haskell supporting existential quantification, somewhat perversely, it is written with the keyword `forall` [39] — the justification being that a datatype declaration such as

$$\mathbf{data}\ D = \exists s.\ \mathit{MkD}\ (s, s \rightarrow \mathit{Integer})$$

introduces a constructor $\mathit{MkD} :: (\exists s.\ (s, s \rightarrow \mathit{Integer})) \rightarrow D$, and this type is isomorphic to $\forall s.\ ((s, s \rightarrow \mathit{Integer}) \rightarrow D)$ because D is independent of s — but in this paper we will pretty-print that keyword as '∃'.)

Packaged up with the internal representation of a complex number are four functions: for creating a new complex number, adding on a second complex number (obtaining an updated representation), and extracting the real and imaginary components. These can be given more user-friendly names:

$$\begin{array}{l}
\mathit{new} :: \mathit{Complex} \rightarrow \mathit{Double} \rightarrow \mathit{Double} \rightarrow \mathit{Complex} \\
\mathit{new}\ (C\ n\ a\ r\ i\ s)\ x\ y = C\ n\ a\ r\ i\ (n\ s\ (x, y)) \\
\mathit{add} :: \mathit{Complex} \rightarrow \mathit{Complex} \rightarrow \mathit{Complex} \\
\mathit{add}\ (C\ n\ a\ r\ i\ s)\ c \quad = C\ n\ a\ r\ i\ (a\ s\ c) \\
\mathit{rea}, \mathit{ima} :: \mathit{Complex} \rightarrow \mathit{Double} \\
\mathit{rea}\ \ (C\ n\ a\ r\ i\ s) \qquad = r\ s \\
\mathit{ima}\ (C\ n\ a\ r\ i\ s) \qquad = i\ s
\end{array}$$

Crucially, nothing other than these four functions can access the internal representation; that is guaranteed by the quantification over the type variable s. In particular, there is no way to extract the representation itself. So of course, the set of operations made available has to be considered carefully; in contrast to concrete datatypes, which support pattern matching and hence easy extension with new functions, adding a new operation to an abstract datatype inexpressible in terms of existing functions requires a change to the datatype definition [8].

The abstract datatype specifies a signature, but not an implementation. Here is one implementation, in the expected Cartesian coordinates:

$$\begin{array}{l}
\mathit{zeroC} :: \mathit{Complex} \\
\mathit{zeroC} = C\ (\lambda(x, y) \rightarrow \lambda z \rightarrow z) \\
\qquad\qquad\ (\lambda(x, y) \rightarrow \lambda c \rightarrow (x + \mathit{rea}\ c, y + \mathit{ima}\ c)) \\
\qquad\qquad\ (\lambda(x, y) \rightarrow x) \\
\qquad\qquad\ (\lambda(x, y) \rightarrow y) \\
\qquad\qquad\ (0.0, 0.0)
\end{array}$$

Notice the type $s \rightarrow \mathit{Complex} \rightarrow s$ for the addition function, allowing complex numbers of different representations to be added. Consequently, the implementation of *add* has privileged access to the representation of the first argument (through the fields *x* and *y*), but only public access to the second argument (through the operations *rea* and *ima*). This inefficiency is a well-known problem with binary methods in object orientation [4]. Mitchell and Plotkin's approach [32] differs, providing privileged access to the representations of both arguments of *add* but therefore requiring both arguments to have the same representation; their alternative is more efficient, but less flexible. The difference is essentially a matter of whether the scope of the existential quantification is narrowed down to specific objects, or widened out to the whole program.

Note also the rather odd type *Complex* → *Double* → *Double* → *Complex* for *new*, requiring an existing complex number before a new one can be created. Of course, the implementation of an abstract data structure has to come from somewhere; the function *new* creates a new structure using the operations and data representation of an existing structure, simply assigning a new state. This is more analogous to cloning in prototype-based languages [54] than it is to construction *de novo* in more traditional object-oriented programming.

2.2 An Alternative Implementation

Here is an alternative implementation of complex numbers, using polar coordinates.

```
data Polar = P{mag :: Double, phase :: Double}

zeroP :: Complex
zeroP = C (λp → λz → c2p z)
          (λp → λc → let (x, y) = p2c p in c2p (x + rea c, y + ima c))
          (λp → fst (p2c p))
          (λp → snd (p2c p))
          (P{mag = 0.0, phase = 0.0})

p2c p = (mag p × cos (phase p), mag p × sin (phase p))
c2p (x, y) = P{mag = sqrt (x×x + y×y), phase = atan2 y x}
```

(A wiser definition of *c2p* would scale the two coordinates before multiplying, to avoid overflows; but the naive version above is clearer.) Note that although *zeroC* and *zeroP* have different representations, the existential quantification allows them to be of the same type.

Now, the reality check for complex numbers becomes:

```
isReal :: Complex → Bool
isReal z = (ima z == 0)
```

We can no longer use pattern matching on the representation; but this definition works just as well for a polar — or indeed, any other — representation as for Cartesian.

2.3 An Explicit Signature for Complex Numbers

The type declaration *Complex* is rather complicated, on account of the number of operations provided. Moreover, those operations all have a common domain, the hidden representation type; so they can be coalesced into one function returning a tuple. We will adopt a convention of separating out the description of the signature (that is, the number and types of the operations) from the existential quantification, writing the following mutually recursive definitions instead.

```
data ComplexF s = CF{_new :: (Double, Double) → s,
                     _add :: Complex → s,
                     _rea  :: Double,
                     _ima :: Double}
data Complex = ∃s. C (s → ComplexF s) s
```

Here is the Cartesian implementation of zero:

```
zeroC :: Complex
zeroC = C fc (0.0, 0.0) where
  fc :: (Double, Double) → ComplexF (Double, Double)
  fc = (λ(x, y) → CF{_new = λz → z,
                     _add = λc  → (x + rea c, y + ima c),
                     _rea = x,
                     _ima = y})
```

Note that the function *fc* is analogous to a class in object-oriented terms: it determines the data representation, and provides implementations of the methods on that representation. We can provide user-friendly wrappers *rea*, *ima*, *add*, *new* as before. Similarly for the polar implementation *zeroP*:

```
zeroP :: Complex
zeroP = C fp (P{mag = 0.0, phase = 0.0}) where
  fp :: Polar → ComplexF Polar
  fp = (λp → CF{_new = λz → c2p z,
                _add = λc  → let (x, y) = p2c p in
                               c2p (x + rea c, y + ima c),
                _rea = fst (p2c p),
                _ima = snd (p2c p)})
```

2.4 Abstract Datatype Genericity

Of course, there is nothing special about the particular datatype of complex numbers; the approach generalises very nicely. This leads to *datatype-generic abstract datatypes*, parametrised by the signature [14]. The signature should be a *strictly positive functor*: it should be functorial in the state type (in order to allow the definition of the unfold for the final coalgebra semantics), and no occurrences of the type parameter may appear to the left of an arrow (to maintain the encapsulation of the hidden state). The abstract datatype itself packages up some hidden state with the operations, of type specified by the signature.

$$\mathbf{data}\ \mathit{Functor}\ f \Rightarrow \mathit{ADT}\ f = \exists s.\ D\ (s \rightarrow f\ s)\ s$$

(The Haskell type class context '*Functor f*' entails an operation *fmap* of type $(a \rightarrow b) \rightarrow (f\ a \rightarrow f\ b)$.) Instantiating the signature parameter to *ComplexF* yields complex numbers:

$$\begin{aligned}
&\mathbf{type}\ \mathit{Complex} = \mathit{ADT}\ \mathit{ComplexF}\\
&\mathit{zeroCG}, \mathit{zeroPG} :: \mathit{Complex}\\
&\mathit{zeroCG} = D\ \mathit{fc}\ (0.0, 0.0)\\
&\mathit{zeroPG} = D\ \mathit{fp}\ (P\{\mathit{mag} = 0.0, \mathit{phase} = 0.0\})
\end{aligned}$$

(Note that *ComplexF* and *Complex* are mutually recursive, so this redefinition of *Complex* entails also a redefinition of *ComplexF*.)

3 Data and Codata

Abstract datatypes are inhabited by *codata*, as opposed to the ordinary data inhabiting the more familiar algebraic datatypes. In general, codata is manipulated through destructors instead of constructors. Kieburtz [27] identifies some fundamental respects in which codata differs from data:

- Codata structures have hidden representations, accessible only via operations specifically provided for this purpose; whereas the representations of data structures are visible, for example through pattern matching.
- Consequently, standard datatype-generic operations such as pretty-printing and comparison, automatically defined or easily derived from the structure of an arbitrary datatype, are generally not applicable to codatatypes.
- Codata is typically infinite, since it may provide operations that yield other instances of the same type, as with the *new* and *add* operations in the complex number example in Section 2 (but records with field extractors are an exception to this rule). Data is usually finite (although with lazy evaluation, infinite recursive algebraic data structures can be constructed; we see an example shortly).

As it happens, codatatypes are generally *greatest fixpoints* of recursive type equations, whereas datatypes are *least fixpoints*. In some settings (such as that of continuous functions between complete partial orders, as embodied in Haskell for example), least and greatest type fixpoints coincide; but in many settings (such as total functions between sets, as used in Cockett's Charity [7] and Turner's Total Functional Programming [52,53]) the two are distinguished.

3.1 Greatest Fixpoint Types as Codata

Given a suitable operation F on types (technically, a covariant functor; but for simplicity, think of some combination of sums and products), the least fixpoint

$\mu(F)$ of F is the smallest type X such that $F\ (X) \approx X$. This corresponds to the algebraic datatype declaration

$$\mathbf{data}\ \mathit{Mu}\ f = \mathit{In}\{\ \mathit{out} :: f\ (\mathit{Mu}\ f)\}$$

when read in terms of total functions between sets (rather than continuous functions between complete partial orders), capturing just the total and finite data structures of a given shape. The constructor $\mathit{In}{::}f\ (\mathit{Mu}\ f) \rightarrow \mathit{Mu}\ f$ and destructor $\mathit{out} :: \mathit{Mu}\ f \rightarrow f\ (\mathit{Mu}\ f)$ are the witnesses to the isomorphism.

There is a well-known technique called *Church encoding* [3,19] for representing such least fixpoint or *initial* recursive datatypes in the polymorphic lambda calculus, without having to introduce new language constructs like **data** and pattern matching. The encoding is as a higher-order functional type:

$$\mu(F) = \forall X.\ (F\ (X) \rightarrow X) \rightarrow X$$

(Technically, *strong initiality*, conferring also a corresponding proof principle, requires additional assumptions, such as parametricity or "theorems for free" in the underlying category [46,57].)

For example, integer lists have shape determined by $L\ (X) = 1 + \mathit{Integer} \times X$, where 1 denotes the unit type with a single element, *Integer* the type of integers, + disjoint union, and × Cartesian product. Integer lists therefore have the Church encoding $\mu(L) = \forall X.\ (L\ (X) \rightarrow X) \rightarrow X$. Note that, using standard type isomorphisms, we have $L\ (X) \rightarrow X \approx X \times (\mathit{Integer} \times X \rightarrow X)$, so an equivalent definition is $\mu(L) = \forall X.\ (X \times (\mathit{Integer} \times X \rightarrow X)) \rightarrow X$. Moreover, similar type isomorphisms yield a type $\forall a.\ [a] \rightarrow \forall b.\ (b, (a, b) \rightarrow b) \rightarrow b$ for the function *foldr* from the Haskell standard library [40]. In other words, when specialised to $a = \mathit{Integer}$, *foldr* computes the Church encoding of a list of integers.

What is rather less well known is that this encoding dualises, allowing the representation also of greatest fixpoint or *final* recursive types [58,62]:

$$\nu(F) = \exists X.\ (X \rightarrow F\ (X)) \times X$$

(Again, parametricity is required in order to deduce *strong finality*, conferring the corresponding proof principle). For example, the type of finite and infinite integer lists, the greatest fixpoint $\nu(L)$ of the functor L, is encoded as $\exists X.\ (X \rightarrow 1 + \mathit{Integer} \times X) \times X$. Moreover, standard type isomorphisms yield a type $\forall a.\ (\exists b.\ (b \rightarrow \mathit{Maybe}\ (a, b), b)) \rightarrow [a]$ for the function *unfoldr* from the Haskell standard library [40]. In other words, when specialised to $a = \mathit{Integer}$, *unfoldr* computes a co-list of integers from its co-Church encoding.

In summary, whereas least fixpoint types correspond to universal type quantifications, greatest fixpoint types correspond to existential quantifications.

3.2 Proof Methods for Codata

So, *zeroCG* and *zeroPG* are both elements of a greatest fixpoint type. We might expect them to be 'equal' in some sense, since they both represent 'the same' complex number. But in what sense could they be equal? They have different representations, so straightforward structural comparisons are inappropriate.

The generally accepted approach to take to equality on codata, such as between instances of abstract datatypes, is *observational equivalence*, or "equality as far as we can see" [24]. Two instances of an abstract datatype are clearly different if there is an experiment — that is, a sequence of operations provided by the signature — yielding distinguishable *concrete* outputs (which might without loss of generality be as primitive as bits, but we take here to include types like *Integer* and *Double*); and conversely, if no such experiment exists, we consider the two instances to be equal.

That informal characterisation of observational equivalence can be formalised in two ways, which turn out to be equivalent: via *bisimulation and coinduction* [21,25,36] or via *universal properties of final coalgebras* [23,26,45,59]. In this context, bisimulation amounts to the same thing as logical relations and relational parametricity [31,44]. Bisimulation and universal properties are compared in a recent survey paper [15]. That survey applies the techniques to proving equality of concrete datatypes, specifically streams, for which structural comparisons are also available. It therefore takes the structural comparison as the definition of equality, and proves that the other notions coincide with it. Since the present paper concentrates on abstract datatypes, structural comparison is unavailable; but these two notions of observational equivalence still agree with each other.

3.3 Final Coalgebras

Here, we take the final coalgebra approach. The single experimental steps available on an abstract data structure of type *ADT f* are captured, with their required inputs and specified outputs, by the signature functor *f*. The tree of all possible experiments is obtained by repeatedly applying these operations.

```
data Tree f = T{ unT :: f (Tree f)}
tree :: Functor f ⇒ ADT f → Tree f
tree (D h s) = unfold h s where
  unfold :: Functor f ⇒ (a → f a) → a → Tree f
  unfold f x = T (fmap (unfold f) (f x))
```

As we noted above, because some experimental steps may yield new abstract data structures, observation trees will typically be infinite; accordingly, *Tree f* is the greatest fixpoint of *f*. Nevertheless, we consider *Tree* to be a type of data rather than of codata: it is amenable to pattern matching and to structural datatype-generic operations such as pretty-printing and comparison. Although in Haskell *Mu f* and *Tree f* coincide semantically, we use different datatypes to reinforce the distinction between least and greatest fixpoints.

Now, the claim that the abstract data structures *zeroCG* and *zeroPG* are *observationally equivalent* reduces to a more amenable statement that the concrete data structures *tree zeroCG* and *tree zeroPG* are *structurally equal*: an experiment distinguishing *zeroCG* and *zeroPG* corresponds to a difference between the two trees, and the absence of such an experiment implies the equality of those trees. The claim is still not effectively decidable, because the trees are both infinitely deep and infinitely wide; but at least it is now open to proof via familiar equational reasoning at the meta-level.

3.4 Proving Equivalence

Observational equivalence of the complex numbers *zeroCG* and *zeroPG* follows from structural equality of their observation trees *tree zeroCG* and *tree zeroPG*, which can be demonstrated using the universal property of *unfold*:

$$h = \mathit{unfold}\,f \iff \mathit{unT} \cdot h = \mathit{fmap}\ h \cdot f$$

We have

$$\mathit{tree\ zeroCG} = \mathit{tree\ zeroPG} \iff \mathit{unfold\,fc}\ (0.0, 0.0) = \mathit{unfold\,fp}\ (P\ 0.0\ 0.0)$$

But it isn't immediately obvious how to apply the universal property here: this is not an equation between two functions of the form *unfold h*, but rather between two trees of the form *unfold h s*. How can we move forward?

Fortunately, this is a somewhat special case, because there is a simulation relationship between the two instances. Specifically, we can abstract from the initial state $(0.0, 0.0)$ of the Cartesian implementation, since $(0.0, 0.0) = \mathit{p2c}\ (P\ 0.0\ 0.0)$, obtaining the proof obligation $\mathit{unfold\,fc} \cdot \mathit{p2c} = \mathit{unfold\,fp}$. This equation can be proved using the fusion law of *unfold*, a simple corollary of the universal property:

$$\mathit{unfold}\,f \cdot g = \mathit{unfold}\,f' \Longleftarrow f \cdot g = \mathit{fmap}\ g \cdot f'$$

All that remains is to establish the premise, $\mathit{fc} \cdot \mathit{p2c} = \mathit{fmap\,p2c} \cdot \mathit{fp}$ — that is, that *p2c* is the abstraction function relating *fc* and *fp*. The only property of complex numbers required in the proof is that $\mathit{p2c} \cdot \mathit{c2p} = \mathit{id}$. The calculation can be found in an appendix (Section A).

We chose here to abstract from the initial state of the Cartesian implementation of the abstract datatype, effectively expressing that implementation in terms of the polar representation. This particular proof of equivalence is doubly special, because the simulation also works the other way around: we could have abstracted the initial state $P\,0.0\,0.0$ of the polar implementation instead. (The only complication in doing so is that $\mathit{c2p} \cdot \mathit{p2c}$ is not quite the identity function; however, it is the identity on the reachable states — those with non-negative magnitude, phase between 0 and 2π, and zero phase if zero magnitude.)

In general, given two implementations of an abstract datatype, neither will simulate the other; instead, each introduces extensions inexpressible by the other. In that case, each can be shown observationally equivalent to a third implementation that can simulate both. We will see an example in Section 4.8.

4 Stream Fusion

Coutts *et al.* [9] present an elegant technique for obtaining better fusion of list functions, by reimplementing the Standard Haskell list library [40] to use internally an abstract datatype (of 'streams') rather than the familiar algebraic datatype of lists; we summarise work in Sections 4.1–4.4 and 4.7 below. However, they don't prove that their reimplementation is sound; we present such a proof in the remainder of this section.

4.1 An Abstract Datatype of Streams

A simplistic version of Coutts *et al.*'s approach uses a curried version of Haskell's *Maybe* datatype on pairs as the signature of the stream abstract datatype:

```
data Maybe2 a b = Nothing2 | Just2 a b
type Stream a = ADT (Maybe2 a)
```

Thus, a stream has two components, qualified by some existentially bound state type s: an internal state of type s, and a body that when applied to such a state yields either a head and a new state, or nothing. (In other words, the greatest-fixpoint or co-Church encoding is being used.) For example, here is one implementation of the string `"abc"`:

```
abc = D h 0 where
  h i = case i of
    0 → Just2 'a' 1
    1 → Just2 'b' 2
    2 → Just2 'c' 3
    3 → Nothing2
```

The approach can be seen as an implementation of the Iterator design pattern from object-oriented programming [13]. The full story of the approach permits a third outcome of the stream body, to deal with nested recursions; we return to this point in Section 4.7 below.

4.2 Stream Operations

The list library is redefined in terms of streams. For example, the map function on lists is reimplemented as follows:

```
mapS f (D h s) = D h′ s where
  h′ s = case h s of Nothing2  → Nothing2
                     Just2 x s′ → Just2 (f x) s′
```

This function is a rather special case, because the internal representation of the stream *mapS f xs* is the same as that of *xs*. In general, internal representations change; for example, the zip function is:

```
zipS :: Stream a → Stream b → Stream (a, b)
zipS (D h s) (D j t) = D k (s, t) where
  k (s, t) = case (h s, j t) of
    (Just2 x s′, Just2 y t′) → Just2 (x, y) (s′, t′)
    _                        → Nothing2
```

The internal representation (s, t) of the result is of a different type to the representations s and t of the two arguments.

4.3 A List Interface

The standard library is reimplemented using streams, but the interface presented to the programmer still uses the familiar algebraic datatype of lists; therefore, conversion functions $\mathit{stream} :: [\,a\,] \to \mathit{Stream}\ a$ and $\mathit{unstream} :: \mathit{Stream}\ a \to [\,a\,]$ are needed.

```
stream :: [a] → Stream a
stream xs = D uncons xs where
  uncons :: [a] → Maybe2 a [a]
  uncons xs = if null xs then Nothing2 else Just2 (head xs) (tail xs)
unstream :: Stream a → [a]
unstream (D h s) = unfoldr h s where
  unfoldr :: (b → Maybe2 a b) → b → [a]
  unfoldr f y = case f y of Nothing2 → [ ]; Just2 x y′ → x : unfoldr f y′
```

For example, the familiar *map* on lists is retrieved by

```
map f = unstream · mapS f · stream
```

(In fact, *unstream* is essentially a specialisation of *tree*.)

4.4 Eliminating Conversions

The crucial point in Coutts *et al.*'s work is the elimination of redundant conversions in the composition of list operations, such as in the composition of two maps:

```
unstream · mapS f · stream · unstream · mapS g · stream
```

Here, the double conversion $\mathit{stream} \cdot \mathit{unstream}$ from streams to lists and back again is redundant. If it could be eliminated, the two occurrences of *mapS* would become adjacent; and because the definition of *mapS* is non-recursive, standard compiler optimisations — specifically, a case-of-case optimisation — can relatively easily fuse their bodies. The actual elimination of $\mathit{stream} \cdot \mathit{unstream}$ itself is easy, using the Glasgow Haskell Compiler's programmer-definable rewrite rules [42].

In fact, $\mathit{stream} \cdot \mathit{unstream}$ is not quite the identity: $\mathit{stream}\ (\mathit{unstream}\ \bot)$ equals $D\ \mathit{uncons}\ \bot$ rather than $\bot$. In practice, this difference does not cause a problem if (a) the *Stream* datatype is not exported from the library, and (b) the library itself does not construct bottom values of type *Stream*; so their implementation is carefully arranged to satisfy these conditions.

Coutts *et al.* do claim (implicitly) that $\mathit{stream}\,(\mathit{unstream}\,(D\ h\ s)) = D\ h\ s$, but provide no proof of this claim; they say that "it is not entirely trivial to define a useful equivalence relation on streams [. . .] due to the fact that a single list can be modeled by infinitely many streams" [9, p320].

4.5 Destroying Streams

In fact, there is a simple proof of the $\mathit{stream} \cdot \mathit{unstream}$ identity: it is an instance of the *destroy*/*unfoldr* rule [48], the dual of the better-known *foldr*/*build* rule [18]. The function *destroy* is defined as follows:

$$\begin{aligned}&\mathit{destroy} :: (\forall b.\ (b \to \mathit{Maybe}_2\ a\ b) \to b \to c) \to [\,a\,] \to c\\&\mathit{destroy}\ g = g\ \mathit{uncons}\end{aligned}$$

so that $\mathit{stream}\ \mathit{xs} = \mathit{destroy}\ D\ \mathit{xs}$. Then the *destroy*/*unfoldr* rule states that

$$\mathit{destroy}\ g\ (\mathit{unfoldr}\ f\ s) = g\ f\ s$$

The proof of this rule is a straightforward application of Reynolds' parametricity [47]: the free theorem [57] of the type of the argument g of *destroy* is

$$\psi \cdot f = \mathit{fmap}\ f \cdot \phi \implies g\ \psi \cdot f = g\ \phi$$

Letting $\psi = \mathit{uncons}$ and $f = \mathit{unfoldr}\ \phi$ gives the *destroy*/*unfoldr* rule.

We note in passing that whereas *foldr*/*build* fusion has turned out to be a little disappointing in its applicability [33], *destroy*/*unfoldr* fusion seems to have a much wider scope. For example, it is straightforward to apply the latter technique to zip-like functions and functions with accumulating parameters [48], avoiding the need for *augment*-like generalisations [17]. This additional promise lends weight to our advocacy for greater appreciation of corecursive programming [16].

It is quite natural, so to speak, that equivalence proofs for abstract datatypes boil down to applications of parametricity. Intuitively, parametricity results capture "the only thing you can do, for type reasons". For example, for the abstract data structure $D\ h\ s$, the type of the representation s is hidden, and so "the only thing you can do, for type reasons", is to apply h to s. Indeed, Reynolds [46] originally called his result the "representation theorem", and motivated it by appeal to independence from a choice of representation: "We expect that the meaning of [. . .] a program will remain unchanged if the [definition of an abstract datatype] is altered by changing the representation of the type and redefining its primitive operations in a consistent manner" [46].

4.6 Unfolding Observations

Another way of looking at Coutts *et al.*'s problem is to remember that streams are an abstract datatype, and so structural equivalence is the wrong tool to use; observational equivalence is what is needed.

In this case, since we have by definition that

$$\mathit{stream}\ (\mathit{unstream}\ (D\ h\ s)) = D\ \mathit{uncons}\ (\mathit{unfoldr}\ h\ s)$$

it suffices to show observational equivalence of $D\ h\ s$ and $D\ \mathit{uncons}\ (\mathit{unfoldr}\ h\ s)$. (Notice that these two streams will generally have different representations. The latter necessarily uses a list for the state, whereas the former may have an arbitrary state type.)

As we argued in Section 3.3, observational equivalence of abstract data structures is just structural equivalence of their unfoldings to the final coalgebra. For datatype *Stream a*, the signature is the functor $\mathit{Maybe}_2\ a$, whose final coalgebra is possibly infinite lists of *a*s, and the operation to build such a list is the familiar but underappreciated *unfoldr* [16]. So we have to prove

$$\mathit{unfoldr}\ \mathit{uncons}\ (\mathit{unfoldr}\ h\ s) = \mathit{unfoldr}\ h\ s$$

This follows easily from the universal property

$$h = \mathit{unfoldr}\ f \iff \mathit{uncons} \cdot h = \mathit{fmap}\ h \cdot f$$

of *unfoldr*. This alternative proof using the universal property of unfold is important: as we shall see in Section 4.8, it seems to generalise better than the parametricity-based proof underlying *destroy*/*unfoldr*.

4.7 Streams That Skip

The simple version above of Coutts *et al.*'s story uses a representation of streams providing a single observation, yielding either no information (for an empty stream) or a head and the state for a tail (for a non-empty stream). In fact, the complete story is more sophisticated, allowing a third outcome: a new state, but no head.

```
data Step a s = Done | Yield a s | Skip s
type SStream a = ADT (Step a)
```

The extra outcome is needed to support operations such as filtering, which do not produce an element at every step — when the filter discards an element from an underlying stream, or that stream skips itself, then the outer stream skips instead of yielding:

```
filterS :: (a → Bool) → SStream a → SStream a
filterS p (D h s) = D (try h p) s where


try h p s = case h s of
   Done     → Done
   Skip s'  → Skip s'
   Yield x s' → if p x then Yield x s' else Skip s'
```

Without the possibility of skipping, the body of the stream would have to be recursive, thereby complicating fusion optimisations.

The conversions from lists is very similar to the simple case:

```
sstream :: [a] → SStream a
sstream xs = D unconsS xs where
  unconsS :: [a] → Step a [a]
  unconsS []       = Done
  unconsS (x : xs) = Yield x xs
```

The conversion back to lists is more involved, because of the need to handle skips:

```
unsstream :: SStream a → [a]
unsstream (D h s) = unfoldr (force h) s where
  force :: (s → Step a s) → (s → Maybe2 a s)
  force h s = case h s of
    Done       → Nothing2
    Yield x s′ → Just2 x s′
    Skip s′    → force h s′
```

Note that the body *force* of the unfold here is recursive, so it would be difficult for standard compiler optimisations to fuse a following function. That is not a big problem, because *unsstream* is intended to be used only when leaving the improved implementation of the list library, when fusion is not expected anyway. Moreover, note that *unsstream* may be unproductive, although for example even *filterS* (*const False*) is always productive: that particular lump in the carpet has been shuffled under the furniture, but no library reimplementation can eliminate it altogether.

4.8 Reasoning with Skips

The presence of skips has interesting consequences for proofs. We should no longer take pure observational equivalence as the appropriate notion of equality on skipping streams, because we ought to treat some observationally distinguishable skipping streams as effectively equivalent. Coutts *et al.* say that "equivalence on streams should be defined modulo *Skip* values [...] semantics should not be affected by the presence or absence of *Skip* values" [9, p320].

For example, consider the stream version of the standard list function *concat*:

```
concatS :: SStream (SStream a) → SStream a
concatS (D hs ss) = D hc (Nothing, ss) where

hc (Nothing, ss) = case hs ss of
  Done        → Done
  Skip ss′    → Skip (Nothing, ss′)
  Yield s ss′ → Skip (Just s, ss′)
```

```
hc (Just (D ha sa), ss) = case ha sa of
   Done        → Skip (Nothing, ss)
   Skip sa′    → Skip (Just (D ha sa′), ss)
   Yield y sa′ → Yield y (Just (D ha sa′), ss)
```

In order to maintain a non-recursive body, this uses a rather complex internal state consisting of an optional *SStream a* and the internal state of a remaining *SStream* (*SStream a*); only if the former is present and yielding does the whole yield. We might expect the following property — one of the monad laws for streams — to hold:

$$concatS \cdot wrapS = id$$

where *wrapS* wraps an element up as a singleton stream:

```
wrapS :: a → SStream a
wrapS x = D fetch (Just x) where
   fetch :: Maybe a → Step a (Maybe a)
   fetch (Just x) = Yield x Nothing
   fetch Nothing = Done
```

The two sides of the property are not even observationally equivalent, because the left-hand side $concatS \cdot wrapS$ introduces quite a few extra *Skip*s.

In fact, the appropriate notion of "equivalence modulo *Skip*s" is obtained precisely by taking structural equality on their unfoldings to lists:

$$unsstream \cdot concatS \cdot wrapS = unsstream \cdot id$$

The proof of this latter property is a fairly straightforward (albeit somewhat tedious) application of the universal property of *unfoldr*; it is relegated to an appendix (Section B).

The alternative proof technique in terms of the universal property of *unfoldr* is important, because the *destroy*/*unfoldr* rule used in Section 4.5 does not seem to generalise nicely to skipping streams. The analogous development would be to introduce a function

```
destroyS :: (∀b. (b → Step a b) → b → c) → [a] → c
destroyS g = g unconsS
```

so that $sstream\ xs = destroyS\ D\ xs$. The free theorem of the type of the argument g of *destroyS* is

$$\psi \cdot f = fmap\ f \cdot \phi \implies g\ \psi \cdot f = g\ \phi$$

but it isn't clear how to instantiate this equation to obtain the desired result; indeed, 'equivalence modulo *Skip*s' feels more ad hoc than parametric.

5 Conclusions

5.1 Related Work

Data abstraction has long been recognised as a crucial tool in managing the complexity of software systems [30,37]. Pattern matching on algebraic datatypes is also widely appreciated as an extremely convenient technique [55]. But as Wadler's proposal [56] noted twenty years ago, it is difficult to marry the two together: data abstraction depends on hiding a data representation that pattern matching relies on revealing. There have been numerous other proposals for combining data abstraction with pattern matching over the years [5,6,34,35,51], and indeed a recent flurry of activity in the area [11,41,49,61].

One could look at *final coalgebra semantics* as a disciplined way of thinking about pattern matching over abstract datatypes. Rather than trying to force these two somewhat conflicting ideas together, one could instead define a *view* of codata (supporting abstraction) as data (supporting pattern matching), using the function *tree* from Section 3.3. In case a full transformation from 'completely codata' to 'completely data' is inappropriate, simply apply the body of the abstract datatype once:

$$\begin{aligned}&\mathit{unpack} :: \mathit{Functor}\, f \Rightarrow \mathit{ADT}\; f \rightarrow f\; (\mathit{ADT}\; f)\\&\mathit{unpack}\; (D\; h\; s) = \mathit{fmap}\; (D\; h)\; (h\; s)\end{aligned}$$

This yields a piece of data (the outermost type constructor f) with codata as components (the inner occurrences of $\mathit{ADT}\; f$). This construction justifies a number of earlier attempts to treat algebraic datatypes abstractly [12,38,50,60].

The idea of using final coalgebras as the semantics of abstract datatypes has a long history. Wand [59] writes that "an abstract data type is a final object in the category of its representations", and Kamin [26] that "only externally observable behavior matters [...] the final data type is the most abstract realization of any given data abstraction." Considering how close the relationship between abstract datatypes and object-oriented classes is, it is surprising that it seems to have taken over a decade for the idea to arise that final coalgebras provide a semantics for classes too [23,45]. For a good historical review of coinduction for behavioural satisfaction, see [20].

5.2 Summary

We have presented an approach to reasoning about abstract datatypes in a functional language, based on the (well-known) model of abstraction through existential quantification over the hidden representation type [28,29,32], and the (somewhat less well-known and appreciated) reasoning principles for codata through universal properties of final coalgebras [15,16]. We have illustrated this approach by considering a problem arising from Coutts *et al.*'s work [9] on stream fusion. In a nutshell, we advocate the following steps for reasoning about abstract datatypes:

- express the signature as a (strictly positive) functor f;
- enforce the abstraction via existential quantification:

$$\textbf{data}\ \mathit{Functor}\ f \Rightarrow \mathit{ADT}\ f = \exists s.\ D\ (s \to f\ s)\ s$$

- capture observations as concrete data:

$$\textbf{data}\ \mathit{Tree}\ f = T\{\mathit{unT} :: f\ (\mathit{Tree}\ f)\}$$

- transform abstract data to concrete data:

$$\begin{array}{l}
\mathit{tree} :: \mathit{Functor}\ f \Rightarrow \mathit{ADT}\ f \to \mathit{Tree}\ f \\
\mathit{tree}\ (D\ h\ s) = \mathit{unfold}\ h\ s\ \textbf{where} \\
\quad \mathit{unfold} :: \mathit{Functor}\ f \Rightarrow (a \to f\ a) \to a \to \mathit{Tree}\ f \\
\quad \mathit{unfold}\ f\ x = T\ (\mathit{fmap}\ (\mathit{unfold}\ f)\ (f\ x))
\end{array}$$

- exploit the universal property of *unfold* for reasoning:

$$h = \mathit{unfold}\ f \iff \mathit{unT} \cdot h = \mathit{fmap}\ h \cdot f$$

- view data as a mixture of concrete and abstract:

$$\mathit{unpack} :: \mathit{Functor}\ f \Rightarrow \mathit{ADT}\ f \to f\ (\mathit{ADT}\ f)$$

Acknowledgements

Particular thanks are due to Duncan Coutts, whose talk about his paper [9] with Roman Leshchinskiy and Don Stewart posed the question that inspired this work. Richard Bird's paper [1] was also an influence, not least on the title. I would also like to thank the Algebra of Programming group at Oxford and Pablo Nogueira, for helpful contributions and discussions, and the anonymous reviewers, whose comments have led to significant improvements.

References

1. Bird, R.: Unfolding pointer algorithms. Journal of Functional Programming 11(3), 347–358 (2001)
2. Bird, R.S.: Introduction to Functional Programming Using Haskell. Prentice-Hall, Englewood Cliffs (1998)
3. Böhm, C., Berarducci, A.: Automatic synthesis of typed λ-programs on term algebras. Theoretical Computer Science 39, 135–154 (1985)
4. Bruce, K., Cardelli, L., Castagna, G.: The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. Theory and Practice of Object Systems 1(3), 221–242 (1995)

5. Burton, F.W., Cameron, R.D.: Pattern matching with abstract data types. Journal of Functional Programming 3(2), 171–190 (1993)
6. Burton, W., Meijer, E., Sansom, P., Thompson, S., Wadler, P.: Views: An extension to Haskell pattern matching (October 1996), `http://www.haskell.org/development/views.html`
7. Cockett, R., Fukushima, T.: About Charity. Department of Computer Science, University of Calgary (May 1992)
8. Cook, W.R.: Object-oriented programming versus abstract data types. In: de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1990. LNCS, vol. 489, pp. 151–178. Springer, Heidelberg (1991)
9. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion: From lists to streams to nothing at all. In: International Conference on Functional Programming, pp. 315–326 (2007)
10. Dijkstra, E.W.: The humble programmer. Communications of the ACM 15(10), 859–866 (1972), `http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html`
11. Emir, B., Odersky, M., Williams, J.: Matching Objects with Patterns. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 273–298. Springer, Heidelberg (2007)
12. Erwig, M.: Categorical Programming with Abstract Data Types. In: Haeberer, A.M. (ed.) AMAST 1998. LNCS, vol. 1548, pp. 406–421. Springer, Heidelberg (1998)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
14. Gibbons, J.: Datatype-generic programming. In: Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.) SSDGP 2006. LNCS, vol. 4719. Springer, Heidelberg (2007)
15. Gibbons, J., Hutton, G.: Proof methods for corecursive programs. Fundamenta Informaticae 66(4), 353–366 (2005)
16. Gibbons, J., Jones, G.: The under-appreciated unfold. In: International Conference on Functional Programming, Baltimore, Maryland, pp. 273–279 (September 1998)
17. Gill, A.: Cheap Deforestation for Non-strict Functional Languages. PhD thesis, Glasgow (1998)
18. Gill, A., Launchbury, J., Jones, S.P.: A short cut to deforestation. In: Functional Programming Languages and Computer Architecture (1993)
19. Girard, J.-Y., Lafont, Y., Taylor, P.: Proofs and Types. Tracts in Theoretical Computer Science, vol. 7. Cambridge University Press, Cambridge (1989), `http://www.monad.me.uk/stable/Proofs+Types.html`
20. Goguen, J., Malcolm, G.: Hidden coinduction: Behavioural correctness proofs for objects. Mathematical Structures in Computer Science 9, 287–319 (1999)
21. Gordon, A.D.: A tutorial on co-induction and functional programming. In: Glasgow Workshop on Functional Programming (1994)
22. Howe, D.: Free on-line dictionary of computing (1993), `http://foldoc.org`
23. Jacobs, B.: Objects and classes, coalgebraically. In: Freitag, B., Jones, C.B., Lengauer, C., Schek, H.-J. (eds.) Object-Orientation with Parallelism and Persistence, pp. 83–103. Kluwer, Dordrecht (1996)
24. Jacobs, B.: Coalgebras in specification and verification for object-oriented languages. Newsletter of the Dutch Association for Theoretical Computer Science (NVTI) 3, 15–27 (1999)
25. Jacobs, B., Rutten, J.: A tutorial on (co)algebras and (co)induction. Bulletin of the European Association for Theoretical Computer Science (62), 222–259 (1997)

26. Kamin, S.: Final data types and their specification. ACM Transactions on Programming Languages and Systems 5(1), 97–123 (1983)
27. R.B. Kieburtz.: Codata and comonads in Haskell. Oregon Graduate Institute (unpublished manuscript, 1999)
28. Läufer, K., Odersky, M.: An extension of ML with first-class abstract types. In: SIGPLAN Workshop on ML and its Applications (June 1992)
29. Läufer, K., Odersky, M.: Polymorphic type inference and abstract data types. ACM Transactions on Programming Languages and Systems 16(5), 1411–1430 (1994)
30. Liskov, B., Zilles, S.: Programming with abstract data types. In: ACM SIGPLAN Symposium on Very High Level Languages, pp. 50–59. ACM Press, New York (1974)
31. Mitchell, J.C.: On the equivalence of data representations. In: Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, pp. 305–329. Academic Press Professional, Inc., San Diego (1991)
32. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. ACM Transactions on Programming Languages and Systems 10(3), 470–502 (1988)
33. Németh, L.: Catamorphism Based Program Transformations for Non-Strict Functional Languages. PhD thesis, Department of Computing Science, University of Glasgow (2000)
34. Okasaki, C.: Views for Standard ML. In: SIGPLAN Workshop in ML, pp. 14–23 (1998)
35. Gostanza, P.P., Peña, R., Núñez, M.: A new look at pattern matching in abstract data types. In: International Conference on Functional Programming, pp. 110–121. ACM Press, New York (1996)
36. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)
37. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
38. Paterson, R.: Haskell hierarchical libraries: Data.Sequence (accessed, 2007), `http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Sequence.html`
39. Jones, S.P.: Explicit quantification in Haskell (1998), `http://research.microsoft.com/~simonpj/Haskell/quantification.html`
40. Jones, S.P.: The Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge (2003)
41. Jones, S.P.: View patterns: Lightweight views for Haskell (January 2007), `http://hackage.haskell.org/trac/ghc/wiki/ViewPatternsArchive`
42. Jones, S.P., Tolmach, A., Hoare, T.: Playing by the rules: Rewriting as a practical optimisation technique in GHC. In: Hinze, R. (ed.) Haskell Workshop (2001)
43. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
44. Plotkin, G., Abadi, M.: A logic for parametric polymorphism. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 361–375. Springer, Heidelberg (1993)
45. Reichel, H.: An approach to object semantics based on terminal co-algebras. Mathematical Structures in Computer Science 5, 129–152 (1995)
46. Reynolds, J.C.: Towards a theory of type structure. In: Robinet, B. (ed.) Programming Symposium. LNCS, vol. 19, pp. 408–425. Springer, Heidelberg (1974)
47. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: Information Processing, vol. 83, pp. 513–523. Elsevier, Amsterdam (1983)

48. Svenningsson, J.: Shortcut fusion for accumulating parameters and zip-like functions. In: International Conference on Functional Programming (2002)
49. Syme, D., Neverov, G., Margetson, J.: Extensible pattern matching via a lightweight language extension. In: International Conference on Functional Programming, pp. 29–40 (2007)
50. Thompson, S.: Higher-order + polymorphic = reusable. Technical Report 224, University of Kent at Canterbury (May 1997), `http://www.cs.kent.ac.uk/pubs/1997/224/`
51. Tullsen, M.: First Class Patterns. In: Pontelli, E., Santos Costa, V. (eds.) PADL 2000. LNCS, vol. 1753, pp. 1–15. Springer, Heidelberg (2000)
52. Turner, D.A.: Elementary strong functional programming. In: Hartel, P.H., Plasmeijer, R. (eds.) FPLE 1995. LNCS, vol. 1022. Springer, Heidelberg (1995)
53. Turner, D.A.: Total functional programming. Journal of Universal Computer Science 10(7), 751–768 (2004)
54. Ungar, D., Smith, R.B.: Self: The power of simplicity. In: Object-Oriented Programming: Systems, Languages and Applications, pp. 227–242 (1987)
55. Wadler, P.: A critique of Abelson and Sussman: Why calculating is better than scheming. SIGPLAN Notices 22(3), 8 (1987)
56. Wadler, P.: Views: A way for mattern matching to cohabit with data abstraction. In: Principles of Programming Languages, pp. 307–313. ACM Press, New York (1987)
57. Wadler, P.: Theorems for free! In: Functional Programming Languages and Computer Architecture, pp. 347–359. ACM, New York (1989)
58. Wadler, P.: Recursive types for free! (July 1990) (unpublished manuscript), `http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`
59. Wand, M.: Final algebra semantics and data type extensions. Journal of Computer and System Sciences 19, 27–44 (1979)
60. Wang, D.C., Murphy VII, T.: Programming with recursion schemes. Agere Systems/Carnegie Mellon University (unpublished manuscript, 2002)
61. Wang, M., Gibbons, J.: Translucent abstraction: Algebraic datatypes with safe views (April 2008) (submitted)
62. Wraith, G.: A note on categorical datatypes. In: Pitt, D.H., Rydeheard, D.E., Dyjber, P., Pitts, A.M., Poigné, A. (eds.) Category Theory and Computer Science. LNCS, vol. 389. Springer, Heidelberg (1989)

A Appendix: Equivalence of Complex Numbers

Section 3.4 shows how to prove observational equivalence of different implementations of complex numbers, using the fusion property of *unfold*. Here, we discharge the proof obligation

$$\mathit{fc} \cdot \mathit{p2c} = \mathit{fmap}\ \mathit{p2c} \cdot \mathit{fp}$$

using the (idealised, since not quite valid in approximate floating point numbers) equation relating polar and Cartesian representations

$$\mathit{p2c} \cdot \mathit{c2p} = \mathit{id}$$

and the definitions

$$\begin{aligned}
\mathit{fc}\,(x,y) = \mathit{CF}\{&\_\mathit{new} = \lambda z \rightarrow z,\\
&\_\mathit{add} = \lambda c \rightarrow (x + \mathit{re}\ c, y + \mathit{im}\ c),\\
&\_\mathit{rea} = x,\\
&\_\mathit{ima} = y\})\\
\mathit{fp}\ p \qquad = \mathit{CF}\{&\_\mathit{new} = \lambda z \rightarrow \mathit{c2p}\ z,\\
&\_\mathit{add} = \lambda c \rightarrow \mathbf{let}\ (x,y) = \mathit{p2c}\ p\ \mathbf{in}\\
&\qquad\qquad\quad \mathit{c2p}\ (x + \mathit{re}\ c, y + \mathit{im}\ c),\\
&\_\mathit{rea} = \mathit{fst}\ (\mathit{p2c}\ p),\\
&\_\mathit{ima} = \mathit{snd}\ (\mathit{p2c}\ p)\})
\end{aligned}$$

Note also that the appropriate definition of *fmap* on *ComplexF* is

$$\begin{aligned}
\mathit{fmap}\ f\ c = \mathit{CF}\{&\_\mathit{new} = \lambda z \rightarrow f\ (\_\mathit{new}\ c\ z),\\
&\_\mathit{add} = \lambda c' \rightarrow f\ (\_\mathit{add}\ c\ c'),\\
&\_\mathit{rea} = \_\mathit{rea}\ c,\\
&\_\mathit{ima} = \_\mathit{ima}\ c\}
\end{aligned}$$

We calculate:

$$\begin{aligned}
&\quad \mathit{fmap}\ \mathit{p2c}\ (\mathit{fp}\ p)\\
&= \quad \{\ \mathit{fp}\ \}\\
&\quad \mathit{fmap}\ \mathit{p2c}\ (\mathit{CF}\{\_\mathit{new} = \lambda z \rightarrow \mathit{c2p}\ z,\\
&\qquad\qquad\qquad\quad \_\mathit{add} = \lambda c \rightarrow \mathbf{let}\ (x,y) = \mathit{p2c}\ p\ \mathbf{in}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{c2p}\ (x + \mathit{re}\ c, y + \mathit{im}\ c),\\
&\qquad\qquad\qquad\quad \_\mathit{rea} = \mathit{fst}\ (\mathit{p2c}\ p),\\
&\qquad\qquad\qquad\quad \_\mathit{ima} = \mathit{snd}\ (\mathit{p2c}\ p)\})\\
&= \quad \{\ \mathit{fmap}\ \text{on}\ \mathit{ComplexF}\ \}\\
&\quad \mathit{CF}\{\_\mathit{new} = \lambda z \rightarrow \mathit{p2c}\ (\mathit{c2p}\ z),\\
&\qquad\quad \_\mathit{add} = \lambda c \rightarrow \mathbf{let}\ (x,y) = \mathit{p2c}\ p\ \mathbf{in}\\
&\qquad\qquad\qquad\qquad \mathit{p2c}\ (\mathit{c2p}\ (x + \mathit{re}\ c, y + \mathit{im}\ c)),\\
&\qquad\quad \_\mathit{rea} = \mathit{fst}\ (\mathit{p2c}\ p),\\
&\qquad\quad \_\mathit{ima} = \mathit{snd}\ (\mathit{p2c}\ p)\}\\
&= \quad \{\ \mathit{p2c} \cdot \mathit{c2p} = \mathit{id}\ \}\\
&\quad \mathit{CF}\{\_\mathit{new} = \lambda z \rightarrow z,\\
&\qquad\quad \_\mathit{add} = \lambda c \rightarrow \mathbf{let}\ (x,y) = \mathit{p2c}\ p\ \mathbf{in}\\
&\qquad\qquad\qquad\qquad (x + \mathit{re}\ c, y + \mathit{im}\ c),\\
&\qquad\quad \_\mathit{rea} = \mathit{fst}\ (\mathit{p2c}\ p),\\
&\qquad\quad \_\mathit{ima} = \mathit{snd}\ (\mathit{p2c}\ p)\}\\
&= \quad \{\ \text{lift out the}\ \mathbf{let}\ \text{binding}\ \}\\
&\quad \mathbf{let}\ (x,y) = \mathit{p2c}\ p\ \mathbf{in}\\
&\quad \mathit{CF}\{\_\mathit{new} = \lambda z \rightarrow z,\\
&\qquad\quad \_\mathit{add} = \lambda c \rightarrow (x + \mathit{re}\ c, y + \mathit{im}\ c),\\
&\qquad\quad \_\mathit{rea} = x,\\
&\qquad\quad \_\mathit{ima} = y\}\\
&= \quad \{\ \mathit{fc}\ \}
\end{aligned}$$

```
    let (x, y) = p2c p in fc (x, y)
  = {  application  }
    fc (p2c p)
```

which completes the proof.

B Appendix: Equivalence of Skipping Streams

Section 4.8 makes a claim about the observational equivalence modulo *Skip*s of the skipping streams *concatS* (*wrapS s*) and *s*, where

```
concatS :: SStream (SStream a) → SStream a
concatS (D hs ss) = D hc (Nothing, ss) where
  hc (Nothing, ss) = case hs ss of
    Done          → Done
    Skip ss′      → Skip (Nothing, ss′)
    Yield s ss′   → Skip (Just s, ss′)
  hc (Just (D ha sa), ss) = case ha sa of
    Done          → Skip (Nothing, ss)
    Skip sa′      → Skip (Just (D ha sa′), ss)
    Yield y sa′   → Yield y (Just (D ha sa′), ss)
wrapS :: a → SStream a
wrapS x = D fetch (Just x) where
  fetch (Just x) = Yield x Nothing
  fetch Nothing = Done
```

The claim boils down to the following equation between functions on lists,

$$unsstream \cdot concatS \cdot wrapS = unsstream$$

where

```
unsstream :: SStream a → [a]
unsstream (D h s) = unfoldr (force h) s where
  force h s = case h s of Done       → Nothing₂
                          Yield x s′ → Just₂ x s′
                          Skip s′    → force h s′
```

Consider for example the skipping stream $s = D\ h\ 0$ where

```
h n = [Skip 1, Yield 'a' 2, Skip 3, Yield 'b' 4, Skip 5, Done] !! n
```

(The operation '!!' denotes list indexing.) Unwinding this stream proceeds through each of the above states in turn, yielding in total the list of characters ['a', 'b']. The stream *concatS* (*wrapS s*), on the other hand, exhibits the following behaviour:

state	output
$(\mathit{Nothing}, \mathit{Just}\ (D\ h\ 0))$	Skip
$(\mathit{Just}\ (D\ h\ 0), \mathit{Nothing})$	Skip
$(\mathit{Just}\ (D\ h\ 1), \mathit{Nothing})$	Yield `'a'`
$(\mathit{Just}\ (D\ h\ 2), \mathit{Nothing})$	Skip
$(\mathit{Just}\ (D\ h\ 3), \mathit{Nothing})$	Yield `'b'`
$(\mathit{Just}\ (D\ h\ 4), \mathit{Nothing})$	Skip
$(\mathit{Just}\ (D\ h\ 5), \mathit{Nothing})$	Skip
$(\mathit{Nothing}, \mathit{Nothing})$	Done

Each row of the table presents a state and the output from that state, omitting the successor state if present. For example,

$$\mathit{hc}\ (\mathit{Just}\ (D\ h\ 1), \mathit{Nothing}) = \mathit{Yield}\ \texttt{'a'}\ (\mathit{Just}\ (D\ h\ 2), \mathit{Nothing})$$

Evidently the closest match between these states and those of the original stream s are those whose first component is a *Just*. We therefore proceed to show that

$$\begin{array}{ll}
 & \mathit{unsstream}\ (\mathit{concatS}\ (\mathit{wrapS}\ s)) \\
= & \{\ \text{definitions}\ \} \\
 & \mathit{unfoldr}\ (\mathit{force}\ (\mathit{hc}\ \mathit{fetch}))\ (\mathit{Nothing}, \mathit{Just}\ (D\ h\ n)) \\
= & \{\ \text{expanding: } f\ x = f\ y \Longrightarrow \mathit{unfoldr}\ f\ x = \mathit{unfoldr}\ f\ y\ \} \\
 & \mathit{unfoldr}\ (\mathit{force}\ (\mathit{hc}\ \mathit{fetch}))\ (\mathit{Just}\ (S\ h\ n), \mathit{Nothing}) \\
= & \{\ \mathit{unfoldr}\ \text{fusion}\ \} \\
 & \mathit{unfoldr}\ (\mathit{force}\ h)\ n \\
= & \{\ \text{definitions}\ \} \\
 & \mathit{unsstream}\ (D\ h\ n)
\end{array}$$

For the 'expansion' step, it is easy to verify that

$$\mathit{fetch}\ (\mathit{Just}\ (D\ h\ n)) = \mathit{Yield}\ (D\ h\ n)\ \mathit{Nothing}$$

and so

$$\mathit{hc}\ \mathit{fetch}\ (\mathit{Nothing}, \mathit{Just}\ (D\ h\ n)) = \mathit{Skip}\ (\mathit{Just}\ (D\ h\ n), \mathit{Nothing})$$

and so

$$\begin{array}{ll}
 & \mathit{force}\ (\mathit{hc}\ \mathit{fetch})\ (\mathit{Nothing}, \mathit{Just}\ (D\ h\ n)) \\
= & \\
 & \mathit{force}\ (\mathit{hc}\ \mathit{fetch})\ (\mathit{Just}\ (D\ h\ n), \mathit{Nothing})
\end{array}$$

as required.

For the 'fusion' step, we define

$$\mathit{inject}\ n = (\mathit{Just}\ (D\ h\ n), \mathit{Nothing})$$

so that the obligation is to prove

$$\mathit{unfoldr}\ (\mathit{force}\ (\mathit{fc}\ \mathit{fetch})) \cdot \mathit{inject} = \mathit{unfoldr}\ (\mathit{force}\ h)$$

This can be done using the fusion rule for *unfoldr*:

$$\mathit{unfoldr}\;f \cdot g = \mathit{unfoldr}\;f' \;\Longleftarrow\; f \cdot g = \mathit{fmap}\;(\mathit{prod}\;\mathit{id}\;g) \cdot f'$$

which reduces the obligation to showing that

$$\mathit{force}\;(\mathit{hc}\;\mathit{fetch}) \cdot \mathit{inject} = \mathit{fmap}\;(\mathit{prod}\;\mathit{id}\;\mathit{inject}) \cdot \mathit{force}\;h$$

This last step has to be done using fixpoint induction, because *force* is not defined using a structured form of recursion. We simplify both sides to the point at which they make a recursive call to *force*; the surrounding contexts turn out to be equal, and so fixpoint induction shows that the least fixpoints are equal. On the left-hand side, we have:

$$\begin{array}{ll}
 & \mathit{force}\;(\mathit{hc}\;\mathit{fetch})\;(\mathit{inject}\;n) \\
= & \quad \{\ \mathit{inject}\ \} \\
 & \mathit{force}\;(\mathit{hc}\;\mathit{fetch})\;(\mathit{Just}\;(D\;h\;n), \mathit{Nothing}) \\
= & \quad \{\ \mathit{force}, \mathit{hc}, \mathit{fetch}\ \} \\
 & \textbf{case}\;h\;n\;\textbf{of} \\
 & \quad \begin{array}{lcl} \mathit{Done} & \rightarrow & \mathit{Nothing}_2 \\ \mathit{Skip}\;m & \rightarrow & \mathit{force}\;(\mathit{hc}\;\mathit{fetch})\;(\mathit{Just}\;(D\;h\;m), \mathit{Nothing}) \\ \mathit{Yield}\;x\;m & \rightarrow & \mathit{Just}_2\;x\;(\mathit{Just}\;(D\;h\;m), \mathit{Nothing}) \end{array} \\
= & \quad \{\ \mathit{inject}\ \} \\
 & \textbf{case}\;h\;n\;\textbf{of} \\
 & \quad \begin{array}{lcl} \mathit{Done} & \rightarrow & \mathit{Nothing}_2 \\ \mathit{Skip}\;m & \rightarrow & \mathit{force}\;(\mathit{hc}\;\mathit{fetch})\;(\mathit{inject}\;m) \\ \mathit{Yield}\;x\;m & \rightarrow & \mathit{Just}_2\;x\;(\mathit{inject}\;m) \end{array}
\end{array}$$

On the right, we have:

$$\begin{array}{ll}
 & \mathit{fmap}\;(\mathit{prod}\;\mathit{id}\;\mathit{inject})\;(\mathit{force}\;h\;n) \\
= & \quad \{\ \mathit{force}\ \} \\
 & \mathit{fmap}\;(\mathit{prod}\;\mathit{id}\;\mathit{inject})\;(\textbf{case}\;h\;n\;\textbf{of} \\
 & \quad \begin{array}{lcl} \mathit{Done} & \rightarrow & \mathit{Nothing}_2 \\ \mathit{Skip}\;m & \rightarrow & \mathit{force}\;h\;m \\ \mathit{Yield}\;x\;m & \rightarrow & \mathit{Just}_2\;x\;m) \end{array} \\
= & \quad \{\ \mathit{fmap}, \mathit{prod}\ \} \\
 & \textbf{case}\;h\;n\;\textbf{of} \\
 & \quad \begin{array}{lcl} \mathit{Done} & \rightarrow & \mathit{Nothing}_2 \\ \mathit{Skip}\;m & \rightarrow & \mathit{fmap}\;(\mathit{prod}\;\mathit{id}\;\mathit{inject})\;(\mathit{force}\;h\;m) \\ \mathit{Yield}\;x\;m & \rightarrow & \mathit{Just}_2\;x\;(\mathit{inject}\;m) \end{array}
\end{array}$$

This completes the proof.

Circulations, Fuzzy Relations and Semirings

Roland Glück and Bernhard Möller

Institut für Informatik, Universität Augsburg,
Universitätsstr. 14, D-86135 Augsburg, Germany
{glueck,moeller}@informatik.uni-augsburg.de
http://www.informatik.uni-augsburg.de/en/chairs/dbis/
pmi/staff/{glueck,moeller}

Abstract. Circulations are similar to flows in capacity-constrained networks, with the difference that they also observe lower bounds and, unlike flows, are not directed from a source to a sink. We give a new description of circulations in networks using a technique introduced by Kawahara; he applied the same methods to network flows. We show the power and flexibility of his approach in a new application, refining it at the same time by introducing the concept of test relations. Furthermore we will give algebraic formulations of a generic algorithm for computing a flow in a network with lower bounds and a sufficient and necessary criterion for the existence of a circulation.

1 Introduction

Networks with and without lower bounds as well as flows and circulations in them have a wide range of applications. They are used for transport problems, for modelling financial and economic situations and are also used in graph theory. Common proofs of Hall's theorem and of the Egervary-König-Theorem ([Jun], Chapter 7) use networks, too. Also problems concerning matchings in bipartite graphs and disjoint path problems can often be solved using networks. So there are a lot of algorithms for and theorems about them.

Usually networks are described as graphs with weighted edges. In the approach we are using they are modelled as so-called fuzzy relations, a natural generalisation of traditional relations. This idea was introduced in [Kaw]. The advantage of this approach is that a lot of proofs can be done in an algebraic manner by simple calculation. This opens the door for automated reasoning about networks and related topics in graph theory.

We take up Kawahara's approach in a new application. At the same time we refine it by introducing the concept of fuzzy test relations, inspired by the theory of tests in semirings [KozT, MB], which leads to a substantial notational and conceptual simplification. This will be beneficial for automated proofs in this problem domain.

In Section two we present the basics of fuzzy relations and other tools we will use. Section three gives an overview over recent algebraic work concerning flows in networks. Finally, from Section four on we present our new ideas about circulations in networks with lower bounds.

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 134–152, 2008.

2 Fuzzy Relations

2.1 Definition and Basic Operations

Definition 2.1 (Fuzzy Relation). A *fuzzy relation* α between sets X and Y, written $\alpha : X \leftrightarrow Y$, is a mapping from $X \times Y$ into the interval $[0, 1]$. The set of all fuzzy relations between X and Y is denoted by $Rel(X, Y)$. A fuzzy relation between a set X and itself is called a *fuzzy endorelation* on X.

A fuzzy endorelation α on a set X can be viewed as representing a weighted directed graph with node set X, where for $x, y \in X$ the value $\alpha(x, y)$ is the weight of edge (x, y). An edge weight $\alpha(x, y) = 0$ means that x and y are considered not to be connected by a direct edge. The notion of a fuzzy relation can be generalised to allow elements of an arbitrary lattice as edge weights. However, the above special case has a number of advantages in our setting; they will be discussed as we go along.

Definition 2.2 (Special Relations). Given sets X and Y, there are three particular fuzzy relations with

$$\begin{array}{ll} \mathbf{0}_{XY} : X \leftrightarrow Y\ , & \mathbf{0}(x,y) = 0\ , \\ \nabla_{XY} : X \leftrightarrow Y\ , & \nabla(x,y) = 1 \\ id_X\ \ : X \leftrightarrow X\ , & id_X(x,y) = \begin{cases} 1 \text{ if } x = y \\ 0 \text{ otherwise} \end{cases} \end{array}$$

for all x, y, called the *empty*, *universal* and *identity* fuzzy relation, resp. When X and Y are clear from the context the indices will be omitted.

In the graph view, $\mathbf{0}$ represents the totally disconnected graph while ∇ is the fully connected graph in which all edges have maximal weight. Finally, *id* is a graph in which every node carries a loop of maximal weight and there are no other edges.

Definition 2.3 (Boolean Fuzzy Relation). A fuzzy relation is called *Boolean* if its range is contained in the set $\{0, 1\}$.

A Boolean fuzzy relation $\alpha : X \leftrightarrow Y$ corresponds in a natural way to a standard relation $\widetilde{\alpha} \subseteq X \times Y$ given by

$$x\, \widetilde{\alpha}\, y \ \Leftrightarrow\ \alpha(x, y) = 1\ .$$

To motivate the following definitions we will now explain how fuzzy relations can be used to model flows and circulations.

A *network* is an edge-labeled directed graph with at most one edge between any two nodes, in which the edge labels are non-negative real numbers modelling transport capacities between the nodes. If there is an upper bound $c \neq 0$ for the capacities (which holds, in particular, for finite networks) we can normalise them to the interval $[0, 1]$ by dividing them by c. Now the network can immediately be represented by a fuzzy relation α, which takes value $\frac{d}{c}$ if there is an edge with weight d from x to y and zero otherwise.

It is well known how to use standard relations for describing certain aspects of graphs. We want to extend these techniques to networks and their describing fuzzy relations.

To this end we first introduce some operations on the interval $[0,1]$. For $a, b \in [0,1]$ we define

- $a \vee b = \max\{a, b\}$
- $a \wedge b = \min\{a, b\}$
- $a \ominus b = \max\{0, a - b\}$
- $a \oplus b = \min\{1, a + b\}$

Consistent with these definitions we define the operators $\bigvee_{x \in X} x$ and $\bigwedge_{x \in X} x$ for arbitrary subsets $X \subseteq [0,1]$ by $\bigvee_{x \in X} x = \sup X$ and $\bigwedge_{x \in X} x = \inf X$.

Definition 2.4 (Basic Operations). For fuzzy relations $\alpha, \beta : X \leftrightarrow Y$ the join $\alpha \sqcup \beta$, the meet $\alpha \sqcap \beta$, the truncating difference $\alpha \ominus \beta$ and the truncating sum $\alpha \oplus \beta : X \leftrightarrow Y$ are the pointwise extensions of the operators defined above:

- $\forall x \in X\ \forall y \in Y : (\alpha \sqcup \beta)(x, y) = \alpha(x, y) \vee \beta(x, y)$
- $\forall x \in X\ \forall y \in Y : (\alpha \sqcap \beta)(x, y) = \alpha(x, y) \wedge \beta(x, y)$
- $\forall x \in X\ \forall y \in Y : (\alpha \ominus \beta)(x, y) = \alpha(x, y) \ominus \beta(x, y)$
- $\forall x \in X\ \forall y \in Y : (\alpha \oplus \beta)(x, y) = \alpha(x, y) \oplus \beta(x, y)$

Two fuzzy relations $\alpha, \beta : X \leftrightarrow Y$ are said to be *disjoint* if $\alpha \sqcap \beta = \mathbf{0}$. Adapting the usual notation $A \dot{\cup} B$ for the union of disjoint sets A and B, we abbreviate the union of two disjoint fuzzy relations α and β by $\alpha \dot{\sqcup} \beta$. Moreover, we write $\alpha \sqsubseteq \beta$ if $\alpha(x, y) \leq \beta(x, y)$ holds for all $(x, y) \in X \times Y$.

Analogously to above we extend $\sqcup$ to sets of fuzzy relations, writing $\bigsqcup_{i \in I} \alpha_i$.

It is easy to see that meet, join and truncating sum are commutative and associative. Moreover, join distributes over meet and vice versa.

The behaviour of the truncating sum is more complex. Due to the truncation the common laws of addition and subtraction are not transferable. With additional assumptions similar rules are valid. For example $\beta \sqsubseteq \alpha$ implies $(\alpha \ominus \beta) \oplus \beta = \alpha$. In Section 2.3 we will state similar properties concerning the connection between the above operations and the cardinality of fuzzy relations.

Definition 2.5 (Scalar Multiplication). For real numbers $k \in [0,1]$ and a fuzzy relation $\alpha : X \leftrightarrow Y$ the *scalar multiplication* $k \cdot \alpha$, also written $k\alpha$, is defined by $(k\alpha)(x, y) = k \cdot \alpha(x, y)$ for all $(x, y) \in X \times Y$.

Scalar multiplication distributes over $\sqcup$, $\sqcap$, $\ominus$ and $\oplus$.

Definition 2.6 (Converse). For a fuzzy relation $\alpha : X \leftrightarrow Y$ the *converse* $\alpha^\sharp : Y \leftrightarrow X$ is defined by

$$\forall x \in X\ \forall y \in Y : \alpha^\sharp(y, x) = \alpha(x, y).$$

Converse commutes with scalar multiplication, i.e., $(k\alpha)^\sharp = k(\alpha^\sharp)$, and distributes over $\sqcup, \sqcap, \ominus$ and $\oplus$. In the graph view this operation reverses all edges while preserving their weights.

2.2 Composition, Powers and Star

Definition 2.7 (Composition). We define the *composition* $\alpha\beta : X \leftrightarrow Z$ of two fuzzy relations $\alpha : X \leftrightarrow Y$ and $\beta : Y \leftrightarrow Z$ by

$$\alpha\beta(x,z) = \bigvee\nolimits_{y \in Y}(\alpha(x,y) \wedge \beta(y,z)).$$

This is a straightforward generalisation of the composition of standard relations. It correctly describes the capacity behaviour along network paths: If there are edges from x to y and from y to z with capacities $\alpha(x,y)$ and $\beta(y,z)$ then at most $\alpha(x,y) \wedge \beta(y,z)$ can be transported from x to z along the concatenation of these two edges. Hence $\alpha\beta$ describes the supremum of transport capacity over all two-edge paths from x to z.

Since the supremum of a bounded subset of the real numbers always exists and is unique, the composition of two fuzzy relations is well defined even if one of the sets X, Y, Z is infinite.

A standard relation $R \subseteq X \times Y$ is called univalent if it relates every element of X to at most one element of Y. This is expressed algebraically by the condition $R^\sharp R \subseteq id_Y$. The dual notion of injectivity is characterised by $RR^\sharp \subseteq id_X$. As a generalisation we call a fuzzy relation *univalent* if $\alpha^\sharp\alpha \sqsubseteq id_Y$ holds. If α satisfies the property $\alpha\alpha^\sharp \sqsubseteq id_X$ it is called *injective.*

The composition of fuzzy relations distributes over join, i.e., $\alpha(\beta \sqcup \gamma) = \alpha\beta \sqcup \alpha\gamma$ and $(\alpha \sqcup \beta)\gamma = \alpha\gamma \sqcup \beta\gamma$. In general composition does not distribute over meet; in this case only $(\alpha \sqcap \beta)\gamma \sqsubseteq \alpha\gamma \sqcap \beta\gamma$ and analogously $\alpha(\beta \sqcap \gamma) \sqsubseteq \alpha\beta \sqcap \alpha\gamma$ hold. However, for univalent α the equality $\alpha(\beta \sqcap \gamma) = \alpha\beta \sqcap \alpha\gamma$ and for injective γ the equality $(\alpha \sqcap \beta)\gamma = \alpha\gamma \sqcap \beta\gamma$ hold.

Composition commutes with scalar multiplication, i.e., $k(\alpha\beta) = (k\alpha)\beta = \alpha(k\beta)$. Finally, composition is contravariant w.r.t. converse, i.e., $(\alpha\beta)^\sharp = \beta^\sharp\alpha^\sharp$.

Definition 2.8 (Powers and Star). The *n-th power* α^n of a fuzzy endorelation $\alpha : X \leftrightarrow X$ is defined inductively by $\alpha^0 = id_X$ and $\alpha^{n+1} = \alpha\alpha^n$ for $n \in \mathbb{N}_0$. The *reflexive and transitive closure* α^* of a fuzzy endorelation is defined by $\alpha^* = \bigsqcup_{n \in \mathbb{N}_0} \alpha^n$.

An elementary argument shows the equality $\alpha^* = \bigsqcup_{0 \leq n < |X|} \alpha^n$ for every fuzzy endorelation α on a finite set X.

2.3 Cardinality of Fuzzy Relations

For a standard relation $R \subseteq X \times Y$ its cardinality $|R|$ is the number of pairs in R; it coincides with the sum $\sum_{(x,y) \in X \times Y} \alpha(x,y)$ where α is the Boolean fuzzy relation corresponding to R, i.e., $\alpha(x,y) = 1 \Leftrightarrow (x,y) \in R$. This is generalised in the following definition due to [Kaw].

Definition 2.9 (Cardinality). The *cardinality* $|\alpha|$ of a fuzzy relation $\alpha : X \leftrightarrow Y$ is defined by $|\alpha| = \sum_{(x,y) \in X \times Y} \alpha(x,y)$.

The cardinality will allow an elegant description of total weights of subnetworks. E.g., the cardinality of the relation consisting of all outgoing edges of a node describes the sum of the weights of all edges leaving that node. This is used, e.g., in Definition 3.2 of flows.

Obvious properties of the cardinality are:

- $|\alpha| \geq 0$
- $|\alpha| = 0 \Leftrightarrow \alpha = \mathbf{0}$
- $|\alpha| = |\alpha^\sharp|$
- Cardinality is an isotone function, i.e., $\alpha \sqsubseteq \beta$ implies $|\alpha| \leq |\beta|$.

The cardinality of a fuzzy relation could become infinite if one of the participating sets is infinite; in this paper we won't deal with such cases. If both X and Y are finite sets the cardinality of a fuzzy relation $\alpha : X \leftrightarrow Y$ is always a nonnegative real number bounded by $|X| \cdot |Y|$.

A fuzzy relation $\alpha : X \leftrightarrow Y$ is called *normalised* if $|\alpha| \leq 1$. This implies $|\beta| \leq 1$ for all fuzzy relations β with $\beta \sqsubseteq \alpha$. We will make use of normalised fuzzy relations when we want to label edges with the cardinality of a fuzzy relation.

A connection between meet and cardinality is the equation

$$|\alpha \sqcup \beta| = |\alpha| + |\beta| - |\alpha \sqcap \beta| \tag{1}$$

for arbitrary fuzzy relations $\alpha, \beta : X \leftrightarrow Y$. In particular, it states $|\alpha \sqcup \beta| = |\alpha| + |\beta|$ for disjoint fuzzy relations α and β.

For fuzzy relations α, β, γ with $\beta \sqsubseteq \alpha$ and $\gamma \sqsubseteq \alpha \ominus \beta$ the equality $|\gamma| + |\beta| = |\gamma \oplus \beta|$ holds. If $\beta \sqsubseteq \alpha$ then $|\alpha \ominus \beta| = |\alpha| \ominus |\beta|$.

2.4 Test Relations

Definition 2.10 (Test Relation). *A Boolean subrelation of id_X is called a test relation on X.*

A test relation τ corresponds in a natural way to the subset $\{x : \tau(x, x) = 1\}$ of X; conversely, every subset T of X can be represented by the test relation τ with $\tau(x, x) = 1 \Leftrightarrow x \in T$.

Hence the test relations form a Boolean subalgebra of $Rel(X, X)$ with $\tau \sqcup \sigma$ being the join and $\tau \sqcap \sigma = \tau\sigma$ being the meet.

Consider a general fuzzy relation α and a test relation τ. Then, by isotony of multiplication, $\tau\alpha \sqsubseteq id\,\alpha = \alpha$. So $\tau\alpha$ is a part of α; it omits all edges of α that do not start in a node in τ. This models enforcing the precondition τ. Similarly, $\alpha\tau$ removes all edges of α that do not end in a node in τ; it enforces τ as a postcondition.

Combining this with the cardinality operator we can express the input or output capacity of a set of nodes characterised by τ compactly as $|\alpha\tau|$ and $|\tau\alpha|$, respectively.

The following property will be used several times:

Lemma 2.11. *Assume $\tau, \sigma, \alpha, \beta, \gamma : X \leftrightarrow X$ such that τ, σ are test relations with $\tau\sigma = \mathbf{0}$ and $\gamma \sqsubseteq \tau\alpha$ and $\gamma \sqsubseteq \sigma\beta$.*

(a) $\gamma = \mathbf{0}$.
(b) $|\tau\alpha \sqcup \sigma\beta| = |\tau\alpha| + |\sigma\beta|$.

Proof. Part (a) follows from the general theory of semirings (see Lemma A.4 in the Appendix). Part (b) holds by Equation (1), since by Part (a) $\tau\alpha \sqcap \sigma\beta = \mathbf{0}$. □

Part (a) means that edge sets with disjoint sets of starting nodes are disjoint as well (remember that the composition of test relations is their conjunction).

To avoid excessive notation, in the remainder we will write τ both for a subset $\tau \subseteq X$ and its characterising test relation. The complement of τ relative to X/id_X is denoted by τ^c.

Moreover, lower case latin letters will be used to denote elements $x \in X$ and the corresponding test relations characterising the singleton sets $\{x\}$. These relations are also called *point relations*.

Every test relation can be written as the join of suitable point relations, i.e., $U = \bigsqcup_{u \in U} u$ where the join ranges over point relations u. In the finite case this can be used to show properties of test relations via structural induction.

Using point relations we can also describe single graph edges. Given points x, y and a weight $k \in [0, 1]$, the fuzzy relation $k(x\nabla y)$ describes a graph connecting just x and y with edge weight k: the test relations x and y reduce the universal relation ∇ to the relation consisting only of the edge from x to y with weight 1, and the weighting is done by scaling with the factor k.

3 Flows in Networks

Flows in networks were already treated in an algebraic manner in [Kaw] and [Glü]. We give a brief summary of the ideas from there relevant to the present paper. For the sake of well-definedness of cardinality let from now on all node sets be finite.

3.1 Networks and Flows

Definition 3.1 (Networks). For a set X of nodes, a fuzzy endorelation $\alpha : X \leftrightarrow X$ is also called a *pseudo-network*. For $x, y \in X$ we call $\alpha(x, y)$ the *capacity* of (x, y). If additionally $\alpha \sqcap \alpha^\sharp = \mathbf{0}$ then N is called a *network*. An *s-t-(pseudo-)network* is a triple $N = (\alpha : X \leftrightarrow X, s, t)$ consisting of a (pseudo-)network α and two distinct elements of X, namely s (the *source*) and t (the *sink*) of N.

Contrary to [Kaw] we admit input edges for the source and output edges for the sink, i.e., we give up the requirements $\alpha s = \mathbf{0}$ and $t\alpha = \mathbf{0}$, to obtain a more general concept.

Our definition of a pseudo-network corresponds to that of a network as given for example in [Jun], Chapter 6, or [AMO], Chapter 1, or [GTT], except that there the capacities can have arbitrary values in $\mathbb{R}_+$.

Our network requirement that excludes antiparallel edges between two nodes by the condition $\alpha \sqcap \alpha^\sharp = 0_{XX}$ may look too strong. But if we are given a fuzzy relation $\alpha : X \leftrightarrow X$ not fulfilling this condition we can construct a network $\hat{N} = (\hat{\alpha} : \hat{X} \leftrightarrow \hat{X}, s, t)$ as follows: for all pairs $(x, y) \in X \times Y$ with $\alpha(x, y) \sqcap \alpha^\sharp(x, y) \neq 0$ we introduce two additional nodes x' and y' and set $\hat{\alpha}(x, y) = \hat{\alpha}(y, x) = \hat{\alpha}(x', y') = \hat{\alpha}(y', x') = 0$, $\hat{\alpha}(x, x') = \hat{\alpha}(x', y) = \alpha(x, y)$, $\hat{\alpha}(y, y') = \hat{\alpha}(y', x) = \alpha(y, x)$ and $\hat{\alpha}(x', x) = \hat{\alpha}(y, x') = \hat{\alpha}(y', y) = \hat{\alpha}(x, y') = 0$.

This strategy turns a pair of antiparallel edges into a bypassing device:

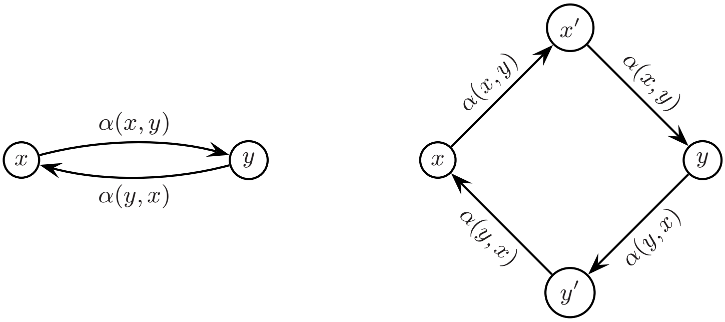


Fig. 1. Bypassing device

In a quite natural way the definition of a network flow (cf. again the standards [Jun] and [AMO]) is translated into the language of fuzzy relations:

Definition 3.2 (Flow). A *flow* φ in a pseudo-network $\alpha : X \leftrightarrow X$ is a fuzzy endorelation $\varphi : X \leftrightarrow X$ such that $\varphi \sqsubseteq \alpha$ and $|\tau\varphi| = |\varphi\tau|$ for all test relations $\tau : X \leftrightarrow X$ on X. A *flow* φ in an s-t-(pseudo-)network $(\alpha : X \leftrightarrow X, s, t)$ is a fuzzy endorelation on X with $\varphi \sqsubseteq \alpha$ and $|\tau\varphi| = |\varphi\tau|$ for all test relations τ with $\tau \sqsubseteq X \backslash \{s, t\}$.

The first part of this definition is the *capacity constraint*: the flow along an edge can be at most as high as allowed by the capacity of that edge. The second part corresponds to the *flow conservation*, commonly written as

$$\textstyle\sum_{v\in V} \varphi(v, u) = \sum_{v\in V} \varphi(u, v) \ \forall u \in X \setminus \{s, t\}, \qquad (*)$$

see, e.g., [Jun], p.147 (with a slightly different notation). Our version of flow conservation seems to be stronger than $(*)$, but the equivalence of the two formulations can be shown by induction over the size of the tests (cf. [Glü], Section 6.2). There is always at least the trivial flow $\mathbf{0}$ in any network $N = (\alpha : X \leftrightarrow X, s, t)$.

Because of flow conservation, in bypassing devices as in Fig. 1 there is a one-to-one correspondence between flows in the original pseudo-network and the modified network.

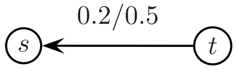
3.2 The Max-Flow Min-Cut Theorem

A flow can be seen as a possibility of transporting a certain amount from s to t. So a flow is the more valuable the more it transports from s to t. This motivates the next definition:

Definition 3.3 (Value of a Flow). The *value* of a flow φ in a network $N = (\alpha : X \leftrightarrow X, s, t)$ is defined as $val(\varphi) = |s\varphi| - |\varphi s|$.

The value measures how much s "generates" in addition to its own input $|\varphi s|$. In Prop. 6.1 of [Glü] it has been shown (in a slightly different notation) that the equality $val(\varphi) = |s\varphi| - |\varphi s| = |\varphi t| - |t\varphi|$ holds. Note that the value of a flow can even be a negative number as shown in the following example:

Example 3.4.



In this network α consists of the single edge (t, s) with capacity 0.5, and the depicted flow φ is given by $\varphi(t, s) = 0.2$. So we have $val(\varphi) = |s\varphi| - |\varphi s| = 0 - 0.2 = -0.2$. However, our main aim will be to send flow from s to t. □

Definition 3.5 (Residual Network). For a flow φ on an s-t-network $N = (\alpha : X \leftrightarrow X, s, t)$ we define the *residual network* N_α *of* N *with respect to* α as the pseudo-network $N_\alpha = (\varphi_\alpha : X \leftrightarrow X, s, t)$ with $\varphi_\alpha = (\alpha \ominus \varphi) \sqcup \varphi^\sharp$.

That means, a flow φ over an edge (x, y) with capacity $\alpha(x, y)$ causes two edges in the residual network: one edge (x, y) with capacity $\alpha(x, y) - \varphi(x, y)$ (note that $\varphi \sqsubseteq \alpha$) and one edge (y, x) in the opposite direction with capacity $\varphi(x, y)$. The intuitive meaning is that one can send along the edge (x, y) an additional flow amount of at most $\alpha(x, y) - \varphi(x, y)$ and the flow along the edge (x, y) can be decreased by an amount of at most $\varphi(x, y)$, which corresponds to increasing the flow over the reverse edge (y, x) by exactly the same amount. In this context the requirement $\alpha \sqcap \alpha^\sharp = \mathbf{0}$ is indispensable.

An important concept for reasoning about flows is that of a cut.

Definition 3.6 (Cut). A *cut* in an s-t-network $N = (\alpha : X \leftrightarrow X, s, t)$ is a test relation τ on X with the property $s \sqsubseteq \tau \sqsubseteq t^c$. Thus a cut corresponds to a subset of X containing s but not t, a definition consistent with the usual one. The *capacity* $c(\tau)$ of a cut τ is given by $c(\tau) = |\tau\alpha\tau^c|$. It describes the maximal flow amount which can be sent out from nodes in τ to nodes in τ^c. A cut τ is *saturated* by a flow φ if $val(\varphi) = c(\tau)$.

Intuitively it is obvious that the value of all flows can not exceed the capacity of a cut, because a cut forms a border between the source and the sink. This basic fact is proved both in [Kaw] and [Glü].

Given a network $N = (\alpha : X \leftrightarrow X, s, t)$, our main aim is to determine a flow of maximal value, i.e., a flow φ with the property $val(\varphi) \geq val(\psi)$ for all flows ψ on N. An important criterion for the maximality of a flow is the so-called Max-Flow Min-Cut Theorem:

Theorem 3.7 (Max-Flow Min-Cut Theorem). *Let $N = (\alpha : X \leftrightarrow X, s, t)$ be network and φ a flow on N. Then the following properties are equivalent:*

(a) φ is maximal.
(b) $t \sqcap s\varphi^*_\alpha = \mathbf{0}$, or equivalently $|s\varphi^*_\alpha t| = 0$.
(c) There exists a cut τ that is saturated by φ.

Intuitively part (b) means that it is impossible to send flow from s to t in the residual network of a maximal flow. Part (c) states that the value of a maximal flow equals the capacity of a certain cut. Because the value of a flow is never larger than the capacity of a cut, this cut must be minimal (hence the name Max-Flow Min-Cut Theorem). Such a cut τ is saturated by a maximal flow φ; in this case $\tau\alpha\tau^c = \tau\varphi\tau^c$ holds. This means that every edge of α leading from τ into its complement τ^c carries the maximally possible amount of flow.

4 Networks with Lower Bounds

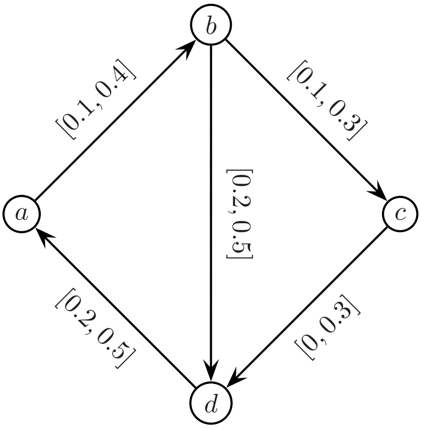
4.1 Definitions and Background

Definition 4.1 (Networks With Lower Bounds). A *pseudo-network with lower bounds* over a node set X is a pair $N = (\alpha : X \leftrightarrow X, \beta : X \leftrightarrow X)$ with $\beta \sqsubseteq \alpha$, where α and β are called the *upper* and *lower bound*. If additionally $\alpha \sqcap \alpha^\sharp = \mathbf{0}$ then N is called a *network with lower bounds*.

Our definition of a (pseudo-)network with lower bounds corresponds to the common one in an analogous way as that of an s-t-(pseudo-)network. The network condition implies $\alpha \sqcap \beta^\sharp = \mathbf{0}$, $\alpha^\sharp \sqcap \beta = \mathbf{0}$ and $\beta \sqcap \beta^\sharp = \mathbf{0}$ (note that in the semiring of fuzzy endorelations on X the zero $\mathbf{0}$ is meet-irreducible).

We represent networks with lower bounds in a graphical way analogous to the common one for graphs. We associate the elements of the basic underlying set with the nodes of a graph and label its edges with $[\beta, \alpha]$ where β and α denote the lower and upper capacity bound of the edge, resp. Edges (x, y) with $\alpha(x, y) = 0$ (and hence $\beta(x, y) = 0$) are omitted in the depiction.

Example 4.2.



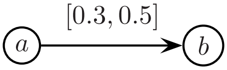
In this example we have $X = \{a, b, c, d\}$. For the upper bound we have, e.g., $\alpha(b, d) = 0.5$ or $\alpha(b, c) = 0.3$ and for the lower bound $\beta(a, b) = 0.1$ or $\beta(d, a) = 0.2$. For pairs $(x, y) \in X \times Y$ without arcs in the depiction we have $\alpha(x, y) = \beta(x, y) = 0$, so for example $\alpha(a, c) = \beta(a, c) = 0$ or $\alpha(a, d) = \beta(a, d) = 0$. □

4.2 Circulations in Networks with Lower Bounds

Definition 4.3 (Circulation). A fuzzy relation γ in a pseudo-network $N = (\alpha : X \leftrightarrow X, \beta : X \leftrightarrow X)$ with lower bounds is called a *circulation* if it is a flow in the network α and satisfies the additional capacity constraint $\beta \sqsubseteq \gamma$.

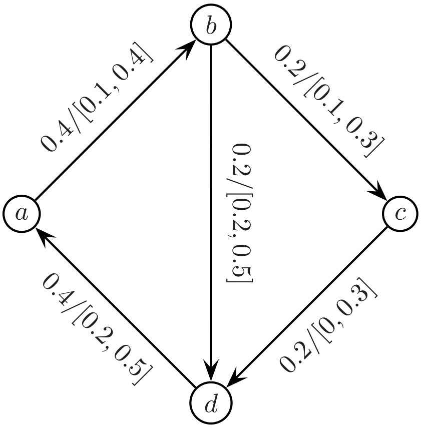
A circulation in a network with lower bounds need not always exist. A simple counterexample is given here:

Example 4.4.



Every fuzzy endorelation γ on X fulfilling the capacity constraint and flow conservation needs to satisfy $0.3 \leq |\gamma b| \leq 0.5$ and $|b\gamma| = 0$ and hence cannot satisfy the requirement $|\gamma b| = |b\gamma|$.

Contrary to this, our network with lower bounds from Example 4.2 admits a circulation as one can see here:



An edge (x, y) carries the information $\gamma(x, y)/[\beta(x, y), \alpha(x, y)]$. □

4.3 Extending Pseudo-networks

In the next section we will need to add nodes to a network. To this end we introduce some notation.

Definition 4.5 (Embedding). For a fuzzy endorelation $\alpha : X \leftrightarrow X$ on X and a superset $\hat{X}$ of X, the *embedding* $\lceil \alpha \rceil$ of α into $\hat{X}$ is $\lceil \alpha \rceil : \hat{X} \leftrightarrow \hat{X}$, given by $\lceil \alpha \rceil(x, y) = \alpha(x, y)$ for all $(x, y) \in X \times X$ and $\lceil \alpha \rceil(x, y) = 0$ otherwise.

This means that the nodes in $\hat{X} \backslash X$ are added as isolated nodes to the graph described by α.

The embedding of fuzzy relations distributes over join, meet, truncating sum, truncating difference and composition, i.e., $\lceil \alpha \circ \beta \rceil = \lceil \alpha \rceil \circ \lceil \beta \rceil$ for $\circ \in \{\sqcup, \sqcap, \oplus, \ominus, \cdot\}$. In particular we have $\lceil \alpha \rceil = \lceil id \rceil \lceil \alpha \rceil = \lceil \alpha \rceil \lceil id \rceil$. Embedding also commutes with converse, i.e., $\lceil \alpha^\sharp \rceil = \lceil \alpha \rceil^\sharp$. Moreover, it is order-preserving: $\alpha \sqsubseteq \beta$ implies $\lceil \alpha \rceil \sqsubseteq \lceil \beta \rceil$. The embedding $\lceil \tau \rceil$ of a test relation τ on X is a test relation on $\hat{X}$. Finally, embedding preserves cardinality: $|\lceil \alpha \rceil| = |\alpha|$.

The dual operation to embedding is projection:

Definition 4.6 (Projection). For a fuzzy endorelation $\alpha : \hat{X} \leftrightarrow \hat{X}$ its *projection* $\lfloor \alpha \rfloor : X \leftrightarrow X$ to a subset $X \subseteq \hat{X}$ is given by $\lfloor \alpha \rfloor(x, y) = \alpha(x, y)$ for all $x, y \in X$.

Projection has algebraic properties similar to the ones of embedding. It is order preserving and commutes with converse and distributes over join, meet, truncating sum and truncating difference, but in general not over composition. The projection $\lfloor \hat{\tau} \rfloor$ of a test relation $\hat{\tau}$ on $\hat{X}$ is a test relation on X. Finally, projection partially preserves cardinality: for $\sigma, \tau \subseteq X$ one has $|\lfloor \sigma \hat{\alpha} \tau \rfloor| = |\sigma \hat{\alpha} \tau|$.

4.4 Existence of Circulations

Let $N = (\alpha : X \leftrightarrow X, \beta : X \leftrightarrow X)$ be a network with lower bounds with a normalised fuzzy relation α as upper bound (recall that this means $|\alpha| \leq 1$). This can be achieved by a suitable scaling of the upper bound. Note that if α is normalised then β is, too. Because of $x\beta \sqsubseteq \beta$ for an arbitrary point relation x we have $|x\beta| \leq 1$ and analogously $|\beta x| \leq 1$.

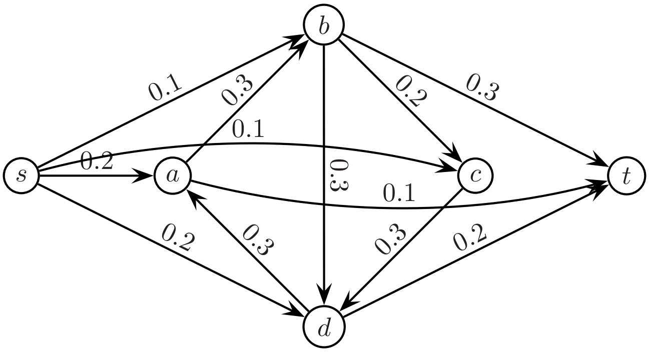
Now our intention is to develop in an algebraic way an algorithm that determines whether a network with lower bounds admits a circulation and, if so, computes one. This will be done using a maximal flow in an s-t-network derived from the original network. The construction we will use is well known in the literature, cf. [Jun], Section 10.2. The advantage of our approach is that we avoid big indexed sums by calculating with test relations and cardinalities.

To this end we define the s-t-pseudo-network $\hat{N} = (\hat{\alpha} : \hat{X} \leftrightarrow \hat{X}, s, t)$ as follows: we choose two new nodes $s, t \notin X$ and set $\hat{X} = X \dot{\cup} \{s\} \dot{\cup} \{t\}$. For the associated point relations s and t on $\hat{X}$ this means $s \sqcap t = \mathbf{0}$ and $(s \sqcup t) \sqcap x = \mathbf{0}$ for all point relations $x \sqsubseteq X$.

As abbreviations for $\nabla_{\hat{X}\hat{X}}, \mathbf{0}_{\hat{X}\hat{X}}$ and $\nabla_{XX}, \mathbf{0}_{XX}$ we use $\hat{\nabla}, \hat{\mathbf{0}}$ and $\nabla, \mathbf{0}$, resp. With this notation the equalities $\lfloor \hat{\nabla} \rfloor = \nabla$ and $\lfloor \hat{\mathbf{0}} \rfloor = \mathbf{0}$ hold.

We define the capacity constraint $\hat{\alpha}$ of $\hat{N}$ by $\hat{\alpha} = \hat{\alpha}_1 \sqcup \hat{\alpha}_2 \sqcup \hat{\alpha}_3$ with three fuzzy endorelations $\hat{\alpha}_1$, $\hat{\alpha}_2$ and $\hat{\alpha}_3$ on $\hat{X}$. First, $\hat{\alpha}_1 = \bigsqcup_{x \in X} |\beta x| s \hat{\nabla} \lceil x \rceil$. Intuitively this means that we have an edge from s to every point x of X having as capacity the sum of the lower bounds of all edges entering x. $\hat{\alpha}_3$ is defined in a similar manner, namely as $\hat{\alpha}_3 = \bigsqcup_{x \in X} |x\beta| \lceil x \rceil \hat{\nabla} t$. This introduces an edge from every point x of X to t having as capacity the sum of the lower bounds of all edges leaving x. Finally, $\hat{\alpha}_2$ is defined as $\hat{\alpha}_2 = \lceil \alpha \ominus \beta \rceil$. This means that every edge in N becomes an edge in $\hat{N}$ having as capacity the difference between its upper and lower bounds (remember that $\beta \sqsubseteq \alpha$). The relation $\hat{\alpha}$ is well defined because all terms of the form $|x\beta|$ and $|\beta x|$ are guaranteed not to be larger than one.

Example 4.7. If we apply this construction to the network with lower bounds shown in Example 4.2 we obtain the following network:



The edge from c to t has not been forgotten, but we have $\hat{\alpha}(c, t) = 0$ and according to our conventions we omit the arc corresponding to this pair. □

Lemma 4.8. *$\hat{N}$ is an s-t-network.*

Proof. We have to show $\hat{\alpha} \sqcap \hat{\alpha}^\sharp = \hat{\mathbf{0}}$. As a preparation we calculate

$$\begin{aligned}
& \hat{\alpha} \sqcap \hat{\alpha}^\sharp \\
= & \quad \{\!\![\text{ definition and rules for converse }]\!\!\} \\
& (\hat{\alpha}_1 \sqcup \hat{\alpha}_2 \sqcup \hat{\alpha}_3) \sqcap (\hat{\alpha}_1^\sharp \sqcup \hat{\alpha}_2^\sharp \sqcup \hat{\alpha}_3^\sharp) \\
= & \quad \{\!\![\text{ distributivity }]\!\!\} \\
& \textstyle\bigsqcup_{i=1}^{3} \bigsqcup_{j=1}^{3} (\hat{\alpha}_i \sqcap \hat{\alpha}_j^\sharp) \ .
\end{aligned}$$

If we succeed in showing that all meets inside the two big joins become $\hat{\mathbf{0}}$ we are done.

For the case $i = j = 1$ we have

$$\begin{aligned}
& \hat{\alpha}_1^\sharp \\
= & \quad \{\!\![\text{ converse distributes over join }]\!\!\} \\
& \textstyle\bigsqcup_{x \sqsubseteq X} |\beta x| \lceil x \rceil^\sharp \hat{\nabla}^\sharp s^\sharp \\
= & \quad \{\!\![\text{ universal and test relations are their own converses }]\!\!\} \\
& \textstyle\bigsqcup_{x \sqsubseteq X} |\beta x| \lceil x \rceil \hat{\nabla} s \\
= & \quad \{\!\![\text{ rules for embedding }]\!\!\} \\
& \textstyle\bigsqcup_{x \sqsubseteq X} |\beta x| \lceil id \rceil \lceil x \rceil \hat{\nabla} s \ .
\end{aligned}$$

Hence

$$\begin{array}{ll}
 & \hat{\alpha}_1 \sqcap \hat{\alpha}_1^\sharp \\
= & \quad \{\![\text{ distributivity }]\!\} \\
 & \bigsqcup_{x,y\in X} |\beta x| s\hat{\nabla}\lceil x\rceil \sqcap |\beta y|\lceil id\rceil\lceil y\rceil\hat{\nabla}s \\
\leq & \quad \{\![\text{ isotony of meet and join }]\!\} \\
 & \bigsqcup_{x,y\in X} s\hat{\nabla}\lceil x\rceil \sqcap \lceil id\rceil\lceil y\rceil\hat{\nabla}s \\
= & \quad \{\![\text{ by } s\lceil id\rceil = \hat{\mathbf{0}} \text{ and Lemma 2.11(a) }]\!\} \\
 & \bigsqcup_{x,y\in X} \hat{\mathbf{0}} \\
= & \quad \{\![\text{ lattice algebra }]\!\} \\
 & \hat{\mathbf{0}}\ .
\end{array}$$

In the case $i = 3$ and $j = 1$ we have to consider $\hat{\alpha}_3 \sqcap \hat{\alpha}_1^\sharp$. Calculations similar to those above lead to $\hat{\alpha}_3 \sqcap \hat{\alpha}_1^\sharp = \bigsqcup_{x,y\in X} |x\beta|\lceil x\rceil\hat{\nabla}t \sqcap |\beta y|\lceil y\rceil\hat{\nabla}s$. Because of $st = \hat{\mathbf{0}}$ we obtain again $\hat{\mathbf{0}}$ by Lemma 2.11(a).

The other cases, except $i = j = 2$, can be treated in a similar manner (note that $\hat{\alpha}_2 = \lceil id\rceil\hat{\alpha}_2 = \hat{\alpha}_2\lceil id\rceil$). For $i = j = 2$ we calculate $\hat{\alpha}_2 \sqcap \hat{\alpha}_2^\sharp = \lceil \alpha\ominus\beta\rceil \sqcap \lceil \alpha\ominus\beta\rceil^\sharp \sqsubseteq \lceil\alpha\rceil \sqcap \lceil\alpha\rceil^\sharp = \lceil \alpha\sqcap\alpha^\sharp\rceil$, and this equals $\hat{\mathbf{0}}$ by the requirement that N be a network with lower bounds. □

Lemma 4.9. *A maximal flow on $\hat{N}$ with value $|\beta|$ saturates the cuts s and t^c.*

Proof. First we show that the capacities of the cuts s and t^c in $\hat{N}$ both equal $|\beta|$. We have

$$\begin{array}{ll}
 & c(s) \\
= & \quad \{\![\text{ definition of capacity }]\!\} \\
 & |s\hat{\alpha}s^c| \\
= & \quad \{\![\text{ definition of } \hat{\alpha} \text{ }]\!\} \\
 & |s(\hat{\alpha}_1 \sqcup \hat{\alpha}_2 \sqcup \hat{\alpha}_3)s^c| \\
= & \quad \{\![\text{ analogously to previous proof }]\!\} \\
 & |s\hat{\alpha}_1| \\
= & \quad \{\![\text{ definition of } \hat{\alpha}_1 \text{ }]\!\} \\
 & |s \bigsqcup_{x\in X} |\beta x| s\hat{\nabla}\lceil x\rceil| \\
= & \quad \{\![\text{ idempotence of test multiplication, distributivity and} \\
 & \quad\quad \text{disjointness of the } x\text{-indices in the big join }]\!\} \\
 & \Sigma_{x\in X}|\,|\beta x| s\hat{\nabla}\lceil x\rceil\,| \\
= & \quad \{\![\text{ by } |s\hat{\nabla}\lceil x\rceil| = 1 \text{ }]\!\} \\
 & \Sigma_{x\in X}|\beta x|.
\end{array}$$

Rewriting the join in the opposite direction we obtain $\Sigma_{x\in X}|\beta x| = |\dot{\bigsqcup}_{x\in X}\beta x| = |\beta id_X| = |\beta|$.

In the same manner one can show $c(t^c) = |\beta|$. According to the assertions of the Max-Flow Min-Cut Theorem a maximal flow with value $|\beta|$ has to saturate both cuts. □

Theorem 4.10. *There is a circulation on N iff the value of a maximal flow on $\hat{N}$ is exactly $|\beta|$.*

Proof. ($\Leftarrow$) Let $\hat{\varphi}$ be a flow on $\hat{N}$ with value $val(\hat{\varphi}) = |\beta|$. Then we define a fuzzy relation $\gamma : X \leftrightarrow X$ by $\gamma = \lfloor \hat{\varphi} \rfloor \oplus \beta$. Because $\hat{\varphi}$ is a flow on $\hat{N}$ it has to respect the capacity constraint, which implies $\gamma \sqsubseteq \lfloor \hat{\alpha} \rfloor = \hat{\alpha} \ominus \beta$ and hence $\gamma \sqsubseteq \hat{\alpha}$. $\beta \sqsubseteq \gamma$ is clear because of the definition of γ. If we succeed in showing that γ satisfies flow conservation on N we are done.

Consider an arbitrary test relation $\tau : X \leftrightarrow X$. Because of the flow properties of $\hat{\varphi}$ we have

$$|\lceil \tau \rceil \hat{\varphi}| = |\hat{\varphi} \lceil \tau \rceil| \ .$$

We introduce a test relation X on $\hat{X}$ by $X = \lceil id_X \rceil = (s \sqcup t)^c$. Note that $id_{\hat{X}} = X \sqcup s \sqcup t$.

Now we reason as follows:

$$\begin{aligned}
& |\lceil \tau \rceil \hat{\varphi}| = |\hat{\varphi} \lceil \tau \rceil| \\
\Leftrightarrow \quad & \{\![\ \text{identity}\ ]\!\} \\
& |\lceil \tau \rceil \hat{\varphi}\, id_{\hat{X}}| = |id_{\hat{X}} \hat{\varphi} \lceil \tau \rceil| \\
\Leftrightarrow \quad & \{\![\ \text{above remark}\ ]\!\} \\
& |\lceil \tau \rceil \hat{\varphi}(X \sqcup s \sqcup t)| = |(X \sqcup s \sqcup t) \hat{\varphi} \lceil \tau \rceil| \\
\Leftrightarrow \quad & \{\![\ X,\ s,\ t \text{ disjoint and Lemma 2.11(b)}\ ]\!\} \\
& |\lceil \tau \rceil \hat{\varphi} X| + |\lceil \tau \rceil \hat{\varphi} s| + |\lceil \tau \rceil \hat{\varphi} t| = |X \hat{\varphi} \lceil \tau \rceil| + |s \hat{\varphi} \lceil \tau \rceil| + |t \hat{\varphi} \lceil \tau \rceil| \\
\Rightarrow \quad & \{\![\ \hat{\alpha} s = \hat{\mathbf{0}} \wedge \hat{\varphi} \sqsubseteq \hat{\alpha} \Rightarrow \hat{\varphi} s = \hat{\mathbf{0}}, \text{ analogously } t\hat{\varphi} = \hat{\mathbf{0}}\ ]\!\} \\
& |\lceil \tau \rceil \hat{\varphi} X| + |\lceil \tau \rceil \hat{\varphi} t| = |X \hat{\varphi} \lceil \tau \rceil| + |s \hat{\varphi} \lceil \tau \rceil| \\
\Rightarrow \quad & \{\![\ \hat{\varphi} \text{ maximal with value } |\beta|, \text{ Lemma 4.9}\ ]\!\} \\
& |\lceil \tau \rceil \hat{\varphi} X| + |\lceil \tau \rceil \hat{\alpha} t| = |X \hat{\varphi} \lceil \tau \rceil| + |s \hat{\alpha} \lceil \tau \rceil| \\
\Rightarrow \quad & \{\![\ \text{definition of } \hat{\alpha}\ ]\!\} \\
& |\lceil \tau \rceil \hat{\varphi} X| + |\tau \beta| = |X \hat{\varphi} \lceil \tau \rceil| + |\beta \tau| \\
\Rightarrow \quad & \{\![\ \text{definition of } X\ ]\!\} \\
& |\tau \lfloor \hat{\varphi} \rfloor| + |\tau \beta| = |\lfloor \hat{\varphi} \rfloor \tau| + |\beta \tau| \\
\Rightarrow \quad & \{\![\ \lfloor \hat{\varphi} \rfloor \sqsubseteq \lfloor \hat{\alpha} \rfloor = \alpha \ominus \beta, \text{ rules of cardinality}\ ]\!\} \\
& |\tau(\lfloor \hat{\varphi} \rfloor \oplus \beta)| = |(\lfloor \hat{\varphi} \rfloor \oplus \beta)\tau| \ .
\end{aligned}$$

($\Rightarrow$) Let γ be a circulation on N. Then we construct a fuzzy relation $\hat{\varphi} : \hat{X} \leftrightarrow \hat{X}$ by $\hat{\varphi} = s\hat{\alpha} \sqcup \lceil \gamma \ominus \beta \rceil \sqcup \hat{\alpha} t$. Because γ satisfies the capacity constraint and because of the construction of $\hat{\varphi}$ the capacity constraint on $\hat{N}$ is satisfied by $\hat{\varphi}$. The value of the thus constructed flow is $|s\hat{\alpha}| = |\beta|$, and it is maximal because it saturates the cut s. Flow conservation can be shown similarly as above. □

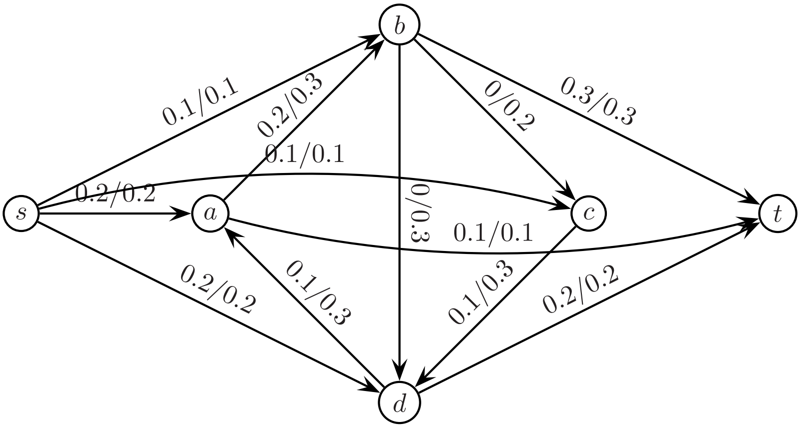
4.5 Algorithmic Aspects

We wouldn't be computer scientists if we didn't give an algorithm for computing a circulation in a network with lower bounds. The construction from the proof of Theorem 4.10 can immediately be used to construct from an algorithm for the computation of a maximal flow in an s-t-network an algorithm for determining a circulation in a network with lower bounds:

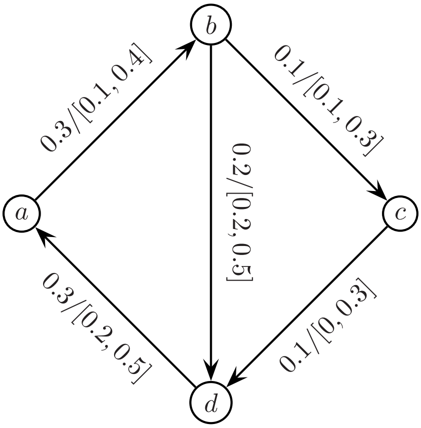
(1) Determine the s-t-network $\hat{N}$ as described above.
(2) Determine a maximal flow φ on $\hat{N}$.
(3) If $val(\varphi) \neq \beta$ then no circulation in N is possible; otherwise return γ as described above.

An algorithm for computing a maximal flow in an s-t-network based on fuzzy relations has been derived in [Kaw].

Now we try out the above algorithm on our running example. E.g. by Kawahara's method one can find the following flow on the induced s-t-network:



This flow is indeed maximal, because it saturates the cut s. Applying our algorithm we obtain the following circulation:



This circulation is different from the one already shown in the second picture of Example 4.4.

4.6 Another Existence Criterion

We will now derive a more elegant criterion for the existence of a flow in a network with lower bounds. We notice that there is a bijective mapping between tests on N and cuts on $\hat{N}$: for a test relation τ on N we choose the cut $s \sqcup \lceil\tau\rceil$ as corresponding cut on $\hat{N}$; the converse direction is obvious.

For the capacity of such a cut $s \sqcup \lceil\tau\rceil$ we can calculate as follows (note that by construction $s^c = X \sqcup t$, $t^c = s \sqcup X$, $X^c = s \sqcup t$ and $\lceil\tau\rceil^c = \lceil\tau^c\rceil \sqcup s \sqcup t$ hold):

$$
\begin{array}{ll}
 & c(s \sqcup \lceil\tau\rceil) \\
= & \quad \{\![\ \text{definition of capacity}\ ]\!\} \\
 & |(s \sqcup \lceil\tau\rceil)\hat{\alpha}(s \sqcup \lceil\tau\rceil)^c| \\
= & \quad \{\![\ \text{above remark}\ ]\!\} \\
 & |(s \sqcup \lceil\tau\rceil)\hat{\alpha}(\lceil\tau^c\rceil \sqcup t)| \\
= & \quad \{\![\ \text{distributivity of composition over join}\ ]\!\} \\
 & |s\hat{\alpha}\lceil\tau^c\rceil \sqcup \lceil\tau\rceil\hat{\alpha}\lceil\tau^c\rceil \sqcup s\hat{\alpha}t \sqcup \lceil\tau\rceil\hat{\alpha}t \\
= & \quad \{\![\ \text{disjointness and Lemma 2.11(b)}\ ]\!\} \\
 & |s\hat{\alpha}\lceil\tau^c\rceil| + |\lceil\tau\rceil\hat{\alpha}\lceil\tau^c\rceil| + |s\hat{\alpha}t| + |\lceil\tau\rceil\hat{\alpha}t| \\
= & \quad \{\![\ s\hat{\alpha}t = \hat{\mathbf{0}}\ \text{by construction}\ ]\!\} \\
 & |s\hat{\alpha}\lceil\tau^c\rceil| + |\lceil\tau\rceil\hat{\alpha}\lceil\tau^c\rceil| + |\lceil\tau\rceil\hat{\alpha}t| \\
= & \quad \{\![\ \text{definition of } \hat{\alpha}\ ]\!\} \\
 & |\beta\tau^c| + |\tau\beta| + |\tau(\alpha \ominus \beta)\tau^c| \\
= & \quad \{\![\ \text{distributivity of test multiplication over } \ominus\ ]\!\} \\
 & |\beta\tau^c| + |\tau\beta| + |\tau\alpha\tau^c \ominus \tau\beta\tau^c| \\
= & \quad \{\![\ \text{properties of cardinality, } \tau\beta\tau^c \sqsubseteq \tau\alpha\tau^c\ ]\!\} \\
 & |\beta\tau^c| + |\tau\beta| + |\tau\alpha\tau^c| - |\tau\beta\tau^c| \\
= & \quad \{\![\ \text{for all } \xi \text{ there are the decompositions } \xi = \tau\xi \mathbin{\dot\sqcup} \tau^c\xi = \xi\tau \mathbin{\dot\sqcup} \xi\tau^c\ ]\!\} \\
 & |\tau\beta\tau^c| + |\tau^c\beta\tau^c| + |\tau\beta\tau| + |\tau\beta\tau^c| + |\tau\alpha\tau^c| - |\tau\beta\tau^c| \\
= & \quad \{\![\ \text{arithmetic and rearrangement}\ ]\!\} \\
 & |\tau\alpha\tau^c| + |\tau\beta\tau| + |\tau\beta\tau^c| + |\tau^c\beta\tau^c| \\
= & \quad \{\![\ \beta = \tau\beta\tau \mathbin{\dot\sqcup} \tau\beta\tau^c \mathbin{\dot\sqcup} \tau^c\beta\tau \mathbin{\dot\sqcup} \tau^c\beta\tau^c\ ]\!\} \\
 & |\tau\alpha\tau^c| - |\tau^c\beta\tau| + |\beta|\ .
\end{array}
$$

By the previous theorem N admits a circulation iff the value of a maximal flow on $\hat{N}$ equals $|\beta|$, so according to the Max-Flow Min-Cut Theorem $|\tau\alpha\tau^c| - |\tau^c\beta\tau|$ has to be ≥ 0 for all test relations τ on N iff N admits a circulation. So we have proved the following theorem:

Theorem 4.11. *A network with lower bounds $N = (\alpha : X \leftrightarrow X, \beta : X \leftrightarrow X)$ admits a circulation iff the condition $|\tau\alpha\tau^c| \geq |\tau^c\beta\tau|$ holds for all test relations τ on X.*

This theorem is not very suitable for application in practice (the number of test relations is growing exponentially in the number of nodes!), but it can serve well in the theoretical investigation of networks with lower bounds.

5 Conclusion and Future Work

We have applied Kawahara's methods developed for flows in networks successfully to circulations in networks with lower bounds. As shown in [GlüI], it is not difficult to do the same for circulations in networks with additional imports as described in [Jun], Chapter 7. There a node can receive an additional amount of flow by the environment or flow can disappear from a node to the outside.

In all cases the basic mathematical tools are fuzzy relations and their cardinality. As we detail in the Appendix, these form a semiring (and even a Kleene algebra) with tests. Since automated reasoning about Kleene algebras is on the rise (cf. [HS]), we expect the first automatically proved theorems in this area soon.

Another challenge in this area will be the description of min-cost flows (cf. [Jun], Chapter 10) with fuzzy relational methods. This will be a little bit more difficult, because in the problem of a min-cost flow the amount of flow is multiplied by a cost function on the edge it crosses. For this purpose we have first to define an operation for the edgewise multiplication of fuzzy relations and to explore its algebraic laws and its behavior in connection with the cardinality.

Acknowledgment. We are grateful to Peter Höfner, Yasuo Kawahara, Walter Vogler and the anonymous referees for helpful comments and suggestions.

References

[AMO] Ahuja, R., Magnanti, T., Orlin, J.: Network Flows. Prentice-Hall, Englewood Cliffs (1993)

[Glü] Glück, R.: Network Flows, Semirings and Fuzzy Relations. Institut für Informatik, Universität Augsburg, Tech. Rep, -01 (2008), `http://www.opus-bayern.de/uni-augsburg/volltexte/2008/726/`

[GlüI] Glück, R.: Import Networks, Fuzzy Relations and Semirings. In: Berghammer, R., Möller, B., Struth, G. (eds.) Relations and Kleene Algebra in Computer Science — PhD Programme Proceedings, RelMiCS10/AKA5, Frauenwörth, Germany, April 7 – April 11, 2008. Institut für Informatik, Universität Augsburg, Technical Report 2008-04, pp. 58–62 (2008)

[GTT] Goldberg, A., Tardos, E., Tarjan, R.: Network Flow Algorithms. In: Korte, B., Lovasz, L., Prömel, H., Schrijver, A. (eds.) Algorithms and Combinatorics. Paths, Rows, and VLSI-Layout, vol. 9, pp. 101–164. Springer, Heidelberg (1990)

[HS] Höfner, P., Struth, G.: Automated Reasoning in Kleene Algebra. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 279–294. Springer, Heidelberg (2007)

[Jun] Jungnickel, D.: Graphs, Networks and Algorithms, 2nd edn. Springer, Heidelberg (2005)

[Kaw] Kawahara, Y.: On the Cardinality of Relations. In: Schmidt, R.A. (ed.) RelMiCS/AKA 2006. LNCS, vol. 4136, pp. 251–265. Springer, Heidelberg (2006)

[KozKA] Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Inf. Comput. 110(2), 366–390 (1994)

[KozT] Kozen, D.: Kleene algebra with tests. ACM Transactions on Programming Languages and Systems 19(3), 427–443 (1997)

[MB] Manes, E., Benson, D.: The Inverse Semigroup of a Sum-Ordered Semiring. Semigroup Forum 31, 129–152 (1985)

A Appendix: Semirings and Kleene Algebras

In this section we embed fuzzy relations into the more general mathematical framework of semirings and Kleene algebras. This allows re-use of many results from there and also automatic proofs of some relevant properties.

A.1 Introduction

Definition A.1 (Semiring). A *semiring* is a quintuple $(M, +, \cdot, 0, 1)$ fulfilling the following properties:

- $(M, +, 0)$ is a commutative monoid.
- $(M, \cdot, 1)$ is a monoid.
- 0 is an annihilator with respect to $\cdot$, i.e.,

$$\forall m \in M : m \cdot 0 = 0 \cdot m = 0$$

- $\cdot$ distributes over $+$, i.e.,

$$\forall a, b, c \in M : a \cdot (b + c) = (a \cdot b) + (a \cdot c),\ (a + b) \cdot c = (a \cdot c) + (b \cdot c)$$

We call $+$ and $\cdot$ *addition* and *multiplication*, resp.

Elements of a semiring can, e.g., be considered as modelling some sort of transition relation, such as the transition from one graph node to another along an edge. An element $x + y$ corresponds to a choice between x and y, while $x \cdot y$ corresponds to the sequential composition of x and y in that order.

As one can see the fuzzy endorelations over a set X form a semiring with join as addition, composition as multiplication, $\mathbf{0}_{XX}$ as zero and id_X as one. Other examples for semirings are $(\mathbb{N}, +, \cdot, 0, 1)$ or $(\mathcal{P}(X), \cup, \cap, \emptyset, X)$ for an arbitrary fixed set X.

Definition A.2 (Idempotent semiring). *If $x+x = x$ holds for all elements x of a semiring the semiring is called idempotent. On such an idempotent semiring a partial order $\sqsubseteq$ is defined by $x \sqsubseteq y \Leftrightarrow x + y = y$.*

The order $x \sqsubseteq y$ means that the choices offered by x are contained in the ones offered by y. The operations $\cdot$ and $+$ are $\sqsubseteq$-isotone in both arguments; moreover 0 is the least element w.r.t. $\sqsubseteq$.

The fuzzy relations over a set X form an idempotent semiring and the ordering introduced in Definition 2.4 coincides with the semiring-theoretical one.

A.2 Tests in Semirings

An important special subset of elements in an idempotent semiring are the so-called *tests*.

Definition A.3 (Test). An element p of an idempotent semiring with natural order $\sqsubseteq$ is called a *test* if there is an element $\neg p$, called the *complement* of p, with the properties $p + \neg p = 1$ and $p \cdot \neg p = 0 = \neg p \cdot p$.

This definition is basically due to [MB]; a slightly more liberal definition of tests is given in [KozT]. The definition implies that $p \sqsubseteq 1$ holds for all tests p and that the complement of an element is unique.

Tests are the algebraic counterparts of assertions in programs or predicates characterising subsets of nodes in graphs. Addition and multiplication of tests correspond to their disjunction and conjunction, resp., while $\neg$ corresponds to logical negation. The definition implies that 0 and 1 are tests; they correspond to the everywhere false and to the everywhere true predicate, resp.

Consider a general semiring element x and a test p. Then, by isotony of multiplication, $px \sqsubseteq 1x = x$. So px is a part of x; it omits all transitions of x that do not start in a point satisfying p. This models enforcing the precondition p. Similarly, xp removes all transitions of x that do not end in a point satisfying p; it enforces p as a postcondition.

This implies the following important property of tests:

Lemma A.4. *Let p, q be tests and a, b arbitrary elements in a semiring such that $pq = 0$. If $z \sqsubseteq pa$ and $z \sqsubseteq qb$ then $z = 0$.*

This means that elements with disjoint sets of starting points (remember that the product of tests is their conjunction) are disjoint as well.

The tests in the semiring $Rel(X, X)$ are precisely the test relations from Definition 2.10.

A.3 Kleene Algebras

Now we add the concept of finite iteration.

Definition A.5 (Kleene Algebra). A *Kleene algebra* [KozKA] is a structure $(S, +, 0, \cdot, 1, ^*)$ such that the reduct $(S, +, 0, \cdot, 1)$ is a semiring and the finite iteration operator * satisfies the unfold and induction axioms

$$1 + xx^* \leq x^* \ , \qquad 1 + x^*x \leq x^*$$

and the *star induction* axioms

$$y + xz \leq z \Rightarrow x^*y \leq z \ , \qquad y + zx \leq z \Rightarrow yx^* \leq z \ .$$

The semiring $Rel(X, X)$ forms a Kleene algebra under the operation * defined in Section 2.2.

Asynchronous Exceptions as an Effect*

William L. Harrison[1], Gerard Allwein[2], Andy Gill[3], and Adam Procter[1]

[1] Dept. of Computer Science, University of Missouri, Columbia, Missouri, U.S.A
[2] Naval Research Laboratory, Code 5543, Washington, DC 20375, U.S.A
[3] Galois, Inc., Beaverton OR 97005, U.S.A

Abstract. Asynchronous interrupts abound in computing systems, yet they remain a thorny concept for both programming and verification practice. The ubiquity of interrupts underscores the importance of developing programming models to aid the development and verification of interrupt-driven programs. The research reported here recognizes asynchronous interrupts as a computational effect and encapsulates them as a building block in modular monadic semantics. The resulting modular semantic model can serve as both a guide for functional programming with interrupts and as a formal basis for reasoning about interrupt-driven computation as well.

1 Introduction

The asynchronous interrupt, according to Dijkstra [4], was a great invention:

> *...but also a Box of Pandora. Because the exact moments of the interrupts were unpredictable and outside our control, the interrupt mechanism turned the computer into a nondeterministic machine with a nonreproducible behavior, and could we control such a beast?*

The construction and verification of programs in the presence of asynchronous exceptions is notoriously difficult. Certainly, one cause of this difficulty is that a setting with interrupts is necessarily concurrent, there being, at a minimum, distinct interrupting and interrupted entities. But, more fundamentally, the notion of computation for asynchronous interrupt is inherently non-deterministic and controlling the non-determinism "beast" remains a challenge.

What do we mean by an asynchronous interrupt? It is a form of exception thrown by the environment external to a thread. A synchronous exception, by contrast, arises from an event internal to a thread's execution; for example, they can originate from run-time errors (e.g., division-by-zero or pattern match failure) and OS system calls. Asynchronous exceptions arise from events external to a thread and may occur at any point between the atomic actions that comprise the thread. Examples of asynchronous exceptions are hardware signals arising non-deterministically from I/O devices as well as the language feature of the same name in Concurrent Haskell [17]. The terms *interrupt* and *asynchronous exception* are used interchangeably throughout this article.

* This research was supported in part by the Gilliom Cyber Security Gift Fund.

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 153–176, 2008.

Is there an modular model that can serve as both a guide for programming with exceptions and a formal semantics for reasoning about them as well? In the past, language semanticists and functional programmers have both turned to monads and monadic semantics when confronted with non-functional effects (e.g., state, exceptions, etc.). This paper argues that one way to understand asynchronous exceptions is as a computational effect encapsulated by a monadic construction. This paper explores precisely this path by decomposing the asynchronous exception effect into monadic components. Surprisingly enough, although all of the monadic components applied here are well-known [19,22], this work is apparently the first to put them together to model interrupts.

This paper argues that monadic semantics [19] (and, particularly, modular monadic semantics (MMS) [16]) is an appropriate foundation for a modular model of asynchronous exceptions. With the monadic model of asynchronous exceptions (MMAE), one may have one's cake and eat it, too: formal specifications in monadic style are easily rendered as executable functional programs. The contributions of this paper are:

- A semantic building block for asynchronous behaviors (Section 3). This building block follows the same lines as other, well-known building blocks in MMS: by a straightforward extension of the underlying monad, we may define a set of operators that, in turn, may be used to define asynchronous interrupts. In the lingo of MMS, asynchronicity becomes a "building block" that may be added to other monadic specifications.
- A denotational framework for modeling a member of the infamous "Awkward Squad" [23]. These are behaviors considered difficult to accommodate within a pure, functional setting. This framework is applied in Section 4 and proved equivalent to a recently published natural semantics of asynchronous exceptions.
- An extension of published formal models of OS kernels with asynchronous behavior (Section 5). We apply the interrupts building block to monadic kernels [9], thereby extending the expressiveness of these models of concurrent systems.

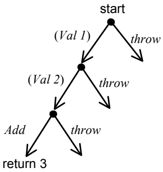
As we use it, the term *interrupt* should not be limited to the hardware mechanism by which a CPU communicates with the external world. Hardware interrupts are a special form of asynchronous exception; they can arise at any time and result in a temporary transfer of control to an interrupt service routine. Asynchronous exceptions do not, in general, involve such a temporary control transfer. The case study we present concerns a hardware interrupt mechanism, although the model we use applies equally well for asynchronous exceptions in general. In particular, the current model has been applied in another significant case study in the semantic specification of the typed interrupt calculus of Palsberg [2,21]. This specification will be published in a sequel.

1.1 Summary of the MMAE

This section summarizes two applications of the monadic model of asynchronous exceptions (MMAE)—one in denotational semantics and the other in functional programming—and will provide the reader with helpful intuitions about the MMAE. Asynchronous exceptions according to this model are really a composite effect, combining non-determinism, concurrency, and some notion of interactivity. Each of these effects corresponds to well-known monads, and the MMAE encapsulates them all within a single monad. Haskell renderings of all the examples in this paper are available online [8].

Non-determinism is inherent in asynchronous exceptions. The reason is simple: when (or if) a computation is interrupted is not determined by the computation itself and so exceptions are, therefore, by definition non-deterministic with respect to the computation. That a notion of concurrency is necessary is less obvious. Computation here is assumed to be "interruptible" only at certain identified points within its execution. These breakpoints (implicitly) divide the computation up into un-interruptible units or atoms. This is tantamount to a theory of concurrency. Finally, a notion of interactivity is required for obvious reasons: the "interruptee" and "interrupter" interact according to some regime.

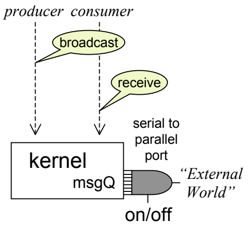
In Section 4, we consider Hutton and Wright's language of arithmetic expressions with synchronous and asynchronous exceptions [13]. This language is defined in detail there, but, for the time being, consider the simple arithmetic expression, $(Add\ (Val\ 1)\ (Val\ 2))$. In Hutton and Wright's operational semantics, there are four possible evaluations of this expression, assuming that exceptions are "on". These evaluations may be described as a tree (see inset) in which each breakpoint "•" represents a place at which the arithmetic computation may be interrupted. The evaluation which returns 3 proceeds along the leftmost path. An asynchronous exception may occur at any of the breakpoints, corresponding to the three evaluations ending in *throw*. In the MMAE, the semantics of $(Add\ (Val\ 1)\ (Val\ 2))$ and its interaction with exceptions is represented in closed form as a single term in a monadic algebra of effects:



$$merge\{merge\{merge\{\eta\, 3, throw\}, throw\}, throw\} \tag{1}$$

The intuitive meaning of this term is clear from it correspondence to the inset tree: the *merge* operator combines two "possible histories", rather like a branch constructor in a tree data type; the *throw* "leaf" is the result of an asynchronous exception; the monadic unit "$(\eta\, 3)$" leaf computation returns the final value.

Figure 1 gives an example application of the MMAE. Recent research has demonstrated how kernels with a broad range of OS behaviors (e.g., message-passing, synchronization, forking, etc.) may be formulated in terms of resumption monads [9] and generalized with monad transformers. The application presented in Section 5 describes how such kernels may be extended to include asynchronous exception behaviors as well. This extension manifests itself in Figure 1 in the creation of the monads K and R underlying the kernel specification. For the kernel without the interrupt-driven port (Figure 1, middle), the monad transformers constructing K and R (i.e., StateT and ResT) are applied to the identity monad, Id; in the kernel with the interrupt-driven port (Figure 1, middle), they are



No Port

$\mathsf{K} = \mathit{StateT\ Sys}\ (\mathit{StateT\ Sto}\ \mathsf{Id})$
$\mathsf{R} = \mathit{ResT}\ \mathsf{K}$

```
Haskell> producer_consumer
broadcasting  1001
broadcasting  1002
        receiving  1001
broadcasting  1003
        receiving  1002 ...
```

With Port & Interrupts on

$\mathsf{K} = \mathit{StateT\ Sys}\ (\mathit{StateT\ Sto}\ \mathsf{N})$
$\mathsf{R} = \mathit{ResT}\ \mathsf{K}$

```
Haskell> producer_consumer
broadcasting  1001
   new datagram: 179
broadcasting  1002
         receiving  179
   new datagram: 204 ...
```

Fig. 1. Kernels with interrupt-driven input port. The kernel design (left) supports an interrupt-driven serial-to-parallel input port and synchronous message-passing primitives. The FIFO message queue, msgQ, stores messages in flight between threads. The input port receives bits non-deterministically from the external world, buffers them, and enqueues a one byte message on msgQ when possible. The baseline kernel (middle) has no such input port; the modified asynchronous kernel (right) extends the baseline kernel with non-determinism, thereby supporting the interrupt-driven port. When a producer-consumer application is run on both kernels, the datagrams received through the port are manifest. See text for further description.

applied instead to the non-determinism monad, N. The addition of asynchronous behaviors requires little more than this refinement of the monad underlying the kernel specification. What precisely all this means and why is the subject of this paper.

2 Background on Monads and Monad Transformers

This section outlines the background material necessary to understand the present work. We must assume of necessity that the reader is familiar with monads. Readers requiring more background should consult the related work (especially, Liang et al. [16]); other readers may skip this section. Section 2.1 contains an overview of the non-determinism monad.

A structure $(\mathsf{M}, \eta, \star)$ is a *monad* if, and only if, M is a type constructor (functor) with associated operations *bind* $(\star : \mathsf{M}\,a \to (a \to \mathsf{M}\,b) \to \mathsf{M}\,b)$ and *unit* $(\eta : a \to \mathsf{M}\,a)$ obeying the well-known "monad laws" [15].

$$\begin{array}{lll} \text{(left-unit)} & (\eta\ v) \star k & = k\ v \\ \text{(right-unit)} & x \star \eta & = x \\ \text{(assoc)} & x \star (\lambda\, v.(k\ v\ \star\ h)) & = (x \star k) \star h \end{array}$$

Two such monads are the identity and state monads:

$$\begin{array}{llll} \mathsf{Id}a & = a & \mathsf{S}\,a & = s \to a \times s \\ \eta_{\mathsf{Id}} v & = v & \eta_{\mathsf{S}} v & = \lambda\sigma.(v, \sigma) \\ x \star_{\mathsf{Id}} f & = f x & \varphi \star_{\mathsf{S}} f & = \lambda\sigma_0.\text{let } (v, \sigma_1) = \varphi\sigma_0 \text{ in } f v \sigma_1 \end{array}$$

Here, s is a fixed type argument, which can be replaced by any type which is to be "threaded" through the computation. What makes monads interesting is not their bind and unit (these are merely computational glue) but, rather, the operations one can define in terms of the extra computational "stuff" they encapsulate. For example, one may define read and write operations, g_S and u_S, to manipulate the underlying state s:

$$\begin{array}{ll}\mathsf{g}_\mathsf{S} \;:\; \mathsf{S}s & \mathsf{u}_\mathsf{S} \;\;\;:\; (s \to s) \to \mathsf{S}()\\ \mathsf{g}_\mathsf{S} = \lambda\sigma.(\sigma,\sigma) & \mathsf{u}_\mathsf{S} f = \lambda\sigma.((),f\sigma)\end{array}$$

The morphisms, g_S and u_S, are pronounced *get* and *update*, respectively. Additional operations on a monad (e.g., u_S on S) are referred to as *non-proper morphisms* to distinguish them from the monadic bind and unit.

Given two monads, M and M', it is natural to ask if their composition, $\mathsf{M} \circ \mathsf{M}'$, is also a monad, but it is well-known that monads generally do not compose in this simple manner [6]. However, *monad transformers* do provide a form of monad composition [6,16,18]. When applied to a monad M, a monad transformer T creates a new monad M'. The monad ($\mathsf{StateT}\,s\,\mathsf{Id}$) is identical to the state monad S. The state monad transformer, ($\mathsf{StateT}\,s$), is shown below.

$$\begin{array}{llll}\mathsf{S}a & = \mathsf{StateT}\,s\,\mathsf{M}\,a = s \to \mathsf{M}(a \times s) & \mathsf{u}f & = \lambda\sigma.\,\eta_\mathsf{M}((),f\,\sigma)\\ \eta_\mathsf{S}\,x & = \lambda\sigma.\,\eta_\mathsf{M}(x,\sigma) & \mathsf{g} & = \lambda\sigma.\,\eta_\mathsf{M}(\sigma,\sigma)\\ x \star_\mathsf{S} f & = \lambda\sigma_0.\,(x\,\sigma_0) \star_\mathsf{M} (\lambda(a,\sigma_1).f\,a\,\sigma_1) & \mathit{lift}_\mathsf{S}\,x & = \lambda\sigma.\,x \star_\mathsf{M} \lambda y.\,\eta_\mathsf{M}(y,\sigma)\end{array}$$

In a composed or *layered* monad, $\mathsf{M}' = \mathsf{T}\mathsf{M}$, any non-proper morphisms on M must be redefined or *lifted* to monad M'. Lifting of non-proper morphisms is frequently performed in terms of a *lift* operator (e.g., lift_S above).

2.1 Non-determinism as a Monad

Semantically, non-deterministic programs (i.e., those with more than one possible value) may be viewed as returning sets of values rather than just one value [1,27]. Consider, for example, the *amb* operator of McCarthy (1963). Given two arguments, it returns either one or the other; for example, the value of $1\ \mathit{amb}\ 2$ is either 1 or 2. The *amb* operator is *angelic*: in the case of the non-termination of one of its arguments, *amb* returns the terminating argument. For the purposes of this exposition, however, we ignore this technicality. According to this view, the meaning of $(1\ \mathit{amb}\ 2)$ is simply the set $\{1,2\}$ and the meaning of $(\mathbf{let}\ x = (1\ \mathit{amb}\ 2)\ ;\ y = (1\ \mathit{amb}\ 2)\ \mathbf{in}\ x + y)$ is $\{2,3,4\}$. The computational theory of non-determinism employed here is based on a monadic form of the powerdomain construction [26,28]. Encoding non-determinism as sets of values is expressed monadically via the finite powerset monad:

$$\begin{array}{ll}\eta \;\;\;:\; a \to \mathcal{P}_{\mathrm{fin}}(a) & \star \;:\; \mathcal{P}_{\mathrm{fin}}(a) \to (a \to \mathcal{P}_{\mathrm{fin}}(b)) \to \mathcal{P}_{\mathrm{fin}}(b)\\ \eta\,x = \{\,x\,\} & S \star f = \bigcup(f\ S)\end{array}$$

where $f\ S = \{ f\ x \mid x \in S \}$ and $\mathcal{P}_{\text{fin}}(-)$ is set of finite subsets drawn from its argument. In the finite set monad, the meaning of $(e\ amb\ e')$ is the union of the meanings of e and e'.

That lists are similar structures to sets is familiar to any functional programmer; a classic exercise in introductory functional programming courses represents sets as lists and set operations as functions on lists (in particular, casting set union ($\cup$) as list append ($+\!\!+$)). Some authors [6,15,31] have made use of the "sets as lists" pun to implement non-deterministic programs within functional programming languages via the list monad; this approach seems to have originated with Hughes and O'Donnell [12]. The list monad (written "[]" in Haskell) is defined by the instance declaration:

```
instance Monad [] where
  return x          = [x]
  (x : xs) >>= f    = f x ++ (xs >>= f)
  [] >>= f          = []
```

This straightforward implementation suffices for our purposes, but it is known to contain an inaccuracy when the lists involved are infinite [30]. Specifically, because $l +\!\!+ k = l$ if the list l is infinite, append ($+\!\!+$) loses information that set union ($\cup$) would not.

The non-determinism monad has a morphism, *merge*, that combines a finite number of non-deterministic computations, each producing a finite set of values, into a single computation returning their union. For the finite powerset monad, it is union ($\cup$), while with the list implementation, *merge* is concatenation:

```
merge[] :: [[a]]→[a]
merge[] = concat
```

Note that the finiteness of the argument of *merge* is assumed and is not reflected in its type.

3 A Monadic Model for Asynchronous Exceptions

This section presents a monadic model for asynchronous exceptions (MMAE). The MMAE is an algebra of effects with monads and associated operators expressing notions of non-determinism, concurrency and interactivity. Section 3.1 first defines this algebra of effects and, then, Section 3.2 presents a number of theorems specifying the interactions between the algebraic operators. The convention that operators associated with a monad are referred to as *effects* is followed throughout this paper.

3.1 Monadic Algebra of Effects

This section presents the effect algebra underlying the semantics in Section 4. In Section 5, this algebra will be extended and generalized. The left hand column in Definition 1 below specifies the functor parts of three monads using a category-theoretic notation while the right hand column presents them as data type declarations in the Haskell language. The intention in doing so is to appeal to the widest audience. One should note, however, that the Haskell representations are really approximate (e.g., lists in Haskell may be infinite).

Definition 1 (Functors for monads N,E,R)

$$\begin{array}{llcl@{\qquad}lcl}
\textit{Non-deter.} & \mathsf{N}\,A & = & \mathcal{P}_{\mathit{fin}}(A) & \textbf{type}\,\mathsf{N}\,a & = & [a] \\
 & & & & \textbf{data}\,\mathit{Err}\;a & = & \mathit{Ok}\;a \mid \mathit{Error} \\
\textit{Exceptions} & \mathsf{E}\,A & = & \mathsf{N}(A + \mathit{Error}) & \textbf{type}\,\mathsf{E}\,a & = & \mathsf{N}(\mathit{Err}\;a) \\
\textit{Concur.} & \mathsf{R}\,A & = & \mathit{fix}\,X.\,A + \mathsf{E}\,X & \textbf{data}\,\mathsf{R}\,a & = & \mathit{Done}\;a \mid \mathit{Pause}\,(\mathsf{E}\,(\mathsf{R}\,a))
\end{array}$$

Technical note: In the categorical definition of R (left column, bottom), the binder $\mathit{fix}\,X$ can be taken to indicate either the least or greatest fixed point solution to the corresponding recursive domain equation. For the semantics in Section 4, either will suffice as there will be no need to represent infinite computations. In Section 5, it will represent the greatest fixed point.

The monad R supports a model of concurrency in which computations have a thread-like form. An R “thread” is a sequencing of E-operations having either the form $(\mathit{Done}\;v)$ or $(\mathit{Pause}\,\varphi)$ for $\varphi : \mathsf{E}(\mathsf{R}\,a)$. The intuition is that, if a thread is finished executing, then it is $(\mathit{Done}\;v)$ for $v : a$, signifying that there are no further E-operations to perform and its final value is v. If the thread is not finished, it performs the E-operation, φ, the computed value of which is its own thread continuation (i.e., an R-computation). For further development of the resumption-monadic model of concurrency, please refer to the references [9,22].

There is a codebase with Haskell implementations of the monadic constructions presented in this paper [8]. Each of the operator definitions below is followed by Haskell examples demonstrating the operators. For consistency, Haskell concrete syntax is eschewed in favor of the mathematical syntax of this paper. So, for example, Haskell lists representing sets are written with set brackets “{“ and “}” rather than with Haskell list constructors “[” and “]” and any other Haskell syntactic details inessential to the presentation are struck altogether.

Definition 2 specifies the unit (η) and bind ($\star$) operators for the N, E and R monads. Discussion motivating the definitions of $\star_{\mathsf{N}}$ and η_{N} can be found in Section 2.1.

Definition 2 (η,$\star$ for monads N,E,R). *The unit* (η) *and bind* ($\star$) *operations have type* $\eta_{\mathsf{m}} : a \rightarrow \mathsf{m}\,a$ *and* $(\star_{\mathsf{m}}) : \mathsf{m}\,a \rightarrow (a \rightarrow \mathsf{m}\,b) \rightarrow \mathsf{m}\,b$ *for monads* $\mathsf{m} = \mathsf{N}$, E, R *and are defined by the following equations:*

$$\begin{array}{l}
\eta_{\mathsf{N}} x \;=\; \{x\} \\
\{\varphi_1, \ldots, \varphi_n\} \star_{\mathsf{N}} f \\
\qquad = \; \bigcup (f\,\varphi_i)
\end{array}
\qquad
\begin{array}{l}
\eta_{\mathsf{E}} \qquad = \eta_{\mathsf{N}} \circ \mathit{Ok} \\
\varphi \star_{\mathsf{E}} f = \\
\qquad \varphi \star_{\mathsf{N}} \lambda v. \\
\qquad\quad \textbf{case}\; v \;\textbf{of} \\
\qquad\qquad (\mathit{Ok}\,x) \;\rightarrow\; f\,x \\
\qquad\qquad \mathit{Error} \;\;\rightarrow\; \eta_{\mathsf{N}} \mathit{Error}
\end{array}
\qquad
\begin{array}{l}
\eta_{\mathsf{R}} \;=\; \mathit{Done} \\
(\mathit{Done}\;v) \star_{\mathsf{R}} f \;\; = f\;v \\
(\mathit{Pause}\,\varphi) \star_{\mathsf{R}} f = \\
\qquad \mathit{Pause}(\varphi \star_{\mathsf{E}} \lambda\kappa.\eta_{\mathsf{E}}(\kappa \star_{\mathsf{R}} f))
\end{array}$$

Below are Haskell examples demonstrating the unit and bind operators of the N and E monads. Binding N-computation $\{1,2,3\}$ to the function $(\lambda v.\eta_{\mathsf{N}}(v+1))$ with $\star_{\mathsf{N}}$ has the effect of incrementing each element. Binding E-computation $\{\mathit{Ok}\,1, \mathit{Ok}\,2, \mathit{Error}\}$ to the function $(\lambda v.\eta_{\mathsf{E}}(v+1))$ with $\star_{\mathsf{E}}$ performs a similar "mapping" to the previous case: each $(\mathit{Ok}\ x)$ element is incremented while the *Error* is unchanged.

```
Haskell> $\eta_{\mathsf{N}}\ 9$
   $\{\,9\,\}$
Haskell> $\eta_{\mathsf{E}}\ 9$
   $\{\ \mathit{Ok}\ 9\ \}$
Haskell> $\{1,2,3\} \star_{\mathsf{N}} (\lambda v.\ \eta_{\mathsf{N}}(v+1))$
   $\{2,3,4\}$
Haskell> $\{\mathit{Ok}\ 1, \mathit{Ok}\ 2, \mathit{Error}\} \star_{\mathsf{E}} (\lambda v.\ \eta_{\mathsf{E}}(v+1))$
   $\{\mathit{Ok}\ 2, \mathit{Ok}\ 3, \mathit{Error}\}$
```

The *step* operator takes an E-computation and produces an R-computation; it is an example of a *lifting* [16] from E to R. The lifted computation, $\mathit{step}\ x$, is atomic in the sense that there are no intermediate *Pause* breakpoints and is, in that sense, indivisible. The *run* operator projects computations from R to E. An R-computation may be thought of intuitively as an E-computation with a number (possibly infinite) of inserted *Pause* breakpoints; *run* removes each of those breakpoints.

Definition 3 (Operators relating E and R)

$$\begin{array}{ll} \mathit{step} : \mathsf{E}\,a \to \mathsf{R}\,a & \mathit{run} : \mathsf{R}\,a \to \mathsf{E}\,a \\ \mathit{step}\,x \ = \ \mathit{Pause}(x \star_{\mathsf{E}} (\eta_{\mathsf{E}} \circ \mathit{Done})) & \mathit{run}(\mathit{Pause}\,\varphi) = \varphi \star_{\mathsf{E}} \mathit{run} \\ & \mathit{run}\ (\mathit{Done}\ v) = \eta_{\mathsf{E}}\ v \end{array}$$

The Haskell session below shows an example of how the *run* operator "unrolls" an R-computation. In fact, applying *run* to the lifting of an E-computation makes no change to that computation. This suggests that *run* is an inverse of *step*, which is, in fact, the case (see Theorem 5 below).

```
Haskell> $\mathit{run}\ (\mathit{step}\ \{\mathit{Ok}\ 1, \mathit{Ok}\ 2, \mathit{Error}\})$
   $\{\mathit{Ok}\ 1, \mathit{Ok}\ 2, \mathit{Error}\}$
```

The *merge* operators on N, E and R are given in Definition 4. As discussed in Section 2.1 above, $\mathit{merge}_{\mathsf{N}}$ is a non-proper morphism in the non-determinism monad. Its definition is lifted to the E and R monads below:

Definition 4 (Merge operators)

$$\begin{array}{l} \mathit{merge}_{\mathsf{N}} : \mathsf{N}(\mathsf{N}\,a) \to \mathsf{N}\,a \\ \mathit{merge}_{\mathsf{N}}\ X = \cup_{(x \in X)}\ x \\ \mathit{merge}_{\mathsf{E}} : \mathsf{N}(\mathsf{E}\,a) \to \mathsf{E}\,a \end{array}$$

$$\begin{aligned}
&\mathit{merge}_{\mathsf{E}} \quad = \mathit{merge}_{\mathsf{N}}\\
&\mathit{merge}_{\mathsf{R}} : \mathsf{N}(\mathsf{R}\,a) \rightarrow \mathsf{R}\,a\\
&\mathit{merge}_{\mathsf{R}}\{\varphi_1, \ldots, \varphi_n\} = \mathit{Pause}(\mathit{merge}_{\mathsf{E}}\{\eta_{\mathsf{E}}\,\varphi_1, \ldots, \eta_{\mathsf{E}}\,\varphi_n\})
\end{aligned}$$

The effect of merging some finite number of computations in N or E together is to collect the sets of their outcomes into a single outcome set. Merging in R has a similar effect, but, rather than collecting outcomes, $\mathit{merge}_{\mathsf{R}}\{\varphi_1, \ldots, \varphi_n\}$ creates a single R-computation that branches out with n “sub-computations”.

Haskell> $\mathit{merge}_{\mathsf{N}}\ \{\{1, 2\}, \{4\}\}$
  $\{1, 2, 4\}$
Haskell> $\mathit{merge}_{\mathsf{E}}\ \{\{\mathit{Ok}\,1, \mathit{Ok}\,2, \mathit{Error}\},\ \{\mathit{Ok}\,4,\ \mathit{Error}\}\}$
  $\{\mathit{Ok}\,1, \mathit{Ok}\,2, \mathit{Ok}\,4, \mathit{Error}\}$ — dupl. *Error* not shown

An equation like “ $\langle\text{raise exception}\rangle \star f \ = \ \langle\text{raise exception}\rangle$” will hold in any exception monad like E. That is, a raised exception trumps any effects that follow. The *status* operators for E and R catch an exception producing computation and “defuse” it. So, if $\varphi : \mathsf{E}\ a$ produces an exception, then the value returned by $\mathit{status}_{\mathsf{E}}(\varphi) : \mathsf{E}\,(a + \mathit{Error})$ will be *Error* itself.

The *status* operators may be used to discern when an exception has occurred in its argument while isolating that exception's effect. Below in Definition 6, an operator *catch* is defined so that $(\mathit{catch}\,\varphi\,\gamma)$ equals φ if φ does not throw an exception, but equals γ otherwise. The *status* operator is used in the definition of *catch* to test if its first argument throws an exception so that *catch* can, in that event, return its second argument.

Definition 5 (Status)

```
$status_{\mathsf{E}} : \mathsf{E}\,a \rightarrow \mathsf{E}(a + Error)$
$status_{\mathsf{E}}\varphi =$
    $\varphi \star_{\mathsf{N}} \lambda v.$
      case $v$ of
        $(Ok\ y) \rightarrow \eta_{\mathsf{E}}(Ok\,y)$
        $Error \quad \rightarrow \eta_{\mathsf{E}} Error$

$status_{\mathsf{R}} : \mathsf{R}\,a \rightarrow \mathsf{R}\,(a + Error)$
$status_{\mathsf{R}}\ (Pause\ \varphi) =$
    $Pause\,(status_{\mathsf{E}}\ \varphi \star_{\mathsf{E}} \lambda v.$
      case $v$ of
        $(Ok\,x) \rightarrow \eta_{\mathsf{E}}(status_{\mathsf{R}}\ x)$
        $Error \ \rightarrow \eta_{\mathsf{E}}(Done\ Error))$
$status_{\mathsf{R}}\ (Done\ v) \ = (Done\ (Ok\ v))$
```

The following examples demonstrate how *status* interacts with computations that throw exceptions. The $\mathit{throw}_{\mathsf{E}} : \mathsf{E}\ a$ operator raises an exception (it is defined below in Definition 6). Note how (in the second example) it “trumps” the remaining computation. The third example, $(\mathit{status}_{\mathsf{E}}\ \mathit{throw}_{\mathsf{E}})$, shows how the effect of the $\mathit{throw}_{\mathsf{E}}$ exception is isolated. Instead of returning the exception $\{\ \mathit{Error}\ \}$, the $(\mathit{status}_{\mathsf{E}}\ \mathit{throw}_{\mathsf{E}})$ computation returns the *Error* token as its value. With a non-exception throwing computation (e.g., $(\eta_{\mathsf{E}}\,9)$), $\mathit{status}_{\mathsf{E}}$ returns the value produced by its argument wrapped in an *Ok*.

Haskell> $\mathit{throw}_{\mathsf{E}}$
$\{\ \mathit{Error}\ \}$
Haskell> $\mathit{throw}_{\mathsf{E}} \star_{\mathsf{E}} \lambda v.\, \eta_{\mathsf{E}}\ (v+1)$
$\{\ \mathit{Error}\ \}$
Haskell> $\mathit{status}_{\mathsf{E}}\ \mathit{throw}_{\mathsf{E}}$
$\{\mathit{Ok}\ \mathit{Error}\}$
Haskell> $\mathit{status}_{\mathsf{E}}\ (\eta_{\mathsf{E}}\ 9)$
$\{\mathit{Ok}\ (\mathit{Ok}\ 9)\}$

Definition 6 gives the specification for the exception-raising and -catching operations, *throw* and *catch*, in E and R and the branching operation, *fork*, in the R monad. The *fork* operation is particularly important in the semantic framework of the next section. If $\varphi : \mathsf{R}\ a$, then $(\mathit{fork}\ \varphi) : \mathsf{R}\ a$ is a computation that, roughly speaking, will do either whatever φ would do or be asynchronously interrupted by exception $\mathit{throw}_{\mathsf{R}}$.

Definition 6 (Control flow operators)

$\mathit{throw}_{\mathsf{E}} : \mathsf{E}\ a$
$\mathit{throw}_{\mathsf{E}} = \eta_{\mathsf{N}}\ \mathit{Error}$
$\mathit{throw}_{\mathsf{R}} : \mathsf{R}\ a$
$\mathit{throw}_{\mathsf{R}} = \mathit{step}\ \mathit{throw}_{\mathsf{E}}$
$\mathit{fork} : \mathsf{R}\ a \to \mathsf{R}\ a$
$\mathit{fork}\ \varphi\ = \mathit{merge}_{\mathsf{R}}\ \{\varphi, \mathit{throw}_{\mathsf{R}}\}$

For monad $\mathsf{m} = \mathsf{E}, \mathsf{R}$,
$\mathit{catch}_{\mathsf{m}} : \mathsf{m}\ a \to \mathsf{m}\ a \to \mathsf{m}\ a$
$\mathit{catch}_{\mathsf{m}}\ \varphi\ \gamma = (\mathit{status}_{\mathsf{m}}\ \varphi) \star_{\mathsf{m}} \lambda s.$
case s **of**
$(\mathit{Ok}\, v) \ \to\ \eta_{\mathsf{m}}\ v$
$\mathit{Error} \ \to\ \gamma$

The first two examples in the following Haskell transcript illustrate the behavior of $\mathit{catch}_{\mathsf{E}}$ and $\mathit{throw}_{\mathsf{E}}$. If the first argument to $\mathit{catch}_{\mathsf{E}}$ raises an exception (as, obviously, $\mathit{throw}_{\mathsf{E}}$ does), then the second argument is returned. If the first argument to *catch* does not raise an exception, then its second argument is ignored. The third and fourth examples illustrate the *fork* effect. The effect of $\mathit{fork}\ (\eta_{\mathsf{R}}\ 9)$ is to create a single R-computation with two "branches" (both underlined below). The first branch is just the argument to *fork* (recall that $\eta_{\mathsf{R}} = \mathit{Done}$) and the second branch is equal to $(\mathit{step}\ \mathit{throw}_{\mathsf{E}})$. This simple example exposes an important construction within the MMAE: an asynchronous exception thrown to a computation is modeled as an alternative branch from the computation. The fourth example shows the application of *run* to the previous example. The result is to collect all possible outcomes from each branch. Following a tree data type analogy, *run* is used to calculate the *fringe*.

Haskell> $\mathit{catch}_{\mathsf{E}}\ \mathit{throw}_{\mathsf{E}}\ (\eta_{\mathsf{E}}\ 9)$
$\{\mathit{Ok}\ 9\}$
Haskell> $\mathit{catch}_{\mathsf{E}}\ (\eta_{\mathsf{E}}\ 9)\ \mathit{throw}_{\mathsf{E}}$
$\{\mathit{Ok}\ 9\}$
Haskell> $\mathit{fork}\ (\eta_{\mathsf{R}}\ 9)$
$\mathit{Pause}\ (\{\ \mathit{Ok}\ \underline{(\mathit{Done}\ 9)},\ \mathit{Ok}\ \underline{(\mathit{Pause}\ (\{\ \mathit{Error}\ \}\ ))}\})$
Haskell> $\mathit{run}\ (\mathit{fork}\ (\eta_{\mathsf{R}}\ 9))$
$\{\mathit{Ok}\ 9, \mathit{Error}\}$

3.2 Interactions between Effects

This section presents a number of theorems characterizing the algebraic effects developed in the previous section and their relationships to one another. These theorems are used in the next section to prove an equivalence between the MMAE semantics given there and recently published semantics for asynchronous exceptions [13].

Theorem 1 gives a distribution rule for $\star$ over *merge*. This distribution exposes the tree-like structure of computations in R.

Theorem 1. *For monads* $\mathsf{m} = \mathsf{N}, \mathsf{E}, \mathsf{R}$, $\star_{\mathsf{m}}$ *distributes over* $\mathit{merge}_{\mathsf{m}}$*:*

$$\mathit{merge}_{\mathsf{m}}\{\varphi_1, \ldots, \varphi_n\} \star_{\mathsf{m}} f = \mathit{merge}_{\mathsf{m}}\{\varphi_1 \star_{\mathsf{m}} f, \ldots, \varphi_n \star_{\mathsf{m}} f\}$$

Proof.

$$\begin{aligned}
&\mathit{merge}_{\mathsf{N}}\{\varphi_1, \ldots, \varphi_n\} \star_{\mathsf{N}} f \\
\{\text{def } \mathit{merge}_{\mathsf{N}}\} &= (\cup\, \varphi_i) \star_{\mathsf{N}} f \\
\{\text{def } \star_{\mathsf{N}}\} &= f\ (\cup\, \varphi_i) \\
&= \cup\, (f\ \varphi_i) \\
\{\text{def } \mathit{merge}_{\mathsf{N}}\} &= \mathit{merge}_{\mathsf{N}}\ \{f\ \varphi_1, \ldots, f\ \varphi_n\} \\
\{\text{def } \star_{\mathsf{N}}\} &= \mathit{merge}_{\mathsf{N}}\ \{\varphi_1 \star_{\mathsf{N}} f, \ldots, \varphi_n \star_{\mathsf{N}} f\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{merge}_{\mathsf{E}}\{\varphi_1, \ldots, \varphi_n\} \star_{\mathsf{E}} f \\
&= \mathit{merge}_{\mathsf{E}}\ \{\varphi_1, \ldots, \varphi_n\} \star_{\mathsf{N}} \hat{f} \\
&\qquad \text{where } \hat{f}\ v \ = \ \ \textbf{case}\ v\ \textbf{of} \\
&\qquad\qquad\qquad (\mathit{Ok}\ x) \ \rightarrow\ \eta_{\mathsf{N}}\ (f\ x) \\
&\qquad\qquad\qquad \mathit{Error} \ \rightarrow\ \eta_{\mathsf{N}}\ \mathit{Error} \\
\{\text{def } \mathit{merge}_{\mathsf{E}}\} &= \mathit{merge}_{\mathsf{N}}\ \{\varphi_1, \ldots, \varphi_n\} \star_{\mathsf{N}} \hat{f} \\
\{\text{prev. case}\} &= \mathit{merge}_{\mathsf{N}}\ \{\varphi_1 \star_{\mathsf{N}} \hat{f}, \ldots, \varphi_n \star_{\mathsf{N}} \hat{f}\} \\
\{\text{def } \star_{\mathsf{E}}\} &= \mathit{merge}_{\mathsf{N}}\ \{\varphi_1 \star_{\mathsf{E}} f, \ldots, \varphi_n \star_{\mathsf{E}} f\} \\
\{\text{def } \mathit{merge}_{\mathsf{E}}\} &= \mathit{merge}_{\mathsf{E}}\ \{\varphi_1 \star_{\mathsf{E}} f, \ldots, \varphi_n \star_{\mathsf{E}} f\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{merge}_{\mathsf{R}}\ \{\varphi_1, \ldots, \varphi_n\} \star_{\mathsf{R}} f \\
\{\text{def } \mathit{merge}_{\mathsf{R}}\} &= \mathit{Pause}\ (\mathit{merge}_{\mathsf{E}}\ \{\eta_{\mathsf{E}}\ \varphi_1, \ldots, \eta_{\mathsf{E}}\ \varphi_n\}) \star_{\mathsf{R}} f \\
\{\text{def } \star_{\mathsf{R}}\} &= \mathit{Pause}\ (\mathit{merge}_{\mathsf{E}}\ \{\eta_{\mathsf{E}}\ \varphi_1, \ldots, \eta_{\mathsf{E}}\ \varphi_n\} \star_{\mathsf{E}} \lambda\kappa.\ \eta_{\mathsf{E}}\ (\kappa \star_{\mathsf{R}} f)) \\
\{\text{prev. case}\} &= \mathit{Pause}\ (\mathit{merge}_{\mathsf{E}}\ \{\varphi'_1, \ldots, \varphi'_n\}) \\
&\qquad \text{where } \varphi'_i \ = \ (\eta_{\mathsf{E}}\ \varphi_i) \star_{\mathsf{E}} \lambda\kappa.\ \eta_{\mathsf{E}}\ (\kappa \star_{\mathsf{R}} f) \\
\{\text{left unit}\} &= \mathit{Pause}\ (\mathit{merge}_{\mathsf{E}}\ \{\ \eta_{\mathsf{E}}\ (\varphi_1 \star_{\mathsf{R}} f), \ldots,\ \eta_{\mathsf{E}}\ (\varphi_n \star_{\mathsf{R}} f)\}) \\
\{\text{def } \mathit{merge}_{\mathsf{R}}\} &= \mathit{merge}_{\mathsf{R}}\ \{\varphi_1 \star_{\mathsf{R}} f, \ldots,\ \varphi_n \star_{\mathsf{R}} f\}
\end{aligned}$$

□

Theorem 2 gives a distribution law for *run* over $\mathit{merge}_{\mathsf{R}}$. This distribution law is something like an inverse "lifting" as it projects resumption-based merged computations into a single computation in E.

Theorem 2. $run(merge_{\mathsf{R}}\{\varphi_1,\ldots,\varphi_n\}) = merge_{\mathsf{E}}\{run\,\varphi_1,\ldots,run\,\varphi_n\}$

Proof.

$$\begin{array}{ll}
 & run\,(merge_{\mathsf{R}}\,\{\varphi_1,\ldots,\varphi_n\}) \\
\{\text{def } merge_{\mathsf{R}}\ \} & = run\,(Pause\,(merge_{\mathsf{E}}\,\{\eta_{\mathsf{E}}\,\varphi_1,\ldots,\eta_{\mathsf{E}}\,\varphi_n\})) \\
\{\text{def } run\ \} & = (merge_{\mathsf{E}}\,\{\eta_{\mathsf{E}}\,\varphi_1,\ldots,\eta_{\mathsf{E}}\,\varphi_n\}) \star_{\mathsf{E}} run \\
\{\text{thm 1}\} & = merge_{\mathsf{E}}\,\{(\eta_{\mathsf{E}}\,\varphi_1) \star_{\mathsf{E}} run\,,\ldots,(\eta_{\mathsf{E}}\,\varphi_n) \star_{\mathsf{E}} run\} \\
\{\text{left unit}\} & = merge_{\mathsf{E}}\,\{run\,\varphi_1,\ldots,\ run\,\varphi_n\}
\end{array}$$
□

Theorem 3 shows how *run* distributes over $\star_{\mathsf{R}}$ to produce an E-computation. Theorem 3 may be proved easily by induction on the number of *Pause* constructors in its argument if that number is finite. If it is not, then the property is trivially true, because, in that case, both sides of Theorem 3 are $\bot$.

Theorem 3. $run(x \star_{\mathsf{R}} f) = (run\ x) \star_{\mathsf{E}} (run \circ f)$

Theorem 4 shows that the *throw* exception overrides any effects that follow it. A consequence of this theorem is that, for any $f, g : a \to \mathsf{E}\,b$

$$throw_{\mathsf{E}} \star_{\mathsf{E}} f \ = \ throw_{\mathsf{E}} \ = \ throw_{\mathsf{E}} \star_{\mathsf{E}} g$$

Theorem 4. $throw_{\mathsf{m}} \star_{\mathsf{m}} f = throw_{\mathsf{m}}$*, for monad* $\mathsf{m} \ = \ \mathsf{E}, \mathsf{R}$.

Proof.

$$\begin{array}{ll}
 & throw_{\mathsf{E}} \star_{\mathsf{E}} f \\
\{\text{def } throw_{\mathsf{E}}\} & = (\eta_{\mathsf{N}}\ Error) \star_{\mathsf{E}} f \\
\{\text{def } \star_{\mathsf{E}}\} & = (\eta_{\mathsf{N}}\ Error) \star_{\mathsf{N}} \lambda v. \\
 & \qquad\qquad \mathbf{case}\ v\ \mathbf{of}\ \{\ (Ok\,x)\ \to\ \eta_{\mathsf{N}}\,(f\ x)\,;\ Error\ \to\ \eta_{\mathsf{N}}\ Error\,\} \\
\{\text{left unit}\} & = \mathbf{case}\ Error\ \mathbf{of}\ \{\ (Ok\,x)\ \to\ \eta_{\mathsf{N}}\,(f\ x)\,;\ Error\ \to\ \eta_{\mathsf{N}}\ Error\,\} \\
 & = \eta_{\mathsf{N}}\ Error\ =\ throw_{\mathsf{E}}
\end{array}$$

$$\begin{array}{ll}
 & throw_{\mathsf{R}} \star_{\mathsf{R}} f \\
\{\text{def } throw_{\mathsf{R}}\} & = (step\ throw_{\mathsf{E}}) \star_{\mathsf{R}} f \\
\{\text{def } step\} & = (Pause\ (throw_{\mathsf{E}} \star_{\mathsf{E}} (\eta_{\mathsf{E}}\ \circ\ Done)) \star_{\mathsf{R}} f \\
\{\text{def } \star_{\mathsf{R}}\} & = Pause\ (throw_{\mathsf{E}} \star_{\mathsf{E}} \lambda\kappa.\ (\eta_{\mathsf{E}}\ (\kappa \star_{\mathsf{R}} f))) \\
\{\text{prev. case}\} & = Pause\ (throw_{\mathsf{E}} \star_{\mathsf{E}} (\eta_{\mathsf{E}}\ \circ\ Done)) =\ step\ throw_{\mathsf{E}}\ =\ throw_{\mathsf{R}}
\end{array}$$
□

Theorem 5 states that the *run* operation is the inverse of *step*. A consequence of this theorem is that $run\ throw_{\mathsf{R}}\ =\ throw_{\mathsf{E}}$.

Theorem 5. $run\,(step\,\varphi)\ =\ \varphi$

Proof.

$$\begin{array}{ll}
 & run\,(step\varphi) \\
\{\text{def } step\} & = run\ (Pause\ (\varphi \star_{\mathsf{E}} (\eta_{\mathsf{E}} \circ Done))
\end{array}$$

$$\begin{aligned}
\{\text{def } \mathit{run}\} &= (\varphi \star_{\mathsf{E}} (\eta_{\mathsf{E}} \circ \mathit{Done})) \star_{\mathsf{E}} \mathit{run} \\
\{\text{assoc.}\} &= \varphi \star_{\mathsf{E}} \lambda v.\, \eta_{\mathsf{E}}(\mathit{Done}\ v) \star_{\mathsf{E}} \mathit{run} \\
\{\text{left unit}\} &= \varphi \star_{\mathsf{E}} \lambda v.\, \mathit{run}\ (\mathit{Done}\ v) \\
\{\text{def } \mathit{run}\} &= \varphi \star_{\mathsf{E}} \lambda v.\, \eta_{\mathsf{E}}\ v \\
\{\text{eta red}\} &= \varphi \star_{\mathsf{E}} \eta_{\mathsf{E}} \\
\{\text{rt unit}\} &= \varphi
\end{aligned}$$

□

4 The MMAE as a Semantic Framework

This section presents the semantics for the exception language of Hutton and Wright [13] in terms of the MMAE. Hutton and Wright define a natural semantics for a small language (called henceforth *Expr*) combining arithmetic expressions and synchronous exceptions (e.g., *Catch* and *Throw*). The natural semantics of this synchronous fragment of the language is just what one would expect. What makes the language interesting is the presence of an asynchronous interrupt and its manifestation within the semantics. The *Expr* language and its natural semantics are found in Figure 2. A monadic semantics for *Expr* is given below and then the equivalence of both semantics is formulated in Theorem 6.

This section presents the formulation of Hutton and Wright's language within the monadic framework of Section 3. The evaluation relation is annotated by a B or U, indicating whether interrupts are blocked or unblocked, respectively. Ignoring this flag for the moment, the first three rows in Figure 2 are a conventional, natural semantics for a language combining arithmetic with synchronous exceptions. Note, for example, that the interaction of *Add* with *Throw* is specified by three rules (*Add*1, *Add*2, *Add*3), the first for the case of exception-free arguments and the second two for the cases when an argument evaluates to *Throw*. The effect of (*Block* e) [(*Unblock* e)] is to turn off [on] asynchronous exceptions in the evaluation of e.

$$\frac{}{\mathit{Val}\ n \Downarrow^{i} \mathit{Val}\ n}\ \text{Val} \qquad \frac{}{\mathit{Throw} \Downarrow^{i} \mathit{Throw}}\ \text{Throw} \qquad \frac{x \Downarrow^{i} \mathit{Val}\ n \quad y \Downarrow^{i} \mathit{Val}\ m}{\mathit{Add}\ x\ y \Downarrow^{i} \mathit{Val}\ (n{+}m)}\ \text{Add1}$$

$$\frac{x \Downarrow^{i} \mathit{Throw}}{\mathit{Add}\ x\ y \Downarrow^{i} \mathit{Throw}}\ \text{Add2} \qquad \frac{y \Downarrow^{i} \mathit{Throw}}{\mathit{Add}\ x\ y \Downarrow^{i} \mathit{Throw}}\ \text{Add3} \qquad \frac{y \Downarrow^{i} v}{\mathit{Seqn}\ x\ y \Downarrow^{i} v}\ \text{Seqn1}$$

$$\frac{x \Downarrow^{i} \mathit{Throw}}{\mathit{Seqn}\ x\ y \Downarrow^{i} \mathit{Throw}}\ \text{Seqn2} \qquad \frac{x \Downarrow^{i} \mathit{Val}\ n}{\mathit{Catch}\ x\ y \Downarrow^{i} \mathit{Val}\ n}\ \text{Catch1} \qquad \frac{x \Downarrow^{i} \mathit{Throw} \quad y \Downarrow^{i} v}{\mathit{Catch}\ x\ y \Downarrow^{i} v}\ \text{Catch2}$$

$$\frac{x \Downarrow^{B} v}{\mathit{Block}\ x \Downarrow^{i} v}\ \text{Block} \qquad \frac{x \Downarrow^{U} v}{\mathit{Unblock}\ x \Downarrow^{i} v}\ \text{Unblock} \qquad \frac{}{x \Downarrow^{U} \mathit{Throw}}\ \text{Int}$$

Fig. 2. Hutton and Wright's Expression Language, Expr, with Asynchronous Exceptions. The essence of asynchronous exceptions in this model is captured in the Int rule.

$$\dfrac{\dfrac{}{Val\ 1 \Downarrow^{U}\ Val\ 1}\,\text{Val} \qquad \dfrac{}{Val\ 2 \Downarrow^{U}\ Val\ 2}\,\text{Val}}{Add\ (Val\ 1)\ (Val\ 2) \Downarrow^{U}\ Val\ 3}\,\text{Add1} \qquad \dfrac{}{Add\ (Val\ 1)\ (Val\ 2) \Downarrow^{U}\ Throw}\,\text{Int}$$

$$\dfrac{\dfrac{}{Val\ 1 \Downarrow^{U}\ Throw}\,\text{Int}}{Add\ (Val\ 1)\ (Val\ 2) \Downarrow^{U}\ Throw}\,\text{Add2} \qquad \dfrac{\dfrac{}{Val\ 2 \Downarrow^{U}\ Throw}\,\text{Int}}{Add\ (Val\ 1)\ (Val\ 2) \Downarrow^{U}\ Throw}\,\text{Add3}$$

Fig. 3. There are four possible evaluations of (*Add* (*Val* 1) (*Val* 2)) according to the unblocked semantics (i.e., with *U* annotation) in Figure 2

To understand how Hutton and Wright's semantics works and to compare it with the monadic semantics given here, the expression (*Add* (*Val* 1) (*Val* 2)) will be used as a running example. The expression (*Add* (*Val* 1) (*Val* 2)) has four possible evaluations when the interrupt flag is *U* (shown in Figure 3). The evaluation in the upper left is what one would expect. But the three other cases involve asynchronous exceptions, evaluating instead to *Throw* via the *Int* rule from Figure 2. The bottom row shows what happens when the first and second arguments, respectively, evaluate to *Throw* and, in the upper right corner, the evaluation is interrupted "before" any computation takes place. The *Int* rule may be applied because the flag is *U* and consequently there are four possible evaluations. When the flag is *B*, the *Int* rule may not be applied, and so exceptions are blocked. There is one and only one evaluation when the flag is *B*:

$$\dfrac{\dfrac{}{Val\ 1 \Downarrow^{B}\ Val\ 1}\,\text{Val} \qquad \dfrac{}{Val\ 2 \Downarrow^{B}\ Val\ 2}\,\text{Val}}{Add\ (Val\ 1)\ (Val\ 2) \Downarrow^{B}\ Val\ 3}\,\text{Add1}$$

Figure 4 presents the semantics of *Expr* using the MMAE framework. The semantics consists of two semantic functions, the "blocked" semantics $\mathcal{B}[\![-]\!]$ and the "unblocked" semantics $\mathcal{U}[\![-]\!]$. The blocked (unblocked) semantics provides the meaning of an expression when interrupts are off (on) and corresponds to the natural semantic relation $\Downarrow^{B}$ ($\Downarrow^{U}$). The first five semantic equations for $\mathcal{B}[\![-]\!]$ are a conventional monadic semantics for arithmetic, sequencing and synchronous exceptions [22]. The definitions for *Block* and *Unblock* require comment, however. The meaning of (*Block e*) in the $\mathcal{B}[\![-]\!]$ has no effect, because asynchronous exceptions are already blocked. The meaning of (*Unblock e*), however, is $\mathcal{U}[\![e]\!]$, signifying thereby that asynchronous exceptions are unblocked in the evaluation of *e*. The unblocked semantics is similar to the blocked, except that *fork* is applied to each denoting computation. This has the effect of creating an asynchronous exception at each application of *fork*. $\mathcal{U}[\![Block\, e]\!]$ is defined as $\mathcal{B}[\![e]\!]$ to "turn off" asynchronous exceptions.

Example: Blocked Semantics. In the following example, the "blocked" denotation of *Add (Val 1) (Val 2)* is simplified with respect to the theorems of Section 3.

$$\begin{aligned}
&\mathcal{B}[\![\mathit{Add}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2)]\!]\\
\{\text{def } \mathcal{B}[\![-]\!]\} &= (\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.(\mathit{step}(\eta_{\mathsf{E}}\ 2)) \star_{\mathsf{R}} \lambda v_2.\eta_{\mathsf{R}}(v_1 + v_2)\\
\{\text{def } \star_{\mathsf{R}}\} &= (\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.(\mathit{Pause}\ ((\eta_{\mathsf{E}}\ 2) \star_{\mathsf{E}} (\eta_{\mathsf{E}} \circ \mathit{Done}))) \star_{\mathsf{R}} \lambda v_2.\eta_{\mathsf{R}}(v_1 + v_2)\\
\{\text{left unit}\} &= (\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.(\mathit{Pause}\ (\eta_{\mathsf{E}}\ (\mathit{Done}\ 2))) \star_{\mathsf{R}} \lambda v_2.\eta_{\mathsf{R}}(v_1 + v_2)\\
\{\text{def } \star_{\mathsf{R}}\} &= (\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.\mathit{Pause}\ (\eta_{\mathsf{E}}\ ((\mathit{Done}\ 2) \star_{\mathsf{R}} \lambda v_2.\eta_{\mathsf{R}}(v_1 + v_2)))\\
\{\text{def } \star_{\mathsf{R}}\} &= (\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.(\mathit{Pause}\ (\eta_{\mathsf{E}}\ (\eta_{\mathsf{R}}(v_1 + 2))))\\
&= \mathit{Pause}\ (\eta_{\mathsf{E}}\ (\mathit{Pause}\ (\eta_{\mathsf{E}}\ (\mathit{Done}\ 3))))
\end{aligned}$$

The last line follows by a similar argument to the first five steps. This last R-computation, $\mathit{Pause}\ (\eta_{\mathsf{E}}\ (\mathit{Pause}\ (\eta_{\mathsf{E}}\ (\mathit{Done}\ 3))))$, captures the operational content of $\mathcal{B}[\![\mathit{Add}(\mathit{Val} 1)\,(\mathit{Val} 2)]\!]$. It is a single thread with two steps in succession, corresponding to the evaluation of $(\mathit{Val}\ 1)$ and $(\mathit{Val}\ 2)$, followed by the return of the computed value 3.

Example: Unblocked Semantics. The next example considers the same expression evaluated under the "unblocked" semantics, $\mathcal{U}[\![-]\!]$. As in the previous example, the denotation is "normalized", so to speak, according to the theorems of Section 3. Before beginning the example of the unblocked semantics, we state and prove a useful simplification in Lemma 1.

Lemma 1. *$\mathit{step}\ (\eta_{\mathsf{E}}\ v) \star_{\mathsf{R}} f\ =\ \mathit{Pause}\ (\eta_{\mathsf{E}}\ (f\ v))$.*

Proof.

$$\begin{aligned}
&\mathit{step}\ (\eta_{\mathsf{E}} v) \star_{\mathsf{R}} f\\
\{\text{def } \mathit{step}\} &= \mathit{Pause}\ ((\eta_{\mathsf{E}}\ v) \star_{\mathsf{E}} (\eta_{\mathsf{E}}\ \circ\ \mathit{Done})) \star_{\mathsf{R}} f\\
\{\text{left unit}\} &= \mathit{Pause}\ (\eta_{\mathsf{E}}\ (\mathit{Done}\ v)) \star_{\mathsf{R}} f\\
\{\text{def } \star_{\mathsf{R}}\} &= \mathit{Pause}\ ((\eta_{\mathsf{E}}\ (\mathit{Done}\ v)) \star_{\mathsf{E}} \lambda\kappa.\ \eta_{\mathsf{E}}(\kappa \star_{\mathsf{R}} f))\\
\{\text{left unit}\} &= \mathit{Pause}\ (\eta_{\mathsf{E}}\ ((\mathit{Done}\ v) \star_{\mathsf{R}} f))\\
\{\text{def } \star_{\mathsf{R}}\} &= \mathit{Pause}\ (\eta_{\mathsf{E}}\ (f\ v))
\end{aligned}$$

□

Notational Convention. A notational convention is borrowed from Haskell. The "null bind" of a monad m, $(\mathtt{>>}_{\mathsf{m}})$, is defined as:

$$\begin{aligned}
(\mathtt{>>}_{\mathsf{m}}) &: \mathsf{m}a \rightarrow \mathsf{m}b \rightarrow \mathsf{m}b\\
\varphi \mathtt{>>}_{\mathsf{m}} \gamma &= \varphi \star_{\mathsf{m}} \lambda d.\gamma
\end{aligned}$$

where d is a dummy variable not occurring in γ. The effect of the computation, $\varphi \mathtt{>>}_{\mathsf{m}} \gamma$, is to evaluate φ, ignore the value it produces, and then evaluate γ.

The unblocked semantics for *Add (Val 1) (Val 2)* unfolds as follows:

$$\mathcal{U}[\![\mathit{Add}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2)]\!]$$

$$
\begin{array}{lll|lll}
\mathcal{B}[\![-]\!] : \mathit{Expr} \to \mathsf{R}\,\mathit{Int} & & & \mathcal{U}[\![-]\!] : \mathit{Expr} \to \mathsf{R}\,\mathit{Int} & & \\
\mathcal{B}[\![\mathit{Val}\ i]\!] & = & \mathit{step}(\eta_{\mathsf{E}} i) & \mathcal{U}[\![\mathit{Val}\ i]\!] & = & \mathit{fork}(\mathit{step}(\eta_{\mathsf{E}} i)) \\
\mathcal{B}[\![\mathit{Add}\, e_1\ e_2]\!] & = & \mathcal{B}[\![e_1]\!] \star_{\mathsf{R}} \lambda v_1. & & & \\
 & & \quad \mathcal{B}[\![e_2]\!] \star_{\mathsf{R}} \lambda v_2. & \mathcal{U}[\![\mathit{Add}\, e_1\ e_2]\!] & = & \mathit{fork}\begin{pmatrix} \mathcal{U}[\![e_1]\!] \star_{\mathsf{R}} \lambda v_1. \\ \mathcal{U}[\![e_2]\!] \star_{\mathsf{R}} \lambda v_2. \\ \quad \eta_{\mathsf{R}}(v_1 + v_2) \end{pmatrix} \\
 & & \quad\quad \eta_{\mathsf{R}}(v_1 + v_2) & & & \\
\mathcal{B}[\![\mathit{Seqn}\, e_1\ e_2]\!] & = & \mathcal{B}[\![e_1]\!] \gg_{\mathsf{R}} \mathcal{B}[\![e_2]\!] & \mathcal{U}[\![\mathit{Seqn}\, e_1\ e_2]\!] & = & \mathit{fork}(\mathcal{U}[\![e_1]\!] \gg_{\mathsf{R}} \mathcal{U}[\![e_2]\!]) \\
\mathcal{B}[\![\mathit{Throw}]\!] & = & \mathit{throw}_{\mathsf{R}} & \mathcal{U}[\![\mathit{Throw}]\!] & = & \mathit{fork}\ \mathit{throw}_{\mathsf{R}} \\
\mathcal{B}[\![\mathit{Catch}\, e_1\ e_2]\!] & = & \mathit{catch}_{\mathsf{R}}\,(\mathcal{B}[\![e_1]\!])\,(\mathcal{B}[\![e_2]\!]) & \mathcal{U}[\![\mathit{Catch}\, e_1\ e_2]\!] & = & \mathit{fork}(\mathit{catch}_{\mathsf{R}}\,(\mathcal{U}[\![e_1]\!])\,(\mathcal{U}[\![e_2]\!])) \\
\mathcal{B}[\![\mathit{Block}\, e]\!] & = & \mathcal{B}[\![e]\!] & \mathcal{U}[\![\mathit{Block}\, e]\!] & = & \mathit{fork}\ \mathcal{B}[\![e]\!] \\
\mathcal{B}[\![\mathit{Unblock}\, e]\!] & = & \mathcal{U}[\![e]\!] & \mathcal{U}[\![\mathit{Unblock}\, e]\!] & = & \mathit{fork}\ \mathcal{U}[\![e]\!]
\end{array}
$$

Fig. 4. Monadic Semantics for Hutton & Wright's Language using MMAE Framework

$$
\begin{array}{ll}
 & = \mathit{fork}\,(\mathit{fork}\,(\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.\mathit{fork}\,(\mathit{step}(\eta_{\mathsf{E}}\ 2)) \star_{\mathsf{R}} \lambda v_2.\eta_{\mathsf{R}}(v_1 + v_2)) \\
\{\text{thm 1,def 6}\} & \\
 & = \mathit{fork}\,(\mathit{fork}\,(\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.\mathit{merge}_{\mathsf{R}}\,\{(\mathit{step}(\eta_{\mathsf{E}}\ 2)) \star_{\mathsf{R}} f,\ \mathit{throw}_{\mathsf{R}} \star_{\mathsf{R}} f\}) \\
 & \qquad \text{where } f = \lambda v_2.\eta_{\mathsf{R}}(v_1 + v_2) \\
\{\text{thm 4}\} & = \mathit{fork}\,(\mathit{fork}\,(\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.\mathit{merge}_{\mathsf{R}}\,\{(\mathit{step}(\eta_{\mathsf{E}}\ 2)) \star_{\mathsf{R}} f,\ \mathit{throw}_{\mathsf{R}}\ \}) \\
\{\text{lem 1}\} & = \mathit{fork}\,(\mathit{fork}\,(\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} \lambda v_1.\mathit{merge}_{\mathsf{R}}\,\{\mathit{Pause}\ (\eta_{\mathsf{E}}\ (f\ 2)),\ \mathit{throw}_{\mathsf{R}}\ \}) \\
\{\text{thm 1,def 6}\} & \\
 & = \mathit{fork}\,(\mathit{merge}_{\mathsf{R}}\ \{\ (\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} f', \mathit{throw}_{\mathsf{R}} \star_{\mathsf{R}} f'\}) \\
 & \qquad \text{where } f' = \lambda v_1.\mathit{merge}_{\mathsf{R}}\,\{\mathit{Pause}\,(\eta_{\mathsf{E}}\,(f\,2)), \mathit{throw}_{\mathsf{R}}\} \\
\{\text{thm 4}\} & = \mathit{fork}\,(\mathit{merge}_{\mathsf{R}}\ \{\ (\mathit{step}(\eta_{\mathsf{E}}\ 1)) \star_{\mathsf{R}} f', \mathit{throw}_{\mathsf{R}}\ \}) \\
\{\text{lem 1}\} & = \mathit{fork}\,(\mathit{merge}_{\mathsf{R}}\ \{\ \mathit{Pause}\ (\eta_{\mathsf{E}}\ (f'\ 1), \mathit{throw}_{\mathsf{R}}\ \}) \\
\{\text{def } f,f'\} & = \mathit{fork}\,(\mathit{merge}_{\mathsf{R}}\ \{\ \mathit{Pause}\ (\eta_{\mathsf{E}}\ (\mathit{merge}_{\mathsf{R}}\ \{\mathit{Pause}\ (\eta_{\mathsf{E}}\ 3),\ \mathit{throw}_{\mathsf{R}}\ \})), \mathit{throw}_{\mathsf{R}}\ \}) \\
\{\text{def } \mathit{fork}\} & = \mathit{merge}_{\mathsf{R}} \left\{ \begin{array}{l} \mathit{merge}_{\mathsf{R}} \left\{ \begin{array}{l} \mathit{Pause}\,(\eta_{\mathsf{E}}\,(\mathit{merge}_{\mathsf{R}} \left\{ \begin{array}{l} \mathit{Pause}\,(\eta_{\mathsf{E}}\ (\eta_{\mathsf{R}}\ 3)), \\ \mathit{throw}_{\mathsf{R}} \end{array} \right\})), \\ \mathit{throw}_{\mathsf{R}} \end{array} \right\}, \\ \mathit{throw}_{\mathsf{R}} \end{array} \right\}
\end{array}
$$

The last term in the above evaluation is written in a manner that emphasizes the underlying tree-like structure of denotations in R. The $\mathit{merge}_{\mathsf{R}}$ operators play the rôle of a "tree branch" constructor with $\mathit{throw}_{\mathsf{R}}$ and $\mathit{Pause}\,(\eta_{\mathsf{E}}\ (\eta_{\mathsf{R}}\,3))$ as the "leaves". This term exposes the operational content of $\mathcal{U}[\![\mathit{Add\ (Val\ 1)\ (Val\ 2)}]\!]$. To wit, either an asynchronous exception occurs at *Add* or it doesn't (left-most $\mathit{merge}_{\mathsf{R}}$); or, an asynchronous exception occurs at *(Val 1)* or it doesn't (middle $\mathit{merge}_{\mathsf{R}}$); or, an asynchronous exception occurs at *(Val 2)* or it doesn't (right-most $\mathit{merge}_{\mathsf{R}}$); or, it returns the integer 3. Indeed, this single algebraic term captures all four evaluations from Figure 3. The term displayed in Example (1) in Section 1.1 can be recovered by projecting $\mathcal{U}[\![(\mathit{Add}\,(\mathit{Val}1)\,(\mathit{Val}2))]\!]$ to the E monad via *run*:

$$\mathit{run}\,\mathcal{U}[\![(\mathit{Add}\,(\mathit{Val}1)\,(\mathit{Val}2))]\!]$$

$$= \mathit{run}\ (\mathit{merge}_{\mathsf{R}}\ \{\ \mathit{merge}_{\mathsf{R}}\ \{\ \varphi,\ \mathit{throw}_{\mathsf{R}}\ \},\ \mathit{throw}_{\mathsf{R}}\ \})$$
$$\text{where } \varphi = \mathit{Pause}(\eta_{\mathsf{E}}(\mathit{merge}_{\mathsf{R}}\{\mathit{Pause}(\eta_{\mathsf{E}}(\eta_{\mathsf{R}}\,3)), \mathit{throw}_{\mathsf{R}}\}))$$
$$\{\text{thm 2}\} \quad = \mathit{merge}_{\mathsf{E}}\ \{\ \mathit{run}\ (\mathit{merge}_{\mathsf{R}}\ \{\ \varphi,\ \mathit{throw}_{\mathsf{R}}\ \}),\ \mathit{run}\ \mathit{throw}_{\mathsf{R}}\ \}$$
$$\{\text{thms 2,5}\} = \mathit{merge}_{\mathsf{E}}\ \{\ \mathit{merge}_{\mathsf{E}}\ \{\ \mathit{run}\,\varphi,\ \mathit{run}\ \mathit{throw}_{\mathsf{R}}\ \},\ \mathit{throw}_{\mathsf{E}}\ \}$$

By repeated applications of Theorems 2 and 5, the definition of *run*, and left unit monad law, $\mathit{run}\,\varphi\ =\ \mathit{merge}_{\mathsf{E}}\{\eta_{\mathsf{E}}3, \mathit{throw}_{\mathsf{E}}\}$. Continuing:

$$\{\text{thm 5}\} \quad = \mathit{merge}_{\mathsf{E}}\ \{\ \mathit{merge}_{\mathsf{E}}\ \{\ \mathit{merge}_{\mathsf{E}}\ \{\ \eta_{\mathsf{E}}\ 3,\ \mathit{throw}_{\mathsf{E}}\ \}\ ,\ \mathit{throw}_{\mathsf{E}}\ \},\ \mathit{throw}_{\mathsf{E}}\ \}$$

Theorem 6 establishes an equivalence between Hutton and Wright's natural semantics (Figure 2) and the MMAE semantics in Figure 4. Due to the presence of non-determinism in both semantic specifications, there is more than one possible outcome for any expression. The theorem states that, for any expression e, a particular outcome (i.e., $(\mathit{Ok}\ v)$ or *Error*) may occur in the set of all possible outcomes for e (i.e., $\mathit{run}\ (\mathcal{U}[\![e]\!])$ or $\mathit{run}\ (\mathcal{B}[\![e]\!])$) if, and only if, the corresponding term (respectively, $(\mathit{Val}\ v)$ or *Throw*) can be reached via the natural semantics. The proof of Theorem 6 is straightforward and long, so rather than including it here, it has been made available online [8].

Theorem 6 (Semantic Equivalence). *For any expression e:*

$$(\mathit{Ok}\ v) \in \mathit{run}(\mathcal{U}[\![e]\!]) \text{ iff } e \Downarrow^{U} (\mathit{Val}\ v) \qquad (\mathit{Ok}\ v) \in \mathit{run}(\mathcal{B}[\![e]\!]) \text{ iff } e \Downarrow^{B} (\mathit{Val}\ v)$$
$$\mathit{Error} \in \mathit{run}(\mathcal{U}[\![e]\!]) \text{ iff } e \Downarrow^{U} \mathit{Throw} \qquad \mathit{Error} \in \mathit{run}(\mathcal{B}[\![e]\!]) \text{ iff } e \Downarrow^{B} \mathit{Throw}$$

5 The MMAE as a Programming Model

This section demonstrates the use of the MMAE in the functional programming of applications with asynchronous behaviors. In particular, the extension of synchronous concurrent kernels with asynchronous, interrupt-driven features is described. The approach elaborated here develops along the same lines as modular interpreters [16] and compilers [11]: an existing specification is extended with a "building block" consisting of a monadic "module" and related operators. By applying the interrupts building block to a kernel without interrupts (middle, Figure 1), a kernel with interrupt-driven input is produced with virtually no other changes.

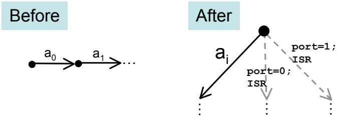
Recent research has demonstrated how kernels with a broad range of OS behaviors (e.g., message-passing, synchronization, forking, etc.) may be formulated in terms of resumption monads [9]. This section outlines the extension of such resumption-monadic kernels with asynchronous behaviors. The presentation is maintained at a high-level in order to expose the simplicity of the asynchronous extension. Interested readers may refer to the code base [8] or to Harrison [9] for the more details concerning kernel construction. The presentation here will frequently appeal to the reader's intuition in order to remain at a high-level.

The kernel and its enhanced functionality are described in Figure 1. The enhanced kernel is kept as simple as possible. It contains no mechanism for blocking interrupts and there is only one interrupt service routine (ISR) connecting the kernel to the input port. Such enhancements are, however, quite simple to make.

Both kernels (Figure 1, middle and right) support synchronous message-passing, maintaining a message queue, msgQ, to hold messages "in-flight". The extended kernel (Figure 1, right) adds an interrupt-driven serial-to-parallel input port. When a bit is ready at the port, an interrupt is thrown, causing the ISR to run. This ISR inserts the new bit into its local bit queue, bitQ. If there are eight bits in bitQ, then it assembles them into a byte and inserts this new datagram into msgQ, thereby allowing user threads to receive messages through the port. Pseudocode for the ISR is:

```
insertQ(port,bitQ);
if (length(bitQ) > 7) {
     make_datagram(x);
     insertQ(x,msgQ);
}
```

Here, port is the bit register in the port that holds the most recent input bit.



A resumption-monadic kernel is an R ()-valued function that takes as input a list of ready threads. In other words, *kernel* ⟨ready⟩ : R (), where ⟨ready⟩ is a Haskell representation of the thread list. The output of the kernel is, then, a (possibly infinite) R-computation consisting of a sequence of atomic actions a_i executed in succession as shown in the inset figure (top, Before). These actions are drawn from user threads in ⟨ready⟩ that have been woven together into a schedule by the kernel. The notion of computation associated with R will also be changed as outlined in Section 5.1. The key to "turning on" asynchronous interrupts is to view each break-point "•" in the inset figure (top, Before) as a branching point. This branch contains three possible histories (inset figure (bottom, After)): one history in which the scheduled atom a_i executes, another where the ISR executes with the input bit 0, and a third where the ISR executes with the input bit 1. The branching can be achieved as in Section 4 with the use of an appropriate *merge* operator. The operation of the enhanced kernel is a computation that elaborates a tree.

There are only two changes necessary to introduce asynchronous interrupts to the synchronous kernel. The first is simply to refine the monad by incorporating non-determinism and state for the port device into the monad R. The second modification is to apply the appropriate *resumption map* to (*kernel* ⟨ready⟩). Resumption maps arise from the structure of R in an analogous manner to the way the familiar list function, $map : (a \to b) \to [a] \to [b]$, operates over lists. The resumption map used here is discussed below in Section 5.2.

5.1 Monad Hierarchy

Two monads associated with the kernel are the kernel monad, K, and the concurrency monad R. K encapsulates the imperative aspects of the system. These are kernel-level operations that update system data structures (*Sys*) and user-level operations that update the user-level storage (*Sto*). K is constructed with the application of two state monad transformers to the identity monad, Id.

$$\mathsf{K} = \mathit{StateT}\ \mathit{Sys}\ (\mathit{StateT}\ \mathit{Sto}\ \mathsf{Id})$$

The concurrency monad R is then built on top of K with an application of the resumption monad transformer [22]:

$$\begin{aligned} &\mathsf{R} && = \mathsf{ResT}\ \ \mathsf{K} \\ &\textbf{data}\ \mathsf{ResT}\ \mathsf{m}\ a && = \mathit{Done}\ a \mid \mathit{Pause}\ (\mathsf{m}\ (\mathit{ResT}\ \mathsf{m}\ a)) \end{aligned}$$

Note that this monad transformer provides an alternative means of defining the R monad from Section 4: $\mathsf{R}\ =\ \mathsf{ResT}\ \mathsf{E}$.

5.2 Resumption Maps

There is a close analogy between computations in R and lists and streams. A non-trivial computation in R consists of a (possibly infinite) sequence of atomic actions in K separated by *Pause* constructors. Definition 7 specifies a resumption map that takes a function h and an R-computation γ and transforms γ one atom at a time:

Definition 7 (Map on R)

$$\begin{aligned} &\mathit{map}_{\mathsf{R}} && : (\mathsf{R}\ a \rightarrow \mathsf{K}(\mathsf{R}\ a)) \rightarrow \mathsf{R}\ a \rightarrow \mathsf{R}\ a \\ &\mathit{map}_{\mathsf{R}}\ h\ (\mathit{Done}\ v) && = \mathit{Done}\ v \\ &\mathit{map}_{\mathsf{R}}\ h\ (\mathit{Pause}\ \varphi) && = \mathit{Pause}\ (h\ (\mathit{Pause}\ \varphi) \star_{\mathsf{K}} (\eta_{\mathsf{K}} \circ \mathit{map}_{\mathsf{R}}\ h)) \end{aligned}$$

5.3 Adding Asynchronicity in Two Steps

Now, the stage is set to add asynchronicity to the synchronous kernel. The first step provides additional computational resources to K and the second uses these resources to "turn on" asynchronicity.

First Step: Adding Non-determinism and a Device to K. The first change to the synchronous kernel is to add non-determinism and storage for the port device to the K and R monads. This is accomplished by replacing Id with N in the definition of K and with another application of the state monad transformer:

$$\begin{aligned} \mathsf{K} &= \mathit{StateT}\,\mathit{Dev}\,(\mathit{StateT}\,\mathit{Sys}\,(\mathit{StateT}\,\mathit{Sto}\,\mathsf{N})) \\ \mathit{Dev} &= [\mathit{Bit}] \times \mathit{Bit} \end{aligned}$$

The N is defined exactly as in Section 4. A device state is a pair, $(\mathit{bitQ}, \mathit{new})$, consisting of a queue of bits, bitQ, and a single bit denoting the current input to the port. The K monad has separate operators for reading and writing the *Dev*, *Sys* and *Sto* states along with a merge operation, $\mathit{merge}_{\mathsf{K}}\ :\ \mathcal{P}_{\mathrm{fin}}(\mathsf{K}\ a) \rightarrow \mathsf{K}\ a$, that is defined in terms of $\mathit{merge}_{\mathsf{N}}$. Note also that R is also affected by this refinement of K, but that the text of its definition is not (i.e., it is still the case that $\mathsf{R}\ =\ \mathsf{ResT}\,\mathsf{K}$). Details may be found in the code base [8].

Second Step: Asynchronicity via Resumption Mapping. First, a resumption mapping is defined that creates a branch with three possible histories.

$$\mathit{branches} \;:\; \mathsf{R}\,a \rightarrow \mathsf{K}(\mathsf{R}\,\mathsf{a})$$

$$\mathit{branches}\ (\mathit{Pause}\ \varphi) = \mathit{merge}_{\mathsf{K}} \left\{ \begin{array}{l} \varphi, \\ (\mathtt{port}{=}0\,;\mathtt{ISR}) \mathbin{\texttt{>>}_{\mathsf{K}}} \varphi, \\ (\mathtt{port}{=}1\,;\mathtt{ISR}) \mathbin{\texttt{>>}_{\mathsf{K}}} \varphi \end{array} \right\} \begin{array}{l} (1) \\ (2) \\ (3) \end{array}$$

Here, ($\mathtt{port}{=}0$; ISR) represents a K-computation that performs the indicated actions. History (1) is one in which no interrupt occurs. In history (2), an interrupt occurs when the input bit of the port is set to 0 and the ISR is then executed. Finally, the interrupted thread, φ, is executed. History (3) is analogous to (2).

Flipping the Port Off and On. It is now a simple matter to turn the port off and on. To operate the kernel with the port off, execute: *kernel* ⟨ready⟩. To operate the kernel with the port on, execute: $\mathit{map}_{\mathsf{R}}$ *branches* (*kernel* ⟨ready⟩). The two sample system runs from Figure 1 are repeated below. There are two threads, a producer and a consumer. The producer thread broadcasts integer messages starting at 1001 and incrementing successively with each broadcast. The consumer thread consumes these messages. When either thread performs a broadcast or receive, the instrumented kernel prints it out. Also, when the ISR creates a new datagram, it also announces the fact.

Port off: *kernel* ⟨ready⟩ Port on: $\mathit{map}_{\mathsf{R}}$ *branches* (*kernel* ⟨ready⟩)

```
Haskell> producer_consumer     Haskell> producer_consumer
broadcasting  1001             broadcasting  1001
broadcasting  1002                new datagram: 179
         receiving  1001       broadcasting  1002
broadcasting  1003                      receiving  179
         receiving  1002 ...      new datagram: 204 ...
```

6 Related Work

Implementations of threaded applications use monads to structure the threaded code. The Haskell libraries for concurrency use IO level primitives to provide a simple and robust user interface [23]. There have been other threading implementations that use other monads, including [14], which uses a continuation passing style monad, giving very high numbers of effectively concurrent threads.

There have been a number of efforts to model the concurrency provided by the Haskell IO monad. One successful effort is reported by Swierstra and Altenkirch [29], where they model the concurrency inside the Haskell IO monad using a small stepping scheduler combined with the QuickCheck framework [3] to provide random interleaving of pseudo-threads. In Dowse and Butterfield [5], an operational model of shared state, input/output and deterministic concurrency is provided. One fundamental difference between these models and our work is granularity. Both of these models assume each computation can not be interrupted, and threads only are scheduled at the IO interaction points. Our model allows interrupts to happen inside computation, capturing the pre-emptive implementations of concurrency described in [17,24], and provided by the Glasgow Haskell compiler.

The language modeled with MMAE in Section 4 was adopted from recent research on the operational semantics of asynchronous interrupts [13]. That research also identified an error within a published operational semantics for interrupts in Haskell [17]. Morris and Tyrrell [20] offer a comprehensive mathematical theory of nondeterminacy, encompassing both its demonic and angelic forms. Their approach adds operators for angelic and demonic choice to each type in a specification or programming language. They present refinement calculus axioms for demonic and angelic nondeterminacy and define a domain model to demonstrate the soundness of these axioms. One open question is whether there exists common ground between their work and the monadic approach presented here. In particular, if the domain theoretic constructions underlying Morris and Tyrrell's semantics may be expressed usefully as monads.

7 Future Work and Conclusions

There has been considerable interest of late in using monads to structure system software [7,14], model it formally [9], and to enforce and verify its security [10]. None of this previous research contains an model of asynchronous behavior and the MMAE was developed as a means of rectifying this situation. The present work is part of an ongoing effort to use monads as an organizing principle for formally specifying kernels. Currently, the direct compilation of monadic kernel designs to both stock and special purpose hardware is being investigated. This research agenda considers these monadic specifications as a source language for semantics-directed synthesis of high-assurance kernels. The ultimate goal of this agenda is to produce high-confidence systems automatically.

What possible applications might our specification methodology have? We have shown is that it is possible to take a MMS constructed executable specification, add a non-deterministic monad giving an accurate monadic based semantics for interrupts. Here we briefly sketch a possible way of using our semantics in practical applications.

Non-determinism is used to model *all* possible outcomes. In an implementation, a lower-level model could replace the use of non-determinism with the exception monad. The exception monad would be used to communicate an interrupt in exactly the same way it would be used to communicate an exception. The fork primitive in our semantics would be implemented in a lower-level model using a simple poll of an interrupt flag. Specifically, in our kernel example, we might use this MMS for K in the lower-level model.

$$\begin{aligned}\mathsf{K} &= \mathit{StateT}\ \mathit{Dev}\ (\mathit{StateT}\ \mathit{Sys}\ (\mathit{StateT}\ \mathit{Sto}\ \mathit{Maybe}))\\ &\quad \text{— } \mathsf{K} = \text{Dev} + \text{Sys} + \text{User} + \text{Exception}\end{aligned}$$

This gives us a basis for an efficient implementation from our monadic primitives. We use the code locations that perform interrupt polling in the lower-level model as *safe-points*. If an interrupt request is made of a specific thread, this thread can be single-stepped to any safe-point, a common implementation technique for interruptible code.

Thus as an implementation path we have our high-level model using nondeterminism, a low-level model using exceptions to simulate interrupts, and a possible implementation with precisely specified safely interruptible points that can be implemented using direct hardware support. This chaining of specifications toward implementation is a long standing problem in building nondeterministic systems, and this research provides an important step toward a more formal methodology of software development of such systems.

The MMAE confronts one of the "impurities" of the infamous "Awkward Squad" [23]: language features considered difficult to accommodate within a pure, functional setting—concurrency, state, and input/output. Most of these impurities have been handled individually via various monadic constructions (consider the manifestly incomplete list [18,22,25]) with the exception of asynchronous exceptions. The current approach combines these individual constructions into a single *layered* monad—i.e., a monad created from monadic building blocks known as monad transformers [6,15]. While it is not the intention of the current work to model either the Haskell *IO* monad or Concurrent Haskell, it is believed that the techniques and structures presented here provide some important building blocks for such models.

References

1. de Bakker, J.W.: Mathematical Theory of Program Correctness. International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1980)
2. Chatterjee, K., Ma, D., Majumdar, R., Zhao, T., Henzinger, T.A., Palsberg, J.: Stack size analysis for interrupt-driven programs. Inf. Comput. 194(2), 144–174 (2004)
3. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: Proc. of International Conference on Functional Programming (ICFP), ACM SIGPLAN (2000)
4. Dijkstra, E.W.: My recollections of operating system design. SIGOPS Oper. Syst. Rev. 39(2), 4–40 (2005)
5. Dowse, M., Butterfield, A.: Modelling deterministic concurrent I/O. In: ICFP 2006: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming, pp. 148–159. ACM, New York (2006)
6. Espinosa, D.: Semantic Lego. PhD thesis, Columbia University (1995)
7. Hallgren, T., Jones, M.P., Leslie, R., Tolmach, A.: A principled approach to operating system construction in Haskell. In: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP05), pp. 116–128. ACM Press, New York (2005)
8. Harrison, W.: The Asynchronous Exceptions As An Effect Codebase, `www.cs.missouri.edu/~harrisonwl/AsynchronousExceptions`
9. Harrison, W.: The Essence of Multitasking. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 158–172. Springer, Heidelberg (2006)

10. Harrison, W., Hook, J.: Achieving information flow security through monadic control of effects. Invited submission to: Journal of Computer Security, 46 (accepted, 2008)
11. Harrison, W., Kamin, S.: Metacomputation-based compiler architecture. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 213–229. Springer, Heidelberg (2000)
12. Hughes, J., O'Donnell, J.: Nondeterministic functional programming with sets. In: Proceedings of the 1990 Banf Conference on Higher Order Reasoning (1990)
13. Hutton, G., Wright, J.: What is the Meaning of These Constant Interruptions? Journal of Functional Programming 17(6), 777–792 (2007)
14. Li, P., Zdancewic, S.: Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In: PLDI 2007: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, pp. 189–199. ACM Press, New York (2007)
15. Liang, S.: Modular Monadic Semantics and Compilation. PhD thesis, Yale University (1998)
16. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 333–343. ACM Press, New York (1995)
17. Marlow, S., Peyton Jones, S., Moran, A., Reppy, J.: Asynchronous exceptions in Haskell. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 274–285 (2001)
18. Moggi, E.: An Abstract View of Programming Languages. Technical Report ECS-LFCS-90-113, Department of Computer Science, Edinburgh University (1990)
19. Moggi, E.: Notions of computation and monads. Inf. Comput. 93(1), 55–92 (1991)
20. Morris, J.M., Tyrrell, M.: Terms with unbounded demonic and angelic nondeterminacy. Sci. Comput. Program. 65(2), 159–172 (2007)
21. Palsberg, J., Ma, D.: A Typed Interrupt Calculus. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 291–310. Springer, Heidelberg (2002)
22. Papaspyrou, N.S.: A Resumption Monad Transformer and its Applications in the Semantics of Concurrency. In: Proceedings of the 3rd Panhellenic Logic Symposium (2001); Expanded version available as a tech. report from the author by request
23. Peyton Jones, S.: Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell. In: Engineering Theories of Software Construction. NATO Science Series, vol. III 180, pp. 47–96. IOS Press, Amsterdam (2000)
24. Peyton Jones, S., Reid, A., Hoare, C.A.R., Marlow, S., Henderson, F.: A semantics for imprecise exceptions. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, pp. 25–36 (May 1999)
25. Peyton Jones, S., Wadler, P.: Imperative functional programming. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 71–84. ACM Press, New York (1993)
26. Plotkin, G.D.: A Powerdomain Construction. SIAM Journal of Computation 5(3), 452–487 (1976)
27. Schmidt, D.A.: Denotational Semantics: A Methodology for Language Development. Allyn and Bacon, Boston (1986)
28. Smyth, M.B.: Powerdomains. Journal of Computer and System Sciences 16(1), 23–36 (1978)

29. Swierstra, W., Altenkirch, T.: Beauty in the beast. In: Haskell 2007: Proceedings of the ACM SIGPLAN workshop on Haskell workshop, pp. 25–36. ACM, New York (2007)
30. Tolmach, A., Antoy, S.: A monadic semantics for core curry. In: Proceedings of the 12th International Workshop on Functional and (Constraint) Logic Programming (June 2003)
31. Wadler, P.: The essence of functional programming. In: Proceedings of the 19th Symposium on Principles of Programming Languages (POPL), pp. 1–14. ACM Press, New York (1992)

The Böhm–Jacopini Theorem Is False, Propositionally

Dexter Kozen and Wei-Lung Dustin Tseng

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501, USA
{kozen,wdtseng}@cs.cornell.edu

Abstract. The Böhm–Jacopini theorem (Böhm and Jacopini, 1966) is a classical result of program schematology. It states that any deterministic flowchart program is equivalent to a while program. The theorem is usually formulated at the first-order interpreted or first-order uninterpreted (schematic) level, because the construction requires the introduction of auxiliary variables. Ashcroft and Manna (1972) and Kosaraju (1973) showed that this is unavoidable. As observed by a number of authors, a slightly more powerful structured programming construct, namely loop programs with multi-level breaks, is sufficient to represent all deterministic flowcharts without introducing auxiliary variables. Kosaraju (1973) established a strict hierarchy determined by the maximum depth of nesting allowed. In this paper we give a purely propositional account of these results. We reformulate the problems at the propositional level in terms of automata on guarded strings, the automata-theoretic counterpart to Kleene algebra with tests. Whereas the classical approaches do not distinguish between first-order and propositional levels of abstraction, we find that the purely propositional formulation allows a more streamlined mathematical treatment, using algebraic and topological concepts such as bisimulation and coinduction. Using these tools, we can give more mathematically rigorous formulations and simpler and more revealing proofs.

1 Introduction

Program schematology was one of the earliest topics in the mathematics of computing. A central problem that has been well studied over the years is that of transforming an unstructured flowgraph to structured form. A seminal result in this area is the Böhm–Jacopini theorem [2], which states that any deterministic flowchart program is equivalent to a while program. This classical theorem has reappeared in many contexts and has been reproved by many different methods. There are dozens of references on this topic; a few commonly cited ones are [1,8,9,10,11].

The Böhm–Jacopini theorem is usually formulated at the first-order interpreted or first-order uninterpreted (schematic) level, as was most early work in program schematology. The first-order formulation allows the introduction of

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 177–192, 2008.

auxiliary individual or Boolean variables to preserve information during the restructuring. This is an essential ingredient of the Böhm–Jacopini construction, and they asked whether it was strictly necessary. This question was answered affirmatively by Ashcroft and Manna [1] and Kosaraju [4].

Böhm and Jacopini's question, and Ashcroft and Manna and Kosaraju's solutions, were phrased in terms of the necessity of introducing auxiliary variables. This view is repeated in subsequent works, e.g. [8]. However, this is not really the best way to phrase the question. What was really shown was that a purely propositional formulation of the Böhm–Jacopini theorem is false: there is a deterministic propositional flowchart that is not equivalent to any propositional while program. This result is implicit in [1,4], although it was not stated this way.

As observed by a number of authors (e.g. [4,9]), a slightly more powerful structured programming construct, namely loops with multi-level breaks, is sufficient to represent all deterministic flowcharts without introducing auxiliary variables. Kosaraju [4] established a strict hierarchy based on the levels of the multi-level breaks that are allowed. He showed that for any $n \geq 1$, there exists a loop program with break m for $m \leq n$ that is not equivalent to any loop program with break m for $m \leq n - 1$. Again, however, these results were formulated and proved at the first-order interpreted level, despite the fact that they are essentially propositional.

Inexpressibility proofs such as those of [1,4] that reason in terms of a particular first-order interpretation may appear contrived, because any number of other interpretations could serve the same purpose. One runs the risk of obscuring the underlying principles at work by the details of the particular construction, which are largely irrelevant. Moreover, the classical approach to program schematology relies heavily on graphs and combinatorial graph restructuring operations, which can be difficult to reason about formally.

Automata on guarded strings, the automata-theoretic counterpart of Kleene algebra with tests (KAT), provide an opportunity to reset the theory of program schemes on more rigorous algebraic foundations. We have found that a purely propositional, automata-theoretic reformulation of some of the questions mentioned above allows a more streamlined treatment. Algebraic and topological concepts such as bisimulation and coinduction, absent in earlier treatments, predominate here. We feel that the resulting proofs are simpler, more rigorous, and more revealing of the underlying principles at work.

This paper is organized as follows. In Section 2, we briefly discuss the differences between propositional and first-order formulations, and recall the basic definitions regarding guarded strings and automata with tests. We introduce a special restricted form of automata with tests, which we call *strictly deterministic*, corresponding to deterministic flowchart schemes. We argue that every deterministic flowchart scheme is semantically equivalent to a strictly deterministic automaton. Also in Section 2, we define bisimulation for strictly deterministic automata and mention several more or less standard results regarding bisimulations. We also recall the definition of the structured programming constructs for while and loop programs and their semantics. Many proofs in this section are quite routine and are omitted.

Using these tools, we then give a purely propositional account of three known results: that the Böhm–Jacopini theorem is false at the propositional level, that loop programs with multi-level breaks are sufficient to represent all deterministic flowcharts, and that the Kosaraju hierarchy is strict. These results are proved in Sections 3, 4, and 5, respectively. We conclude with some open problems in Section 6.

2 Preliminaries

2.1 Propositional vs. First-Order Logic

The notions of functions on a domain and variables ranging over that domain are inherent in first-order logic, but are not present in propositional logic. Whereas we may consider a variable assignment $x := t$ as a primitive action in first-order program logic, a primitive action in propositional program logic is just a symbol. Since previous constructions establishing the Böhm–Jacopini theorem require the introduction of extra variables, they cannot be formalized at the propositional level of abstraction.

In this paper, we model propositional deterministic flowcharts and structured programs as strictly deterministic automata with tests, and we model program executions as guarded strings (both defined below). If desired, our propositional formulation can be extended with a first order interpretation. A subtle but important point is that all behaviors of a propositional program have a first-order realization; that is, given any guarded string representing a possible execution of a propositional program, there is a first-order interpretation that realizes that execution.

2.2 Guarded Strings

Guarded strings were introduced in [3]. They model program executions propositionally. Let Σ be a finite set of *action symbols* and T a finite set of *test symbols* disjoint from Σ. The symbols T generate a free Boolean algebra B; elements of B are called *tests*. An *atom* is a minimal nonzero element of B. The set of atoms is denoted At. The elements of At can be regarded either as conjunctions of literals of T (elements of T or their negations) or as truth assignments to T, thus $|\mathsf{At}| = 2^{|T|}$. We write $p, q, p_0, \ldots$ for elements of Σ and $\alpha, \beta, \alpha_0, \ldots$ for elements of At. A *guarded string* is a finite alternating sequence of atoms and actions, beginning and ending with an atom; that is, an element of $(\mathsf{At} \cdot \Sigma)^* \cdot \mathsf{At}$. In other words, guarded strings represent the join-irreducible elements of the free KAT on generators Σ and T. Intuitively, a guarded string records the sequence of primitive actions taken by a program and the tests that are true between any two successive primitive actions.

We will also consider infinite guarded strings, which are members of $(\mathsf{At} \cdot \Sigma)^\omega$, but will always qualify with the adjective "infinite" when doing so.

2.3 Automata with Tests

Automata with tests, also known as automata on guarded strings, were studied in [5]. They are the automata-theoretic counterpart to Kleene algebra with tests (KAT). In the formalism of [5], they have two types of transitions, *action transitions* and *test transitions*, and operate over guarded strings. An ordinary automaton with null transitions is just an automaton with tests over the two-element Boolean algebra. Many of the constructions of ordinary finite-state automata, such as determinization and state minimization, extend readily to automata with tests. In particular, there is a version of Kleene's theorem showing that these automata are equivalent in expressive power to expressions in the language of KAT. See [5] for a more detailed introduction.

2.4 Strictly Deterministic Automata

For the purposes of this paper, we will only need to consider a limited class of automata with tests corresponding to deterministic propositional flowchart schemes. Since actions are uniquely determined, we may elide the action states to obtain what we call a *strictly deterministic automaton.*

Intuitively, a strictly deterministic automaton operates by starting in its start state and scanning a sequence of atoms, which we can view as provided by an external agent. For each atom in succession, the automaton responds deterministically either by emitting an action symbol and moving to a new state, by halting, or by failing, according to its transition function.

Formally, a *strictly deterministic automaton* over Σ and T is a tuple

$$M = (Q,\, \delta,\, \mathsf{start}),$$

where Q is a (possibly infinite) set of *states*, $\mathsf{start} \in Q$ is the *start state*, and δ is a *transition function*

$$\delta : Q \times \mathsf{At} \to (\Sigma \times Q) + \{\mathsf{halt}, \mathsf{fail}\},$$

where $+$ denotes disjoint (marked) union. The elements halt and fail are not states, but universal constants used by an automaton to represent halting and failing, respectively. The components Q, δ, and start may be adorned with the subscript M where necessary to distinguish between automata. States are denoted by $s, t, u, v, \ldots$.

A *trace* in M is a finite or infinite alternating sequence of states and atoms specifying a path through M. Formally, a *trace* is a sequence σ in

$$(Q \cdot \mathsf{At})^* \cdot Q \;+\; (Q \cdot \mathsf{At})^\omega$$

such that for every substring of σ of the form $u\alpha v$, $\delta(u, \alpha) = (p, v)$ for some $p \in \Sigma$. The first state of σ is denoted $\mathsf{first}\,\sigma$ and the last state (if it exists) is denoted $\mathsf{last}\,\sigma$.

Given a state s and an infinite sequence of atoms σ, there is a unique finite or infinite trace $\mathsf{tr}(s, \sigma)$ determined intuitively by starting in state s and running the automaton, making choices at each successive state according to the next atom in the sequence σ as determined by the transition function δ. The trace is finite iff M halts or fails along the way, even though σ is infinite. Formally, the map

$$\mathsf{tr} : Q \times \mathsf{At}^\omega \to (Q \cdot \mathsf{At})^* \cdot Q \ + \ (Q \cdot \mathsf{At})^\omega$$

is defined coinductively as follows:

$$\mathsf{tr}(s, \alpha\,\sigma) \overset{\text{def}}{=} \begin{cases} s \cdot \alpha \cdot \mathsf{tr}(t, \sigma) & \text{if } \delta(s, \alpha) = (p, t) \\ s & \text{if } \delta(s, \alpha) \in \{\mathsf{halt}, \mathsf{fail}\}. \end{cases}$$

This definition determines $\mathsf{tr}(s, \sigma)$ uniquely for all $s \in Q$ and $\sigma \in \mathsf{At}^\omega$.

A similar definition holds for guarded strings. Here we also allow infinite guarded strings as well as finite ones. Given a starting state s and an infinite sequence of atoms σ, there is at most one finite or infinite guarded string $\mathsf{gs}(s, \sigma)$ obtained by running the automaton starting in state s. Formally, the partial map

$$\mathsf{gs} : Q \times \mathsf{At}^\omega \to (\mathsf{At} \cdot \Sigma)^* \cdot \mathsf{At} \ + \ (\mathsf{At} \cdot \Sigma)^\omega$$

is defined coinductively as follows:

$$\mathsf{gs}(s, \alpha\,\sigma) \overset{\text{def}}{=} \begin{cases} \alpha \cdot p \cdot \mathsf{gs}(t, \sigma) & \text{if } \delta(s, \alpha) = (p, t) \\ \alpha & \text{if } \delta(s, \alpha) = \mathsf{halt} \\ \text{undefined} & \text{if } \delta(s, \alpha) = \mathsf{fail}. \end{cases}$$

As with traces, $\mathsf{gs}(s, \sigma)$ is uniquely determined for all $s \in Q$ and $\sigma \in A^\omega$.

The set of (finite) guarded strings represented by the automaton M is

$$\mathsf{GS}(M) \overset{\text{def}}{=} \{\mathsf{gs}(\mathsf{start}_M, \sigma) \mid \sigma \in \mathsf{At}^\omega\} \cap (\mathsf{At} \cdot \Sigma)^* \cdot \mathsf{At}.$$

Two automata are considered semantically equivalent if they represent the same set of finite guarded strings.

The transition function δ determines a map

$$\widehat{\delta} : Q \times \mathsf{At}^* \to Q + \{\mathsf{halt}, \mathsf{fail}\}$$

defined inductively as follows:

$$\widehat{\delta}(s, \sigma) \overset{\text{def}}{=} \begin{cases} s, & \text{if } \sigma = \varepsilon \\ \widehat{\delta}(t, \tau), & \text{if } \sigma = \alpha\tau \text{ and } \delta(s, \alpha) = (p, t) \\ \mathsf{halt}, & \text{if } \sigma = \alpha\tau \text{ and } \delta(s, \alpha) = \mathsf{halt} \\ \mathsf{fail}, & \text{if } \sigma = \alpha\tau \text{ and } \delta(s, \alpha) = \mathsf{fail}. \end{cases}$$

This is either the state that the machine is in after scanning σ starting in state s, or halt or fail if the machine halts or fails while scanning σ starting in state s.

2.5 Bisimulation

Let M and N be two strictly deterministic automata. A *bisimulation* between M and N is a binary relation $\equiv$ between Q_M and Q_N such that

(i) $\mathsf{start}_M \equiv \mathsf{start}_N$, and
(ii) if $s \in Q_M$, $t \in Q_N$, and $s \equiv t$, then for all $\alpha \in \mathsf{At}$,
  (a) $\delta_M(s,\alpha) = \mathsf{halt}$ iff $\delta_N(t,\alpha) = \mathsf{halt}$;
  (b) $\delta_M(s,\alpha) = \mathsf{fail}$ iff $\delta_N(t,\alpha) = \mathsf{fail}$; and
  (c) if $\delta_M(s,\alpha) = (p,s')$ and $\delta_N(t,\alpha) = (q,t')$, then $p = q$ and $s' \equiv t'$.

M and N are said to be *bisimilar* if there exists a bisimulation between M and N. An *autobisimulation* is a bisimulation between M and itself.

Bisimulations are closed under relational composition and arbitrary union, and the identity relation on an automaton is an autobisimulation. Thus the reflexive transitive closure of an autobisimulation is again an autobisimulation. Moreover, if two automata are bisimilar, then there is a unique maximum bisimulation between them, namely the union of all bisimulations between them. We provide three lemmas regarding bisimulation.

To show $\mathsf{GS}(M) = \mathsf{GS}(N)$, it suffices to show that M and N are bisimilar:

Lemma 1. *If M and N are bisimilar, then $\mathsf{GS}(M) = \mathsf{GS}(N)$.*

More interestingly, under certain mild conditions, the converse holds as well:

Lemma 2. *Suppose $\mathsf{GS}(M) = \mathsf{GS}(N)$, M does not contain a fail transition, and halt is accessible from every state of M that is accessible from start_M. Then M and N are bisimilar.*

Proof. For $s \in Q_M$ and $t \in Q_N$, set

$$s \equiv t \overset{\text{def}}{\Longleftrightarrow} \forall\sigma \in \mathsf{At}^\omega\ \mathsf{gs}_M(s,\sigma) = \mathsf{gs}_N(t,\sigma).$$

We show that $\equiv$ is a bisimulation. If $s \equiv t$, then for all $\alpha \in \mathsf{At}$ and $\sigma \in \mathsf{At}^\omega$,

$$\delta_M(s,\alpha) = \mathsf{halt} \Leftrightarrow \mathsf{gs}_M(s,\alpha\sigma) = \alpha \ \Leftrightarrow\ \mathsf{gs}_N(t,\alpha\sigma) = \alpha \ \Leftrightarrow\ \delta_N(t,\alpha) = \mathsf{halt}$$

and similarly for fail, and if $\delta_M(s,\alpha) = (p,s')$ and $\delta_N(t,\alpha) = (q,t')$, then

$$\alpha \cdot p \cdot \mathsf{gs}_M(s',\sigma) = \mathsf{gs}_M(s,\alpha\sigma) \ =\ \mathsf{gs}_N(t,\alpha\sigma) \ =\ \alpha \cdot q \cdot \mathsf{gs}_N(t',\sigma),$$

thus $p = q$ and $\mathsf{gs}_M(s',\sigma) = \mathsf{gs}_N(t',\sigma)$. As σ was arbitrary, $s' \equiv t'$. This establishes property (ii) of bisimulation.

It remains to show property (i); that is, $\mathsf{start}_M \equiv \mathsf{start}_N$, or in other words, $\mathsf{gs}_M(\mathsf{start}_M,\sigma) = \mathsf{gs}_N(\mathsf{start}_N,\sigma)$ for all σ. By assumption, $\mathsf{GS}(M) = \mathsf{GS}(N)$, so if $\mathsf{gs}_M(\mathsf{start}_M,\sigma)$ is finite, then so is $\mathsf{gs}_N(\mathsf{start}_N,\sigma)$ and they are equal. Thus the functions

$$\mathsf{gs}_M(\mathsf{start}_M,-) : \mathsf{At}^\omega \to (\mathsf{At}\cdot\Sigma)^* \cdot \mathsf{At} + (\mathsf{At}\cdot\Sigma)^\omega$$
$$\mathsf{gs}_N(\mathsf{start}_N,-) : \mathsf{At}^\omega \to (\mathsf{At}\cdot\Sigma)^* \cdot \mathsf{At} + (\mathsf{At}\cdot\Sigma)^\omega$$

agree on the set $\{\sigma \mid \mathsf{gs}_M(\mathsf{start}_M, \sigma) \text{ is finite}\}$. The accessibility condition in the statement of the lemma implies that this set is dense in At^ω under the usual metric topology on ω-sequences[1]. Moreover, the two functions are continuous, and continuous functions that agree on a dense set must agree everywhere. □

Lemma 3. *If M and N are bisimilar under $\equiv$, then for any $\sigma \in \mathsf{At}^*$, either both $\widehat{\delta}_M(\mathsf{start}_M, \sigma)$ and $\widehat{\delta}_N(\mathsf{start}_N, \sigma)$ are states, both are* halt*, or both are* fail*; and if they are states, then they are related by $\equiv$.*

2.6 Structured Programming Constructs

Deterministic while programs are formed inductively from sequential composition $(p\ ;\ q)$, conditional tests (if b then p else q), and while loops (while b do p), where b is a test and p, q are programs. We also include instructions skip (do nothing) and fail (looping or abnormal termination), although these constructs are redundant, being semantically equivalent to while false do p and while true do skip, respectively. We do not include a halt instruction; a program terminates normally by falling off the end.

Every while program can be converted to an equivalent strictly deterministic automaton. One first converts the program to a KAT term using the standard translation

$$p\ ;\ q = pq \qquad \mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q = bp + \overline{b}q \qquad \mathsf{while}\ b\ \mathsf{do}\ p = (bp)^*\overline{b},$$

then applies Kleene's theorem for KAT to yield an automaton with test and action states [5], which can be viewed as a deterministic flowchart F. One can then define a strictly deterministic transition function δ on the states of F as follows. For any state s and atom α, start at s and follow test transitions enabled by α until encountering an action state or a halt state. If an action state is encountered, let p be the label of the transition from that state and set $\delta(s, \alpha) = (p, t)$, where t is the target state of the transition. If a halt state is encountered, set $\delta(s, \alpha) = \mathsf{halt}$. If neither of these occur, that is, if the process traces a cycle of enabled test transitions, set $\delta(s, \alpha) = \mathsf{fail}$.

By restricting to the start state and the targets of action transitions, one obtains a strictly deterministic automaton of the form of Section 2.4. An example is shown in Fig. 1. In that figure, an edge from s to t labeled αp denotes the transition $\delta(s, \alpha) = (p, t)$. Note that the set of states of the strictly deterministic automaton is a subset of the states of the original automaton. The conversion of deterministic flowcharts to strictly deterministic automata does not change the set of guarded strings accepted. Moreover, by Kleene's theorem for KAT [5], this is the same as the set of guarded strings represented by the equivalent KAT expression.

In addition to the usual while program constructs, we consider the looping construct loop with nonlocal breaks break n, $n \geq 1$. After conversion to a deterministic flowchart, every loop instruction ℓ has one entry point entry_ℓ and

[1] The distance between two sequences is 2^{-n} if they agree on their first n symbols but differ on their $n + 1$st symbol.

```
while b do {
   while c do q;
   p;
}
```

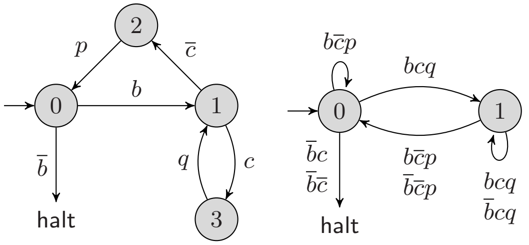


Fig. 1. A while program and its corresponding deterministic flowchart and strictly deterministic automaton

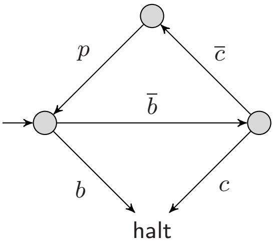


Fig. 2. An example from [4]

one exit point exit_ℓ. Intuitively, the instruction break n transfers control to exit_ℓ, where ℓ is the nth loop in whose scope the break n instruction occurs, counting from innermost to outermost. For while loops ℓ, $\mathsf{entry}_\ell = \mathsf{exit}_\ell$.

One can give a rigorous compositional semantics and an equational axiomatization of loop and break n, but this topic deserves a careful and systematic development that would be too much of a digression for the purposes of this paper, so we defer it to a forthcoming paper [6].

Allowing Boolean combinations of primitive tests in while loops and conditionals is quite natural and allows more flexibility than primitive tests alone. For instance, Kosaraju [4, Theorem 2] presents the flowchart of Fig. 2 as an example of a deterministic program that is not equivalent to any while program. This is true under his definition, but for the uninteresting reason that only primitive tests are allowed. Allowing Boolean combinations, the flowchart is equivalent to while $\overline{b\overline{c}}$ do p. The counterexample of Ashcroft and Manna [1] is much more complicated, requiring 13 nodes. Both proofs are rather lengthy and reason in terms of a particular first-order interpretation.

3 While Programs Are Not Sufficient

In this section we give a three-state strictly deterministic automaton M that cannot be represented by any while program. The states are $0, 1, 2$ with start

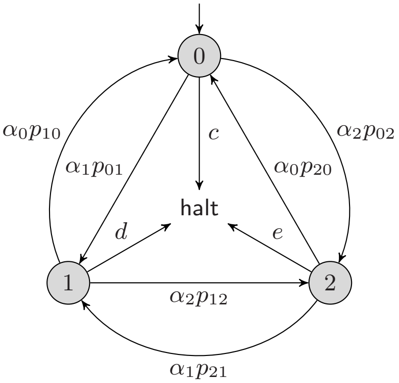


Fig. 3. A strictly deterministic automaton not equivalent to any while program

state 0. The primitive actions are p_{st} for $s, t \in \{0, 1, 2\}$, $s \neq t$, and the primitive tests are a, b, giving four atoms $\alpha_0, \ldots, \alpha_3$. The transitions are $\delta(s, \alpha_t) = (p_{st}, t)$ for $s, t \in \{0, 1, 2\}$, $s \neq t$, and $\delta(s, \alpha_3) = \delta(s, \alpha_s) = \mathsf{halt}$. The automaton M is illustrated in Fig. 3. For example, the edge from 0 to 2 labeled $\alpha_2 p_{02}$ represents the transition $\delta(0, \alpha_2) = (p_{02}, 2)$. The tests c, d, e represent $\alpha_0 + \alpha_3$, $\alpha_1 + \alpha_3$, and $\alpha_2 + \alpha_3$, respectively. The edge labeled c represents the two transitions $\delta(0, \alpha_0) = \delta(0, \alpha_3) = \mathsf{halt}$.

The automaton M has no nontrivial autobisimulation, since $\delta(s, \alpha_t) \neq \delta(t, \alpha_t)$ for $s \neq t$.

Theorem 1. *The strictly deterministic automaton M of Fig. 3 is not equivalent to any* while *program.*

Proof. Suppose for a contradiction that there exists a while program W equivalent to M; that is, such that $\mathsf{GS}(W) = \mathsf{GS}(M)$. Then W has a representation as a deterministic flowchart, and as a consequence of the construction of Section 2.6, as a strictly deterministic automaton S whose states are a subset of the states of W. We can assume without loss of generality that all states of W are accessible from start_W under a string in $\{\alpha_i \mid 0 \leq i \leq 2\}^*$; inaccessible states can be deleted with impunity. By Lemma 2, M and S are bisimilar.

For $s \in Q_S$, let $\mathsf{bisim}(s) \in Q_M$ be the unique state in M to which s is bisimilar. The state $\mathsf{bisim}(s)$ is unique, otherwise by transitivity there would be two bisimilar states of M, contradicting the fact that M is reduced. Also, since $\mathsf{start}_W \in Q_S$, $\mathsf{bisim}(\mathsf{start}_W)$ exists and is equal to 0.

Let $\ell = \mathsf{while}\ c\ \mathsf{do}\ r$ be a while loop in W of maximal depth, and let $s_0 = \mathsf{entry}_\ell = \mathsf{exit}_\ell$. Note that s_0 is not necessarily in Q_S. Let $s, t \in Q_S$ and $\alpha \in \mathsf{At}$ such that $\widehat{\delta}(s, \alpha) = (p_{ij}, t)$, s is not in the body of ℓ, and t is in the body of ℓ. It may be that $s = s_0$, but not necessarily. The states s and t exist, otherwise the body of ℓ is inaccessible. By symmetry, we may assume without loss of generality that $i = 0$ and $j = 1$. Thus $\mathsf{bisim}(s) = 0$, $\mathsf{bisim}(t) = 1$, $\alpha = \alpha_1$, and $\delta(s, \alpha_1) = \delta(s_0, \alpha_1) = (p_{01}, t)$.

Let σ be a maximum-length string of the form $(\alpha_2\alpha_1)^n$ or $(\alpha_2\alpha_1)^n\alpha_2$ such that the computation in W under σ starting from t does not meet s_0. The string σ exists, since ℓ has no inner loops, so all sufficiently long computations will loop back to s_0. Let $u = \widehat{\delta}(t,\sigma)$. The string σ cannot be of the form $(\alpha_2\alpha_1)^n\alpha_2$, because then we would have $\mathsf{bisim}(u) = 2$ and $\delta(u,\alpha_1) = \delta(s_0,\alpha_1) = (p_{21},w)$ for some w, a contradiction. Thus σ is of the form $(\alpha_2\alpha_1)^n$, and $\delta(s_0,\alpha_2) = (p_{12},w)$ for some w.

Suppose there is a state y in the body of ℓ with $\mathsf{bisim}(y) \in \{0,2\}$. Consider a maximum-length string of alternating α_2 and α_0 such that the computation sequence under this string starting at y does not meet s_0. The first atom of the sequence is α_2 if $\mathsf{bisim}(y) = 0$ and α_0 if $\mathsf{bisim}(y) = 2$. As above, the last state v of the sequence cannot be bisimilar to 0, because then we would have $\delta(v,\alpha_2) = \delta(s_0,\alpha_2) = (p_{02},z)$ for some z, a contradiction. Thus we must have $\delta(v,\alpha_0) = \delta(s_0,\alpha_0) = (p_{20},x)$ for some x.

Collecting information about ℓ so far, we have

$$\delta(s_0,\alpha_0) = (p_{20},x), \qquad \delta(s_0,\alpha_1) = (p_{01},t), \qquad \delta(s_0,\alpha_2) = (p_{12},w).$$

But now if we start from t and follow a sufficiently long path of the form $(\alpha_0\alpha_2\alpha_1)^*$, we will achieve a contradiction no matter what. Thus our assumption that the body of ℓ contains a state y with $\mathsf{bisim}(y) \in \{0,2\}$ was fallacious. The body of ℓ contains only the state $t \in Q_S$ with $\mathsf{bisim}(t) = 1$, and x and w are outside the body of ℓ. The loop ℓ is only entered under α_1, after which it performs the action p_{01} and immediately halts or exits the loop. Thus ℓ is equivalent to a conditional test.

By inductively replacing all maximally deeply nested while loops with equivalent conditional tests in this way, we can eventually eliminate all while loops. This is a contradiction. □

Theorem 1 shows that the Böhm-Jacopini theorem is false propositionally. In Section 5, a similar argument is used to prove the Kosaraju hierarchy theorem [4].

4 Loop Programs with Multi-level Breaks

As we saw in Section 3, while programs cannot express all programs represented by strictly deterministic automata. On the other hand, if an automaton has no cycles, then by duplicating states it can be converted to a tree, which is equivalent to a program built from just the if-then-else construct.

Motivated by this idea, our construction will first construct an equivalent tree-like automaton consisting of (downward-directed) tree transitions and (upward-directed) back transitions, then convert the resulting tree-like automaton to a loop program. This is done in three steps. The first step "unwinds" the original automaton to an infinite tree. This is a fairly standard construction, although we do it here with traces and bisimulations. The second step identifies states in the infinite tree with equivalent ancestors to obtain a finite tree-like automaton. In both steps, there is a bisimulation that guarantees equivalence. Finally, the tree-like automaton is converted to a loop program by using loop and $\mathsf{break}\ n$ to effect the back transitions and halting.

The "unwinding" of an automaton M to an infinite tree is done formally as follows. Let

$$U \stackrel{\text{def}}{=} (Q_U,\, \delta_U,\, \mathsf{start}_U)$$

where

$$\begin{aligned}
Q_U &\stackrel{\text{def}}{=} \{\text{finite traces } \sigma \text{ of } M \text{ such that } \mathsf{first}\,\sigma = \mathsf{start}_M\},\\
\delta_U(\sigma,\alpha) &\stackrel{\text{def}}{=} \begin{cases} (p,\, \sigma\alpha t) & \text{if } \delta_M(\mathsf{last}\,\sigma,\alpha) = (p,t)\\ \mathsf{halt} & \text{if } \delta_M(\mathsf{last}\,\sigma,\alpha) = \mathsf{halt}\\ \mathsf{fail} & \text{if } \delta_M(\mathsf{last}\,\sigma,\alpha) = \mathsf{fail},\end{cases}\\
\mathsf{start}_U &\stackrel{\text{def}}{=} \mathsf{start}_M.
\end{aligned}$$

Lemma 4. *The relation $\{(\sigma, \mathsf{last}\,\sigma) \mid \sigma \in Q_U\}$ is a bisimulation between U and M. Thus by Lemma 1, $\mathsf{GS}(U) = \mathsf{GS}(M)$.*

A *congruence* on M is an equivalence relation $\equiv$ that is an autobisimulation on M. Property (ii) of bisimulations says that the action of δ is well defined on $\equiv$-congruence classes, thus we can form the quotient automaton $M/\equiv$ whose states are the $\equiv$-congruence classes. Denote the congruence class of state u by $[u]$.

Lemma 5. *The relation $\{(u, [u]) \mid u \in Q_M\}$ is a bisimulation between M and $M/\equiv$. Thus by Lemma 1, $\mathsf{GS}(M) = \mathsf{GS}(M/\equiv)$.*

We can now use this construction to form a tree-like automaton with finitely many states equivalent to U. Here *tree-like* means that it has tree edges that form a rooted tree, but also may contain back edges to ancestors.

Recall that the states of U are the finite traces of M starting with start_M. For $\sigma, \tau \in Q_U$, set $\sigma\, R\, \tau$ iff

- all states of τ occur exactly once in τ except $\mathsf{last}\,\tau$, which occurs exactly twice;
- σ is the unique proper prefix of τ such that $\mathsf{last}\,\sigma = \mathsf{last}\,\tau$.

Let $\equiv$ be the smallest congruence containing R. That is, $\equiv$ is the smallest binary relation on Q_U such that

(i) $\equiv$ contains R,
(ii) $\equiv$ is an equivalence relation, and
(iii) if $\sigma \equiv \tau$ and $\delta_M(\mathsf{last}\,\sigma,\alpha) = \delta_M(\mathsf{last}\,\tau,\alpha) = (p,v)$, then $\sigma\alpha v \equiv \tau\alpha v$.

It can be shown inductively that $\mathsf{last}\,\sigma = \mathsf{last}\,\tau$ whenever $\sigma \equiv \tau$, so condition (iii) makes sense. The quotient automaton $U/\equiv$ has finitely many states, at most $(|Q_M| - 1)!$ in fact, since each $\equiv$-congruence class contains a unique trace

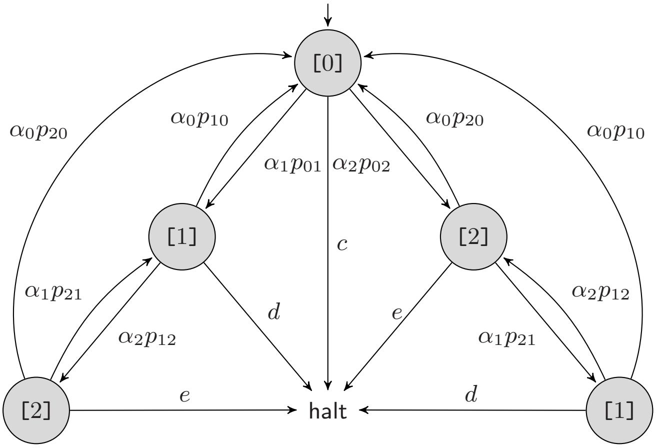


Fig. 4. A tree-like automaton equivalent to the automaton of Fig. 3

with no repeated states beginning with start_M. This trace is of minimal length among all elements of its $\equiv$-class. It can be obtained from any other element of the class by repeatedly deleting the subtrace between the first recurring state and its earlier occurrence.

We can view the states of $U/\equiv$ as arranged in a tree with root $[\mathsf{start}_M]$, tree edges to descendants, and back edges to ancestors. By Lemma 5, $\mathsf{GS}(U) = \mathsf{GS}(U/\equiv)$.

Now we can convert this automaton to a loop program as follows. Let C be the set of traces of M with no repeated states beginning with start_M. This is the set of canonical representatives of the $\equiv$-classes. Let $\alpha_1, \ldots, \alpha_m$ be the elements of At. For each $\sigma \in C$, let L_σ be the following loop program:

```
loop {
   if $\alpha_1$ then $S_1$
   else if $\alpha_2$ then $S_2$
   ...
   else if $\alpha_m$ then $S_m$
}
```

where

(i) if $\delta_M(\mathsf{last}\,\sigma, \alpha_i) = (p, t)$, then

$$S_i = \begin{cases} p \,;\, L_{\sigma\alpha_i t} & \text{if } t \text{ does not occur in } \sigma, \\ p & \text{if } t = \mathsf{last}\,\sigma, \\ p \,;\, \mathsf{break}\ n & \text{if } t \text{ occurs in } \sigma \text{ but } t \neq \mathsf{last}\,\sigma, \end{cases}$$

where in the last case, n is the number of states occurring after t in σ;

(ii) if $\delta_M(\mathsf{last}\,\sigma, \alpha_i) = \mathsf{halt}$, then $S_i = \mathsf{break}\ n$, where n is the number of states in σ; and

(iii) if $\delta_M(\mathsf{last}\,\sigma, \alpha_i) = \mathsf{fail}$, then $S_i = \mathsf{loop\ skip}$.

The choice of n in the last case of (i) causes control to return to the top of L_τ, where τ is the unique prefix of σ such that $\mathsf{last}\,\tau = t$. This is tantamount to taking the back edge from the node of the tree represented by σ to its ancestor represented by τ. The choice of n in case (ii) causes the program to halt by exiting the outermost loop L_{start}.

Example 1. Fig. 4 shows a tree-like automaton equivalent to the automaton of Fig. 3. (The central edges leading to halt are not considered part of the tree, since halt is not a state of the automaton.) A corresponding loop program is shown in Fig. 5. This is not exactly the program that would be produced by the construction given above; we have removed the innermost loops to save space.

```
loop {
    if $\overline{a}$ then break 1;
    if $b$ then {
        $p$;
        loop {
            if $\overline{a}$ then break 2;
            if $\overline{b}$ then { $t$; break 1; }
            else {
                $s$;
                if $\overline{a}$ then break 2;
                if $b$ then { $v$; break 1; }
                else $w$;
            }
        }
    } else {
        $q$;
        loop {
            if $\overline{a}$ then break 2;
            if $b$ then { $v$; break 1; }
            else {
                $w$;
                if $\overline{a}$ then break 2;
                if $\overline{b}$ then { $t$; break 1; }
                else $s$;
            }
        }
    }
}
```

Fig. 5. A loop program equivalent to the automaton of Fig. 4; we have removed the innermost loops to save space

5 The Loop Hierarchy

Now we give an alternative proof of the hierarchy result of Kosaraju [4], namely that there is a strict hierarchy of loop programs determined by the depth of nesting of loop instructions.

We construct an automaton P_n as follows. The states of P_n are all strings over the alphabet $\{0, 1, \ldots, n-1\}$ with no repeated letters, including the empty string ε. There are roughly $n!e$ states. The atoms and actions are $0, 1, \ldots, n-1$ and E. The transition i appends i to the current string if it does not already occur, or else truncates back to the prefix ending in i if it does occur. The transition E erases the string. We also include an atom H such that $\delta(s, H) = \mathsf{halt}$ for states s of maximum length n and $\delta(s, H) = \mathsf{fail}$ for the other states. This precludes nontrivial autobisimulations.

An illustration of a depth-4 implementation of P_7 is shown in Fig. 6. Each box represents a loop instruction. Only four paths of the nested loop program are shown; there are many others not shown. There is one top-level loop, $\binom{n}{2}2!$ second-level loops, $\binom{n}{4}4!$ third-level loops within each second-level loop, etc.

At the entry point of each loop, there is a multiway branch depending on the current atom. The resulting action is the same as the atom, and the new state is the one whose last symbol is the action just performed (except for ε, which is only obtained by E). Note that every prefix of every string occurs in the same loop or an ancestor, therefore is accessible by a break instruction, and every string obtained by appending one symbol is in the same loop or a child loop.

For example, suppose the current state is 012. If we perform action 3, we would enter the subloop below 012 containing 0123. If we perform action 4, we would enter a parallel subloop not shown. If we perform action 1, we would loop to the top of the current loop, execute the action 1, and enter state 01.

Theorem 2. *The program P_n can be implemented in depth $\lfloor n/2+1 \rfloor$ and no less.*

Proof. For the upper bound, Fig. 6 illustrates the pattern that achieves $\lfloor n/2+1 \rfloor$.

For the lower bound, the proof is by induction on n. The idea is illustrated in Fig. 6: note that all strings in the 01 subloop begin with 01, so it has the same structure as the outer loop, but with two fewer letters. We actually prove a stronger result, namely that the bound holds irrespective of which state is the start state.

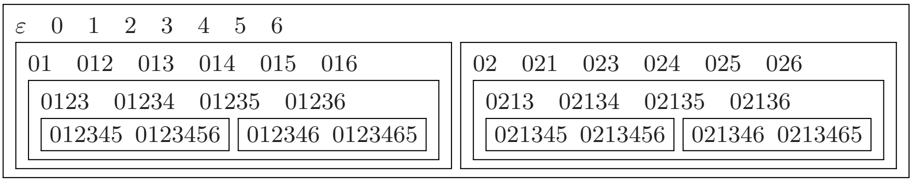


Fig. 6. Automaton P_7 implemented with a loop program of depth 4

The basis for $n = 0$ and $n = 1$ is trivial, since there always must be at least one loop. For $n = 2$, there are only three transitions but five states requiring self-loops, so the depth must be at least two.

Now let $n \geq 3$ and let W be any implementation of P_n. Let ℓ_0 be an outer loop of W and $s_0 = \mathsf{entry}_{\ell_0}$. There must be some pair i, j such that ij does not appear as a prefix of any x for $\delta(s_0, k) = (k, x)$. Say $ij = 01$ without loss of generality.

Consider the subprogram ℓ_1 consisting of all states of W that are accessible from 01 after deleting the transitions E and 0 (that is, setting them to fail). Since all states of ℓ_1 have prefix 01, the entry point s_0 of ℓ_0 is no longer accessible, so the outer loop can be deleted. The represented automaton is isomorphic to P_{n-2}. The transition 1 plays the role of E. By the induction hypothesis, it must have depth at least $\lfloor (n-2)/2 + 1 \rfloor$, thus ℓ_0 must have depth at least $\lfloor (n-2)/2 + 1 \rfloor + 1 = \lfloor n/2 + 1 \rfloor$.

6 Conclusion and Open Problems

We have shown three results giving upper and lower bounds on the power of various programming constructs to represent flowchart programs, modeled as automata on guarded strings. On the one hand, the simple three-state automaton in Section 3 cannot be represented by any while program. On the other hand, we present a congruence in Section 4 that transforms any automaton into a tree-like structure and show how a tree-like automaton can be turned into a loop program with multi-level breaks. We also give an alternative proof of Kosaraju's hierarchy result for loop programs with multi-level breaks.

We did not give a formal proof of equivalence between the tree-like automaton and its corresponding loop program with multi-level breaks constructed in Section 4. However, it is possible to prove their equivalence formally. The break n construct, and more generally the goto construct, although representing nonlocal flow of control, can nevertheless be given a formal equational semantics in the style of KAT. We have developed this semantics and an equational axiomatization and have shown how to use it to give rigorous proofs of the correctness of transformations like those of Section 4 [6].

One popular line of research has been to develop restructuring techniques that minimize the amount of duplication of code [8,10]. The construction given in Section 4 is as bad in this regard as it can possibly be: it transforms an n-state automaton to an $(n-1)!$-state tree-like automaton in the worst case. Are there more efficient transformations at the propositional level? Or is this an inescapable feature of the propositional formulation?

Acknowledgements

We would like to thank the reviewers for their helpful suggestions for improving the presentation. This work was supported by NSF grant CCF-0635028 and a NSF Graduate Research Fellowship.

References

1. Ashcroft, E., Manna, Z.: The translation of goto programs into while programs. In: Freiman, C.V., Griffith, J.E., Rosenfeld, J.L. (eds.) Proceedings of IFIP Congress 71, vol. 1, pp. 250–255. North-Holland, Amsterdam (1972)
2. Böhm, C., Jacopini, G.: Flow diagrams, Turing machines and languages with only two formation rules. In: Communications of the ACM, pp. 366–371 (May 1966)
3. Kaplan, D.M.: Regular expressions and the equivalence of programs. J. Comput. Syst. Sci. 3, 361–386 (1969)
4. Kosaraju, S.R.: Analysis of structured programs. In: Proc. 5th ACM Symp. Theory of Computing (STOC 1973), pp. 240–252. ACM, New York (1973)
5. Kozen, D.: Automata on guarded strings and applications. Matématica Contemporânea 24, 117–139 (2003)
6. Kozen, D.: Nonlocal flow of control and Kleene algebra with tests. Technical Report http://hdl.handle.net/1813/10595 Computing and Information Science, Cornell University (April 2008); Proc. 23rd IEEE Symp. Logic in Computer Science (LICS 2008) (to appear, June 2008)
7. Morris, P.H., Gray, R.A., Filman, R.E.: Goto removal based on regular expressions. J. Software Maintenance: Research and Practice 9(1), 47–66 (1997)
8. Oulsnam, G.: Unraveling unstructured programs. The Computer Journal 25(3), 379–387 (1982)
9. Peterson, W., Kasami, T., Tokura, N.: On the capabilities of while, repeat, and exit statements. Comm. Assoc. Comput. Mach. 16(8), 503–512 (1973)
10. Ramshaw, L.: Eliminating goto's while preserving program structure. Journal of the ACM 35(4), 893–920 (1988)
11. Williams, M., Ossher, H.: Conversion of unstructured flow diagrams into structured form. The Computer Journal 21(2), 161–167 (1978)

The Expression Lemma*

Ralf Lämmel[1] and Ondrej Rypacek[2]

[1] The University of Koblenz-Landau, Germany
[2] The University of Nottingham, UK

Abstract. Algebraic data types and catamorphisms (folds) play a central role in functional programming as they allow programmers to define recursive data structures and operations on them uniformly by structural recursion. Likewise, in object-oriented (OO) programming, recursive hierarchies of object types with virtual methods play a central role for the same reason. There is a semantical correspondence between these two situations which we reveal and formalize categorically. To this end, we assume a coalgebraic model of OO programming with functional objects. The development may be helpful in deriving refactorings that turn sufficiently disciplined functional programs into OO programs of a designated shape and vice versa.

Keywords: expression lemma, expression problem, functional object, catamorphism, fold, the composite design pattern, program calculation, distributive law, free monad, cofree comonad.

1 Introduction

There is a folk theorem that goes as follows. Given is a recursively defined data structure with variants $d_1, \ldots, d_q$, and operations $o_1, \ldots, o_r$ that are defined by structural recursion on the data variants. There are two equivalent implementations. In the *functional* style, we define recursive functions $f_1, \ldots, f_r$ such that each f_i implements o_i and is defined by q equations, one equation for each d_j. In the *object-oriented* (OO) style, we define object types $t_1, \ldots, t_q$ such that each t_j uses d_j as its opaque state type, and it implements a common interface consisting of methods $m_1, \ldots, m_r$ with the types of $f_1, \ldots, f_r$, except that the position of structural recursion is mapped to "self" (say, "this"). The per-variant equations of the functional style correspond to the per-variant method implementations of the OO style. Refer to Fig. 1 for a Haskell and a Java program that are related in the described manner.

This folk theorem is related to the so-called *expression problem* [28], which focuses on the extensibility trade-offs of the different programming styles, and aims at improved language designs with the best possible extensibility for all possible scenarios. Such a comparison of styles can definitely benefit from an understanding of the semantical correspondence between the styles, which is indeed the overall contribution of the present paper. We coin the term *expression lemma* to refer to the sketched functional/OO correspondence. We do not discuss the expression problem any further in this paper, but

* See `http://www.uni-koblenz.de/~laemmel/expression/` (the paper's web site) for an extended version.

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 193–219, 2008.

```
-- Arithmetic expression forms
data Expr = Num Int | Add Expr Expr

-- Evaluate expressions
eval :: Expr → Int
eval (Num i) = i
eval (Add l r) = eval l + eval r

-- Modify literals modulo v
modn :: Expr → Int → Expr
modn (Num i) v = Num (i ‘mod‘ v)
modn (Add l r) v = Add (modn l v) (modn r v)
```

```
public abstract class Expr {
    public abstract int eval ();
    public abstract void modn(int v);
}

public class Num extends Expr {
    private int value;
    public Num(int value) { this.value = value; }
    public int eval() { return value; }
    public void modn(int v) { this.value = this.value % v; }
}

public class Add extends Expr {
    private Expr left , right ;
    public Add(Expr left, Expr right) { this. left = left; this. right = right; }
    public int eval() { return left .eval() + right .eval (); }
    public void modn(int v) { left .modn(v); right.modn(v); }
}
```

Fig. 1. A Haskell program and a Java program; the two programs define the same kind of structurally recursive operations on the same kind of recursive data structure

we contend that the expression lemma backs up past and future work on the expression problem. It is also assumed that the lemma contributes to the foundation that is needed for future work on refactorings between the aforementioned styles, e.g., in the context of making imperative OO programs more pure, more functional, or more parallelizable.

Contributions

The paper provides a technical formulation of the expression lemma, in fact, the first such formulation, as far as we know. The formulation is based on comparing functional programs (in particular, folds) with coalgebraically modeled OO programs. We provide first ever, be it partial answers to the following questions:

- When is a functional program that is defined by recursive functions on an algebraic data type semantically equivalent to an OO program that is defined by recursive methods on object structures?
- What is the underlying common definition of the two programs?

The formal development is done categorically, and essentially relies on the established concept of distributive laws of a functor over a functor and (co)monadic generalizations thereof. A class of pairs of functional and OO programs that are duals of each other is thus revealed and formalized. This makes an important contribution to the formal understanding of the semantical equivalence of functional and OO programming. Nontrivial facets of dualizable programs are identified — including facets for the freewheeling use of term construction (say, object construction) and recursive function application (say, method invocation).[1]

Road-Map

The paper is organized as follows. § 2 sketches (a simple version of) the expression lemma and its proof. § 3 approaches the expression lemma categorically. § 4 interprets the basic formal development, and suggests extensions to cover a larger class of dualizable programs. § 5 formalizes the proposed extensions. § 6 discusses related work. § 7 concludes the paper.

2 Informal Development

In order to clarify the correspondence between the functional and the OO style, we need a setup that admits the comparison of both kinds of programs. In particular, we must introduce a suitable formalism for objects. We adopt the coalgebraic view of functional objects, where object interfaces are modeled as *interface endofunctors*, and implementations are modeled as *coalgebras* of these functors. We will explain the essential concepts here, but refer to [14, 22] for a proper introduction to the subject, and to [21] for a type-theoretical view. In the present section, we illustrate the coalgebraic model in Haskell, while the next section adopts a more rigorous, categorical approach.

2.1 Interfaces and Coalgebras

The following type constructor models the *interface* of the base class Expr of Fig. 1:

```
type IExprF x = (Int, Int → x)
```

Here, x is the type parameter for the object type that implements the interface; the first projection corresponds to the observer eval (and hence its type is Int); the second projection corresponds to the modifier modn (and hence its type is Int→x, i.e., the method takes an Int and returns a new object of the same type x). We also need a type that hides the precise object type — in particular, its state representation; this type would effectively be used as a bound for all objects that implement the interface. To this end, we may use the recursive closure of IExprF:

```
newtype IExpr = InIExpr { outIExpr :: IExprF IExpr }
```

Method calls boil down to the following convenience projections:

```
call_eval  = fst . outIExpr
call_modn = snd . outIExpr
```

[1] The paper comes with a source distribution that contains a superset of all illustrations in the paper as well as a Haskell library that can be used to code dualizable programs. Refer to the paper's web site.

```
−− eval function for literals
numEval :: Int → Int
numEval = id
```

```
−− modn function for literals
numModn :: Int → Int
numModn = mod
```

```
−− eval function for additions
addEval :: (Int, Int) → Int
addEval = uncurry (+)
```

```
−− modn function for addition
addModn :: (Int→a,Int→a)→Int→(a,a)
addModn = uncurry (/\)
```

Fig. 2. Component functions of the motivating example

Implementations of the interface are *coalgebras* of the IExprF functor. Thus, the type of IExprF implementations is the following:

```
type IExprCoalg x = x → IExprF x
```

Here are the straightforward implementations for the Expr hierarchy:

```
numCoalg :: IExprCoalg Int
numCoalg = numEval /\ numModn

addCoalg :: IExprCoalg (IExpr, IExpr)
addCoalg = (addEval . (call_eval <∗> call_eval)) /\
           (addModn . (call_modn <∗> call_modn))
```

For clarity (and reuse), we have factored out the actual functionality per function and data variant into helper functions numEval, addEval, numModn, and addModn; c.f. Fig. 2.[2] Note that the remaining polymorphism of addModn is essential to maintain enough naturality needed for our construction.

The above types clarify that the implementation for literals uses int as the *state type*, whereas the implementation for additions uses (IExpr, IExpr) as the state type. Just as the original Java code invoked methods on the components stored in left and right, the coalgebraic code applies the corresponding projections of the IExpr-typed values call_eval and call_modn. It is important to keep in mind that IExpr-typed values are effectively functional objects, i.e., they return a "modified (copy of) self", when mutations are to be modeled.

2.2 Object Construction by Unfolding

Given a coalgebra $x \longrightarrow f\ x$ for some fixed f and x, we can *unfold* a value of type x to the type of the fixed point of f. The corresponding well-known recursion scheme of *anamorphisms* [19] is instantiated for IExpr as follows:

[2] We use point-free (pointless) notation (also known as Backus' functional forms) throughout the paper to ease the categorical development. In particular, we use these folklore operations: f <*> g maps the two argument functions over the components of a pair; f /\ g constructs a pair by applying the two argument functions to the same input; f <|> g maps over a sum with an argument function for each case; f \/ g performs case discrimination on a sum with an argument function for each case. Refer to Fig. 3 for a summary of the operations.

```
(<∗>) :: (a → b) → (c → d) → (a,c) → (b,d)
(f <∗> g) (x, y) = (f x, g y)

(/\) :: (a → b) → (a → c) → a → (b,c)
(f /\ g) x = (f x, g x)

(<|>) :: (a → b) → (c → d) → Either a c → Either b d
(f <|> g) (Left x)  = Left (f x)
(f <|> g) (Right y) = Right (g y)

(\/) :: (a → c) → (b → c) → Either a b → c
(f \/ g) (Left x)  = f x
(f \/ g) (Right y) = g y
```

Fig. 3. Folklore operations on sums and products

```
unfoldIExpr :: IExprCoalg x → x → IExpr
unfoldIExpr c = InIExpr . fmapIExprF (unfoldIExpr c) . c
  where
    fmapIExprF :: (x → y) → IExprF x → IExprF y
    fmapIExprF f = id <∗> (.) f
```

That is, IExprF x is injected into IExpr by recursively unfolding all occurrences of x by means of the functorial map operation, fmapIExprF. The type parameter x occurs in the positions where “a modified (copy of) self” is returned. In OO terms, applications of unfoldIExpr are to be viewed as *constructor* methods:

```
newNum :: Int → IExpr
newNum = unfoldIExpr numCoalg

newAdd :: (IExpr, IExpr) → IExpr
newAdd = unfoldIExpr addCoalg
```

This completes the transcription of the Java code of Fig. 1 to the coalgebraic setup.

2.3 Converting State Trees to Object Trees

In establishing a semantical correspondence between functional and OO programs, we can exploit the following property: *the functional style is based on recursion into terms whose structure coincides with the states of the objects.* Hence, let us try to convert such trees of states of objects (state trees) into trees of objects (object trees). We will then need to compare the semantics of the resulting objects with the semantics of the functional program.

In the introduction, we defined the data type Expr as an algebraic data type; for the sake of a more basic notation, we define it here as the fixed point of a “sum-of-products” functor ExprF equipped with convenience injections num and add — reminiscent of the algebraic data-type constructors of the richer notation. Thus:

```
type ExprF x = Either Int (x,x)
newtype Expr = InExpr { outExpr :: ExprF Expr }
```

```
num = InExpr . Left
add = InExpr . Right
```

The objects we construct have internal state either of type Int or of type (IExpr,IExpr). The corresponding sum, Either Int (IExpr,IExpr), coincides with ExprF IExpr, i.e., the mere state trees resemble the term structure in the functional program. We have defined coalgebras numCoalg and addCoalg for each type of state. We can also define a coalgebra for the union type ExprF IExpr:

```
eitherCoalg :: IExprCoalg (ExprF IExpr)
eitherCoalg = ((id <*> (.) Left) \/ (id <*> (.) Right)) .
              (numCoalg <|> addCoalg)
```

The coalgebra eitherCoalg may be viewed as an implementation of an object type that physically uses a sum of the earlier state types. Alternatively, we may view the coalgebra as an object factory (in the sense of the *abstract factory* design pattern [10]). That is, we may use it as a means to construct objects of either type.

```
newEither :: ExprF IExpr → IExpr
newEither = unfoldIExpr eitherCoalg
```

It is important to understand the meaning of ExprF IExpr: the type describes states of objects, where the state representation is only exposed at the top-level, but all deeper objects are already opaque and properly annotated with behavior. While newEither facilitates one level of object construction, we ultimately seek the recursive closure of this concept. That is, we seek to convert a pure state tree to a proper object tree. It turns out that the fold operation for Expr immediately serves this purpose.

In general, the fold operation for a given sums-of-products functor is parametrized by an *algebra* that associates each addend of the functor's sum with a function that combines recursively processed components and other components. For instance, the algebra type for expressions is the following:

```
type ExprAlg x = ExprF x →x
```

Given an algebra $f\ x \longrightarrow x$ for some fixed f and x, we can *fold* a value of the type of the fixed point of f to a value of type x. The corresponding well-known recursion scheme of *catamorphisms* is instantiated for Expr as follows:

```
foldExpr :: ExprAlg x → Expr → x
foldExpr a = a . fmapExprF (foldExpr a) . outExpr
  where
    fmapExprF :: (x → y) → ExprF x → ExprF y
    fmapExprF f = id <|> (f <*> f)
```

We should simplify the earlier type for newEither as follows:

```
newEither :: ExprAlg IExpr
```

Hence, we can fold over state trees to obtain object trees.

```
−− Fold the unfold
fu :: Expr → IExpr
fu = foldExpr newEither
```

2.4 Implementing Interfaces by Folds over State Trees

It remains to compare the semantics of the constructed objects with the semantics of the corresponding functional program. To this end, we also model the construction of objects whose object type corresponds to an abstract data type (ADT) that exports the functions of the functional program as its operations.

The initial definitions of the functions eval and modn used general recursion. Our development relies on the fact that the functions are catamorphisms (subject to further restrictions). Here are the new definitions; subject to certain preconditions, programs that use general recursion can be automatically converted to programs that use the catamorphic scheme [11, 17]:

```
evalAlg :: ExprAlg Int
evalAlg = numEval \/ addEval

eval :: Expr → Int
eval = foldExpr evalAlg

modnAlg :: ExprAlg (Int → Expr)
modnAlg = ((.) num . numModn) \/ ((.) add . addModn)

modn :: Expr → Int → Expr
modn = foldExpr modnAlg
```

Just like objects combine behavior for all operations in their interfaces, we may want to tuple the folds, such that all recursions are performed simultaneously. That is, the result type of the paired fold is the product of the result types of the separated folds. Again such pairing (tupling) is a well-understood technique [5, 9, 12, 19] that can also be used in an automated transformation. Thus:

```
bothAlg :: ExprAlg (IExprF Expr)
bothAlg = (evalAlg <*> modnAlg) . ((id <|> (fst <*> fst)) /\
                (id <|> (snd <*> snd)))

both :: IExprCoalg Expr
both = foldExpr bothAlg
```

That is, both does both, eval and modn. Now we can construct objects whose behavior is immediately defined in terms of both (hence, essentially, in terms of the original functions eval and modn). To this end, it is sufficient to realize that both readily fits as a coalgebra, as evident from its type:

```
Expr → (Int, Int → Expr) ≡ Expr → IExprF Expr ≡ IExprCoalg Expr
```

Thus, we can construct objects (ADTs, in fact) as follows:

```
-- Unfold the fold
uf :: Expr → IExpr
uf = unfoldIExpr both
```

That is, we have encapsulated the functional folds with an argument term such that the resulting interface admits the applications of the functional folds to the term.

2.5 The Expression Lemma

Let us assume that we were able to prove the following identity:

```
foldExpr (unfoldIExpr eitherCoalg) = unfoldIExpr (foldExpr bothAlg)
```

We refer to the generic form of this identity as the expression lemma; c.f. § 3.4. Roughly, the claim means that *objects that were constructed from plain state trees, level by level, behave the same as shallow objects that act as abstract data types with the functional folds as operations*. Hence, this would define a proper correspondence between anamorphically (and coalgebraically) phrased OO programs and catamorphically phrased functional programs. § 3 formalizes this intuition and proves its validity.

The formal development will exploit a number of basic categorical tools, but a key insight is that the defining coalgebra of the OO program (i.e., eitherCoalg) and the defining algebra of the functional program (i.e., bothAlg) essentially involve the same function (in fact, a *natural transformation*). To see this, consider the expanded types of the (co)algebras in question:

```
eitherCoalg :: ExprF IExpr→ IExprF (ExprF IExpr)
bothAlg :: ExprF (IExprF Expr)→ IExprF Expr
```

The types differ in the sense that eitherCoalg uses the recursive closure IExpr in one position where bothAlg uses an application of the functor IExprF instead, and bothAlg uses the recursive closure Expr in another position where eitherCoalg uses an application of the functor ExprF instead. Assuming a natural transformation lambda, both functions can be defined as follows:

```
eitherCoalg = lambda . fmapExprF outIExpr
bothAlg = fmapIExprF InExpr . lambda
lambda :: ExprF (IExprF x) → IExprF (ExprF x)
lambda = ???
```

Note that naturality of lambda is essential here. It turns out that we can define lambda in a "disjunctive normal form" over the same ingredients that we also used in the original definitions of eitherCoalg and bothAlg:

```
lambda = (numEval /\ ((.) Left . numModn)) \/ ((addEval . (fst <∗> fst)) /\
            ((.) Right . addModn . (snd <∗> snd)))
```

It is straightforward to see that eitherCoalg and bothAlg as redefined above equate to the original definitions of eitherCoalg and bothAlg based on just trivial laws for sums and products. We have thus factored out a common core, a distributive law, lambda, from which both programs can be canonically defined.

3 The Basic Categorical Model

The intuitions of the previous section will now be formalized categorically. The used categorical tools are established in the field of functional programming theory. The contribution of the section lies in leveraging these known tools for the expression lemma.

3.1 Interface Functors

Definition 1. *An* interface functor *(or simply* interface*) is a polynomial endofunctor on a category*[3] $\mathbf{C}$ *of the form*

$$O \times M \;\; with \;\; O = \prod_{i \in I} A_i^{B_i} \;\; and \;\; M = \prod_{j \in J} (C_j \times \mathsf{Id})^{D_j}$$

where $\prod$ *denotes iterated product, all* A*s ,* B*s ,* C*s and* D*s are constant functors,* Id *is the identity functor and all products and exponents are lifted to functors.* I *and* J *are finite sets.*

Note that here and in the rest of the text we use the exponential notation for the function space as usual. Informally, O collects all "methods" that do not use the type of "self" in their results, as it is the case for observers; M collects all "methods" that return a "mutated (copy of) self" (c.f. the use of "Id"), and possibly additional data (c.f. the Cs).

Example 1. IExprF is an interface functor.

$$\mathsf{IExprF} \;=\; \mathsf{Int} \times \mathsf{Id}^{\mathsf{Int}} \;\cong\; \mathsf{Int}^1 \times (1 \times \mathsf{Id})^{\mathsf{Int}}$$

3.2 *F*-(co)Algebras and Their Morphisms

Definition 2. *Let* F *be an endofunctor on a category* $\mathbf{C}$*,* A*,* B *objects in* $\mathbf{C}$*.*

- *An* F***-algebra*** *is an arrow* $F\,A \longrightarrow A$*. Here* A *is called the* ***carrier*** *of the algebra.*
- *For* F*-algebras* $\varphi \;:\; F\,A \longrightarrow A$ *and* $\psi \;:\; F\,B \longrightarrow B$*, an* F***-algebra morphism*** *from* φ *to* ψ *is an arrow* $f \;:\; A \longrightarrow B$ *of* $\mathbf{C}$ *such that the following holds:*

$$f \circ \varphi \;=\; \psi \circ F f \tag{1}$$

- F*-algebras and* F*-algebra morphisms form a category denoted* $\mathbf{C}^F$*. The initial object in this category, if it exists, is the* ***initial*** F***-algebra****. Explicitly, it is an* F*-algebra,* $\mathsf{in}_F \;:\; F\,\mu F \longrightarrow \mu F$*, such that for any other* F*-algebra,* $\varphi \;:\; F\,A \longrightarrow A$*, there exists a* unique F*-algebra morphism* $(\!|\, \varphi \,|\!)_F$ *from* in_F *to* φ*. Equivalently:*

$$h \;=\; (\!|\, \varphi \,|\!)_F \;\Leftrightarrow\; h \circ \mathsf{in}_F \;=\; \varphi \circ F h \tag{2}$$

- *The duals of the above notions are* F***-coalgebra****,* F***-coalgebra morphism*** *and the* ***terminal*** F***-coalgebra****. Explicitly, an* F*-coalgebra is an arrow* $\varphi : A \longrightarrow F\,A$ *in* $\mathbf{C}$*. The category of* F*-coalgebras is denoted* $\mathbf{C}_F$*. The terminal* F*-coalgebra is denoted* $\mathsf{out}_F \;:\; \nu F \longrightarrow F \nu F$*, and the unique terminal* F*-coalgebra morphism from* φ *is denoted* $[\!(\, \varphi \,)\!]_F : A \longrightarrow \nu F$*. These satisfy the following duals of (1) and (2).*

$$\psi \circ g = F g \circ \varphi \tag{3}$$

$$h \;=\; [\!(\, \varphi \,)\!]_F \Leftrightarrow \mathsf{out}_F \circ h \;=\; F h \circ \varphi \tag{4}$$

[3] For simplicity, in this paper, we assume a category $\mathbf{C}$ with enough structure to support our constructions. The category $\mathbf{SET}$ is always a safe choice.

Example 2. (Again, we relate to the Haskell declarations of § 2.) ExprF is an endofunctor; $\mu\mathsf{ExprF}$ corresponds to type Expr; $\mathsf{evalAlg}$ and $\mathsf{modnAlg}$ are ExprF-algebras. The combinator $(\!|\ |\!)_{\mathsf{ExprF}}$ corresponds to $\mathsf{foldExpr}$. Likewise, IExprF is an endofunctor; $\nu\mathsf{IExprF}$ corresponds to IExpr; $\mathsf{numCoalg}$ and $\mathsf{addCoalg}$ are IExprF-coalgebras. The combinator $[\!(\ )\!]_{\mathsf{IExprF}}$ corresponds to $\mathsf{unfoldIExpr}$.

3.3 Simple Distributive Laws

Our approach to proving the correspondence between OO and functional programs critically relies on distributive laws as a means to relate algebras and coalgebras. In the present section, we only introduce the simplest form of distributive laws.

Definition 3. *A* ***distributive law*** *of a* ***functor*** F *over a* ***functor*** B *is a natural transformation* $FB \longrightarrow BF$*.*

Example 3. Trivial examples of distributive laws are algebras and coalgebras. That is, *any* coalgebra $X \longrightarrow BX$ is a distributive law where F in Def. 3 is fixed to be a constant functor. Dually, *any* algebra $FX \longrightarrow X$ is a distributive law where B in Def. 3 is fixed to be a constant functor. This fact is convenient in composing algebras and coalgebras (and distributive laws), as we will see shortly.

Example 4. Here are examples of nontrivial distributive laws:[4]

$$\begin{aligned} \mathsf{addModn} &: \mathsf{Id}^2\, \mathsf{Id}^{\mathsf{Int}} \longrightarrow \mathsf{Id}^{\mathsf{Int}}\, \mathsf{Id}^2 \\ \mathsf{lambda} &: \mathsf{ExprF}\, \mathsf{IExprF} \longrightarrow \mathsf{IExprF}\, \mathsf{ExprF} \end{aligned}$$

Distributive laws can be combined in various ways, e.g., by $\oplus$ and $\otimes$ defined as follows:

Definition 4. *Let* $\lambda_i : F_i B \longrightarrow BF_i,\ i \in \{1, 2\}$ *be distributive laws.*
Then we define a distributive law:

$$\begin{aligned} &\lambda_1 \oplus \lambda_2 : (F_1 + F_2)B \longrightarrow B(F_1 + F_2) \\ &\lambda_1 \oplus \lambda_2 \equiv (B\iota_1 \triangledown B\iota_2) \circ (\lambda_1 + \lambda_2) \end{aligned}$$

Here, $f \triangledown g$ *is the* cotuple *of* f *and* g *with injections* ι_1 *and* ι_2*, that is the unique arrow such that* $(f \triangledown g) \circ \iota_1 = f$ *and* $(f \triangledown g) \circ \iota_2 = g$*.*

Definition 5. *Let* $\lambda_i : FB_i \longrightarrow B_iF,\ i \in \{1, 2\}$ *be distributive laws.*
Then we define a distributive law:

$$\begin{aligned} &\lambda_1 \otimes \lambda_2 : F(B_1 \times B_2) \longrightarrow (B_1 \times B_2)F \\ &\lambda_1 \otimes \lambda_2 \equiv (\lambda_1 \times \lambda_2) \circ (F\pi_1 \vartriangle F\pi_2) \end{aligned}$$

Here, $f \vartriangle g$ *is the* tuple *of* f *and* g *with projections* π_1 *and* π_2*, that is the unique arrow such that* $\pi_1 \circ (f \vartriangle g) = f$ *and* $\pi_2 \circ (f \vartriangle g) = g$*.*

We assume the usual convention that $\otimes$ binds stronger than $\oplus$.

[4] Note that we use just juxtaposition for functor composition. Confusion with application is not an issue because application can be always considered as composition with a constant functor. Also note that $F^2 \cong F \times F$ in a bicartesian closed category.

Example 5. As algebras and coalgebras are distributive laws, $\oplus$ and $\otimes$ readily specialize to combinators on algebras and coalgebras, as in $\mathsf{bothAlg}$ or $\mathsf{eitherCoalg}$. A nontrivial example of a combination of distributive laws is lambda:

$$\mathsf{lambda} \;=\; \mathsf{numEval} \otimes \mathsf{numModn} \;\oplus\; \mathsf{addEval} \otimes \mathsf{addModn} \tag{5}$$

The following lemma states a basic algebraic property of $\oplus$ and $\otimes$.

Lemma 1. *Let* $\lambda_{i,j} \;:\; F_{i,j}B_{i,j} \longrightarrow B_{i,j}F_{i,j}\,,\; i,j \in \{1,2\}$ *be distributive laws. Then:*

$$(\lambda_{1,1} \otimes \lambda_{2,1}) \oplus (\lambda_{1,2} \otimes \lambda_{2,2}) \;=\; (\lambda_{1,1} \oplus \lambda_{1,2}) \otimes (\lambda_{2,1} \oplus \lambda_{2,2})$$

Proof. By elementary properties of tuples and cotuples, in particular, by the following law (called the "abides law" in [19]):

$$(f \vartriangle g) \triangledown (h \vartriangle i) \;=\; (f \triangledown h) \vartriangle (g \triangledown i) \qquad \square$$

Example 6. By the above lemma (compare with (5)):

$$\mathsf{lambda} \;=\; (\mathsf{numEval} \oplus \mathsf{addEval}) \otimes (\mathsf{numModn} \oplus \mathsf{addModn}) \tag{6}$$

Examples 5 and 6 illustrate the duality between the functional and OO approaches to *program decomposition*. In functional programming, different cases of the same function, each for a different component of the top-level disjoint union of an algebraic data type, are cotupled by case distinction (c.f. occurrences of $\oplus$ in (6)). In contrast, in OO programming, functions on the same data are tupled into object types (c.f. occurrences of $\otimes$ in (5)).

3.4 The Simple Expression Lemma

We have shown that we may extract a natural transformation from the algebra of a functional fold that can be reused in the coalgebra of an unfold for object construction. It remains to be shown that the functional fold and the OO unfold are indeed semantically equivalent in such a case.

Given a distributive law $\lambda \;:\; FB \longrightarrow BF$, one can define an arrow $\mu F \longrightarrow \nu B$ by the following derivation:

$$\lambda_{\nu B} \circ F\mathsf{out}_B : F\nu B \longrightarrow BF\nu B \tag{7}$$

$$[\![\, \lambda_{\nu B} \circ F\mathsf{out}_B \,]\!]_B : F\nu B \longrightarrow \nu B \tag{8}$$

$$(\!|\, [\![\, \lambda_{\nu B} \circ F\mathsf{out}_B \,]\!]_B \,|\!)_F : \mu F \longrightarrow \nu B \tag{9}$$

An example of (7) is $\mathsf{eitherCoalg}$ in § 2.5.
Dually, the following also defines an arrow $\mu F \longrightarrow \nu B$:

$$B\mathsf{in}_F \circ \lambda_{\mu F} : FB\mu F \longrightarrow B\mu F \tag{10}$$

$$(\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F : \mu F \longrightarrow B\mu F \tag{11}$$

$$[\![\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,]\!]_B : \mu F \longrightarrow \nu B \tag{12}$$

An example is $\mathsf{bothAlg}$ in § 2.5.

The following theorem shows that (12) is equal to (9) and thus, as discussed in § 2.5, establishes a formal correspondence between anamorphically phrased OO programs and catamorphically phrased functional programs.

Theorem 1 ("Simple expression lemma"). *Let* out_B *be the terminal* B*-coalgebra and* in_F *be the initial* F*-algebra. Let* $\lambda : FB \longrightarrow BF$*. Then*

$$(\!|\, [\!(\, \lambda_{\nu B} \circ F\mathsf{out}_B \,)\!]_B \,|\!)_F \quad = \quad [\!(\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,)\!]_B$$

Proof. We show that the right-hand side, $[\!(\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,)\!]_B$, satisfies the universal property of the left-hand side; c.f. (2):

$$[\!(\, \lambda_{\nu B} \circ F\mathsf{out}_B \,)\!]_B \circ F[\!(\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,)\!]_B \quad = \quad [\!(\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,)\!]_B \circ \mathsf{in}_F \quad (13)$$

The calculation is straightforward by a two-fold application of the following rule, called "AnaFusion" in [19]:

$$[\!(\, \varphi \,)\!]_B \circ f \;=\; [\!(\, \psi \,)\!]_B \quad \Leftarrow \quad \varphi \circ f \;=\; Bf \circ \psi \qquad (14)$$

The premise of the rule is precisely the statement that f is a B-coalgebra morphism to φ from ψ. The proof is immediate by compositionality of coalgebra morphisms and uniqueness of the universal arrow. Using this rule we proceed as follows:

$$\begin{array}{ll}
 & [\!(\, \lambda \circ F\mathsf{out}_B \,)\!]_B \circ F[\!(\, (\!|\, B\mathsf{in}_F \;\circ \lambda \,|\!)_F \,)\!]_B \\
= & \{ \text{ By (14) and the following:} \\
 & \quad \lambda \circ F\mathsf{out}_B \circ F[\!(\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,)\!]_B \\
 & = \quad \{ \text{ functor composition } \} \\
 & \quad \lambda \circ F(\mathsf{out}_B \circ [\!(\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,)\!]_B) \\
 & = \quad \{ \; [\!(\, \ldots \,)\!]_B \text{ is a coalgebra morphism } \} \\
 & \quad \lambda \circ F(B[\!(\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,)\!]_B \circ (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F) \\
 & = \quad \{ \; \lambda \text{ is natural } \} \\
 & \quad (BF[\!(\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,)\!]_B) \circ \lambda \circ F(\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \quad \} \\
 & [\!(\, \lambda \circ F(\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,)\!]_B \\
= & \{ \text{ By (14) and the following fact:} \\
 & \quad (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \circ \mathsf{in}_F \\
 & = \quad \{ \; (\!|\, \ldots \,|\!)_F \text{ is a } F\text{-algebra morphism } \} \\
 & \quad B\mathsf{in}_F \circ \lambda \circ F(\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \quad \} \\
 & [\!(\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,)\!]_B \circ \mathsf{in}_F
\end{array}$$

□

4 Classes of Dualizable Folds

The present section illustrates important classes of folds that are covered by the formal development of this paper (including the elaboration to be expected from § 5). For each class of folds, we introduce a variation on the plain fold operation so that the characteristics of the class are better captured. The first argument of a varied fold operation is not a fold algebra formally; it is rather used for composing a proper fold algebra, which is to be passed to the plain fold operation.

4.1 Void Folds

Consider again the Java rendering of the modn function:[5]

```
public abstract void Expr.modn(int v); // Modify literals modulo v
public void Num.modn(int v) { this.value = this.value % v; }
public void Add.modn(int v) { left.modn(v); right.modn(v); }
```

That is, the modn method needs to mutate the value fields of all Num objects, but no other changes are needed. Hence, the imperative OO style suggests to defer to a method without a proper result type, i.e., a void method. In the reading of functional objects, a void method has a result type that is equal to the type parameter of the interface functor. There is a restricted fold operation that captures the idea of voidity in a functional setup:

```
type VExprArg x =
        (Int → x → Int,
         (x → Expr, x → Expr) → x → (Expr, Expr))
vFoldExpr :: VExprArg x →Expr → x → Expr
vFoldExpr a = foldExpr (((.) num . fst a) \/ ((.) add . snd a))
```

The void fold operation takes a product — one type-preserving function for each data variant. The type parameter x enables void folds with extra arguments. For instance, the modn function can be phrased as a void fold with an argument of type Int:

```
modn :: Expr → Int → Expr
modn = vFoldExpr (numModn, addModn)
```

Rewriting a general fold to a void fold requires nothing more than factoring the general fold algebra so that it follows the one that is composed in the vFoldExpr function above. That is, for each constructor, its case preserves the constructor. A void fold must also be sufficiently natural in order to be dualizable; c.f. the next subsection.

4.2 Natural Folds

Consider again the type of the fold algebra for the modn function as introduced in § 2.4:

```
  modnAlg :: ExprAlg (Int → Expr)
≡ modnAlg :: ExprF (Int → Expr) → (Int → Expr)
```

[5] We use a concise OO notation such that the hosting class of an instance method is simply shown as a qualifier of the method name when giving its signature and implementation.

The type admits observation of the precise structure of intermediate results; c.f. the occurrences of Expr. This capability would correspond to unlimited introspection in OO programming (including "instance of" checks and casts). The formal development requires that the algebra must be amenable to factoring as follows:

```
modnAlg = fmapIExprF InExpr . lambda
 where
  lambda :: ExprF (Int → x) → Int → ExprF x
  lambda = ...
```

That is, the algebra is only allowed to observe one layer of functorial structure. In fact, it is easy to see that the actual definition of modnAlg suffices with this restriction. We may want to express that a fold is readily in a "natural form". To this end, we may use a varied fold operation whose argument type is accordingly parametric:[6]

```
type NExprArg x = forall y. ExprF (x → y) → x → ExprF y
nExpr :: NExprArg x →Expr →x → Expr
nExpr a = foldExpr ((.) InExpr . a)
```

It is clear that the type parameter x is used at the type Expr, but universal quantification rules out any exploitation of this fact, thereby enabling the factoring that is required by the formal development. For completeness' sake, we also provide a voidity-enforcing variation; it removes the liberty of replacing the outermost constructor:

```
type VnExprArg x = forall y. (Int→x→Int,(x→y,x→y)→x→(y,y))
vnExpr :: VnExprArg x →Expr →x → Expr
vnExpr a = foldExpr ((.) InExpr . (((.) Left . fst a) \/ ((.) Right . snd a)))
```

The modn function is a void, natural fold:

```
modn :: Expr → Int → Expr
modn = vnExpr (numModn, addModn)
```

The distributive law of the formal development, i.e., $\lambda : FB \longrightarrow BF$, is more general than the kind of natural F-folds that we illustrated above. That is, λ is not limited to type-preserving F-folds, but it uses the extra functor B to define the result type of the F-fold in terms of F. This extra functor is sufficient to cover "constant folds" (such as eval), "paramorphic folds" [18] (i.e., folds that also observe the unprocessed, immediate components) and "tupled folds" (i.e., folds that were composed from separated folds by means of tupling). The source distribution of the paper illustrates all these aspects.

4.3 Free Monadic Folds

The type of the natural folds considered so far implies that *intermediate results are to be combined in exactly one layer of functorial structure*, c.f. the use of ExprF in the result-type position of NExprArg. The formal development will waive this restriction in § 5. Let us motivate the corresponding generalization.

The generalized scheme of composing intermediate results is to arbitrarily nest constructor applications, including the base case of returning one recursive result, as is.

[6] We use a popular Haskell 98 extension for rank-2 polymorphism; c.f. `forall`.

Consider the following function that returns the leftmost expression; its Add case does not replace the outermost constructor; instead, the outermost constructor is dropped:

```
leftmost  ::  Expr → Expr
leftmost  (Num i) = Num i
leftmost  (Add l  r)  = leftmost  l
```

The function cannot be phrased as a natural fold. We can use a plain fold, though:

```
leftmost  = foldExpr (num \/ fst)
```

The following OO counterpart is suggestive:

```
public abstract Expr Expr.leftmost();
public Expr Num.leftmost() { return this;  }
public Expr Add.leftmost() {  return  left . leftmost ();   }
```

Consider the following function for “exponential cloning”; it combines intermediate results in a *nest* of constructor applications (while the leftmost function dropped off a constructor):

```
explode ::  Expr → Expr
explode x@(Num i) = Add x x
explode (Add l  r)  = clone (Add (explode l)  (explode r))
  where clone x = Add x x
```

Again, the function cannot be phrased as a natural fold. We can use a plain fold:

```
explode ::  Expr → Expr
explode =  fFoldExpr ((clone .  freeNum) \/ (clone . freeAdd .  (var  <∗> var)))
  where clone = freeAdd . (id  /\  id)
```

The following OO counterpart uses the “functional” constructors for the object types.

```
public abstract Expr Expr.explode();
public Expr Num.explode() { return new Add(new Num(value),new Num(value)); }
public Expr Add.explode() {
        return new Add(
                new Add(left.explode(), right .explode()),
                new Add(left.explode(), right .explode ()));
}
```

We need a generalized form of natural F-folds where the result of each case may be either of type x, or of type $F\,x$, or of any type $F^n\,x$ for $n \geq 2$. The corresponding union of types is modeled by the free type of F, i.e., the type of free terms (variable terms) over F. This type is also known as the free monad. For instance, the free type of expressions is defined as follows:

```
newtype FreeExpr x = InFreeExpr { outFreeExpr :: Either x (ExprF (FreeExpr x)) }
```

We also assume the following convenience injections:

```
var  = InFreeExpr . Left
term = InFreeExpr . Right
freeNum = term . Left
freeAdd = term . Right
```

Natural folds are generalized as follows:

```
type FExprArg x = forall x. ExprF x → FreeExpr x
fFoldExpr :: FExprArg x →Expr → Expr
fFoldExpr a = foldExpr (collapse . a)
  where
    collapse :: FreeExpr Expr →Expr
    collapse = (id \/ (leaf \/ (fork . (collapse <*> collapse)))) . outFreeExpr
```

(For simplicity, we only consider folds without extra arguments here.) That is, we compose together a fold algebra that first constructs free-type terms from intermediate results, and then collapses the free-type layers by conversion to the recursive closure of the functor at hand. Term construction over the free type is to be dualized to object construction. The two motivating examples can be expressed as “free” natural folds:

```
leftmost = fFoldExpr (freeNum \/ (var . fst ))
explode = fFoldExpr ((clone . freeNum) \/ (clone . freeAdd . (var <*> var)))
  where clone = freeAdd . (id /\ id)
```

4.4 Cofree Comonadic Folds

The type of the natural folds considered so far implies that *the algebra has access to the results of applying the recursive function to immediate components exactly once*. The formal development will waive this restriction in § 5. Let us motivate the corresponding generalization.

In general, we may want to apply the recursive function any number of times to each immediate component. We may think of such expressiveness as an iteration capability. Here is a very simple example of a function that increments twice on the left, and once on the right:

```
leftist :: Expr → Expr
leftist (Num i) = Num (i+1)
leftist (Add l r) = Add ( leftist ( leftist l)) ( leftist r)
```

Such iteration is also quite reasonable in OO programming:

```
public abstract void Expr. leftist ();
public void Num.leftist() { value++; }
public void Add. leftist () { left . leftist (); left . leftist (); right . leftist (); }
```

The simple example only involves a single recursive function, and hence, arbitrary repetitions of that function can be modeled as a *stream* (i.e., a coinductive list). A designated, streaming-enabled fold operation, sFoldExpr, deploys a stream type as the result type. The argument of such a fold operation can select among different numbers of repetitions in the following style:

```
leftist :: Expr → Expr
leftist = sFoldExpr ((+1) <|> ((head . tail . tail ) <*> (head . tail )))
```

Here, head maps to “0 applications”, head . tail maps to “1 application”, head . tail . tail maps to “2 applications”. The streaming-enabled fold operation composes together a fold algebra that produces a stream of results at each level of folding. To this end, we need the coinductive unfold operation for streams:

```
type StreamCoalg x y = y →(x,y)
unfoldStream :: StreamCoalg x y →y → [x]
unfoldStream c = uncurry (:) . (id <∗> unfoldStream c) . c
```

The streaming-enabled fold operation is defined as follows:

```
type SExprArg = forall x. ExprF [x] → ExprF x
sFoldExpr :: SExprArg →Expr →Expr
sFoldExpr a = head . tail . foldExpr a'
    where
      a' :: ExprF [Expr] → [Expr]
      a' = unfoldStream (hd /\ tl)
      hd = InExpr . (id <|> (head <∗> head))
       tl = a . (id <|> (iterate tail <∗> iterate tail ))
```

The argument type of the operation, SExprArg, can be interpreted as follows: given a stream of repetitions for each immediate component, construct a term with a selected number of repetitions for each immediate component position. In fact, the selection of the favored number of repetitions is done by cutting of only the disfavored prefix of the stream (as opposed to actual selection, which would also cut off the postfix). Thereby, iteration is enabled; each step of iteration further progresses on the given stream.

If we look closely, we see that the argument a :: SExprArg is not provided with a *flat* stream of repetitions but rather *a stream of streams* of remaining repetitions; c.f. the use of the standard function, iterate, for coinductive iteration. The type SExprArg protects the nesting status of the stream by universal quantification, thereby avoiding that the stream of remaining repetitions is manipulated in an undue manner such as by reshuffling. For comparison, an unprotective type would be the following:

```
forall x. ExprF [[x]] → ExprF [x]
```

When multiple (mutually recursive) functions are considered, then we must turn from a stream of repetitions to infinite trees of repeated applications; each branch corresponds to the choice of a particular mutation or observation. This tree structure would be modeled by a functor, reminiscent of an interface functor, and the stream type is generalized to the cofree comonad over a functor.

5 The Categorical Model Continued

We will now extend the theory of § 3 in order to cater for the examples of § 4. In particular our notion of a dualizable program, which used to be a simple natural transformation of type $FB \longrightarrow BF$, will be extended to more elaborate distributive laws between a monad and a comonad [2, 25]. This extra structure provides the power needed to cover functional folds that iterate term construction or recursive function application. We show that all concepts from § 3 lift appropriately to (co)monads. We also provide means to construct the generalized distributive laws from more manageable natural transformations, which are the key to programming with the theory and justify the examples of § 4. We eventually generalize the final theorem of § 3 about the semantical correspondence between functions and objects. The used categorical tools are relatively straightforward and well known; [1] is an excellent reference for this section. We only claim originality in their adoption to the semantical correspondence problem of the expression lemma.

5.1 ((Co)free) (Co)monads

We begin with a reminder of the established definitions of (co)monads.

Definition 6. *Let $\mathbf{C}$ be a category. A* ***monad*** *on $\mathbf{C}$ is a triple $\langle T, \mu, \eta\rangle$, where T is an endofunctor on $\mathbf{C}$, $\eta : \mathsf{Id}_{\mathbf{C}} \longrightarrow T$ and $\mu : T^2 \longrightarrow T$ are natural transformations satisfying the following identities:*

$$\mu \circ \mu_T = \mu \circ T\mu \tag{15}$$

$$\mu \circ \eta_T = \mathsf{id} = \mu \circ T\eta \tag{16}$$

A ***comonad*** *is defined dually as a triple $\langle D, \delta, \epsilon\rangle$ where $\delta : D \longrightarrow D^2$ and $\epsilon : D \longrightarrow \mathsf{Id}_{\mathbf{C}}$ satisfy the duals of (15) and (16):*

$$\delta_D \circ \delta = D\delta \circ \delta \tag{17}$$

$$\epsilon_D \circ \delta = \mathsf{id} = D\epsilon \circ \delta \tag{18}$$

To match § 4.3 and § 4.4, we need (co)free (co)monads.

Definition 7. *Let F be an endofunctor on $\mathbf{C}$. Let $F^\dagger_X$ be the functor $F^\dagger_X = X + F\mathsf{Id}$. The* ***free monad*** *of the functor F is a functor T_F that is defined as follows:*

$$\begin{aligned} T_F\, X &= \mu F^\dagger_X \\ T_F\, f &= (\!|\, \mathsf{in}_{F^\dagger_Y} \circ (f + \mathsf{id}) \,|\!)_{F^\dagger_X} \quad , \text{ for } f : X \longrightarrow Y \end{aligned}$$

We make the following definitions:

$$\begin{aligned} \eta_X &= \mathsf{in}_{F^\dagger_X} \circ \iota_1 : X \longrightarrow T_F\, X \\ \tau_X &= \mathsf{in}_{F^\dagger_X} \circ \iota_2 : FT_FX \longrightarrow T_FX \\ \mu_X &= (\!|\, \mathsf{id}_{T_FX} \triangledown \tau_X \,|\!)_{F^\dagger_{T_X}} \end{aligned}$$

Now, η and μ are natural transformations, $\langle T_F, \mu, \eta\rangle$ is a monad.

Example 7. We can think of the type T_FX as of the type of non-ground terms generated by signature F with variables from X; c.f. the Haskell data type FreeExpr in § 4.3. Then, η makes a variable into a term (c.f. var); τ constructs a term from a constructor in F applied to terms (c.f. term), μ is substitution (c.f.collapse, which is actually μ_0; the type collapse : ExprF Expr → Expr reflects the fact that $T_F0 \cong \mu F$).

Cofree comonads are defined dually.

Definition 8. *Let B be an endofunctor on $\mathbf{C}$, let $B^{\ddagger}_{X}$ be the functor $B^{\ddagger}_{X} \ = \ X \times B\mathsf{Id}$. The* ***cofree comonad*** *of the functor B is a functor D_B that is defined as follows:*

$$\begin{aligned} D_B\, X &= \nu B^{\ddagger}_{X} \\ D_B\, f &= [\![\, (f \times \mathsf{id}) \circ \mathsf{out}_{B^{\ddagger}_{X}} \,]\!]_{B^{\ddagger}_{Y}} \\ \epsilon_X &= \pi_1 \circ \mathsf{out}_{B^{\ddagger}_{X}} : D_B\, X \longrightarrow X \\ \xi_X &= \pi_2 \circ \mathsf{out}_{B^{\ddagger}_{x}} : D_B X \longrightarrow B D_B X \\ \delta_X &= [\![\, \mathsf{id}_{D_B X} \vartriangle \xi_X \,]\!]_{B^{\ddagger}_{D_X}} \end{aligned}$$

Example 8. Let us consider a special case: $D_{\mathsf{Id}}\, X$. We can think of this type as the stream of values of type X; c.f. the coinductive use of the Haskell's list-type constructor in § 4.4. This use corresponds to the following cofree comonad:

$$\mathsf{Stream}\ X \ = \ \nu Y.X \times Y \ = \ D_{\mathsf{Id}}\, X$$

Here ϵ is head, ξ is tail, δ is $\mathsf{tails}\ :\ \mathsf{stream}\ X \to \mathsf{stream}\,(\mathsf{stream}\ X)$: the function turning a stream into the stream of its tails, i.e. $\mathsf{iterate\ tail}$. Also note that $D_B 1 \cong \nu B$.

A note on free and non-free monads and comonads In the present paper, we deal exclusively with free constructions (monads and comonads); free constructions have straightforward interpretations as programs. However part of the development, and Theorem 5 in particular, hold in general, and there are interesting examples of non-free constructions arising for instance as quotients with respect to collections of equations.

5.2 (Co)monadic (Co)algebras and Their Morphisms

The notions of folds (and algebras) and unfolds (and coalgebras) lift to monads and comonads. This status will eventually allows us to introduce distributive laws of monads over comonads. We begin at the level of algebras. An algebra of a monad is an algebra of the underlying functor of the monad, which respects the unit and multiplication. Thus:

Definition 9. *Let $\mathbf{C}$ be a category, let $\langle T, \eta, \mu \rangle$ be a monad on $\mathbf{C}$. An T-algebra is an arrow $\alpha : TX \longrightarrow X$ in $\mathbf{C}$ such that the following equations hold:*

$$\mathsf{id}_X \ = \ \alpha \circ \eta_X \tag{19}$$

$$\alpha \circ \mu_X \ = \ \alpha \circ \ T\alpha \tag{20}$$

A T-algebra morphism between T-algebras $\alpha : TA \longrightarrow A$ and $\beta : TB \longrightarrow B$ is an arrow $f : A \longrightarrow B$ in $\mathbf{C}$ such that

$$f \circ \alpha \ = \ \beta \circ Tf$$

The category of T-algebras for a monad[7] *T is denoted $\mathbf{C}^T$.*

[7] We are overloading the notation here as a monad is also a functor. The convention is that when T is a monad, $\mathbf{C}^T$ is the category of algebras of the *monad*.

The notion of the *category of D-coalgebras*, $\mathbf{C}_D$, for a *comonad D* is exactly dual. We record the following important folklore fact about the relation of (co)algebras of a functor and (co)algebras of its free (co)monad; it allows us to compare the present (co)monadic theory to the simple theory of § 3.

Theorem 2. *Let F and B be endofunctors on $\mathbf{C}$. Then the category $\mathbf{C}^F$ of F-algebras is isomorphic to the category $\mathbf{C}^{T_F}$ of algebras of the free monad. And dually: the category $\mathbf{C}_B$ of B-coalgebras is isomorphic to $\mathbf{C}_{D_B}$, the category of coalgebras of the cofree comonad D_B.*

Proof. We show the two constructions: from $\mathbf{C}^F$ to $\mathbf{C}^{T_F}$ and back. To this end, let $\varphi : FX \longrightarrow X$ be an F-algebra, then $\varphi^* = (\!|\, \mathsf{id}_X \triangledown \varphi \,|\!)_{F^\dagger_X}$ is a T_F-algebra. (Read φ with superscript $*$ as "lift φ freely".) In the other direction, given a T_F-algebra α, the arrow $\alpha_* = \alpha \circ \tau_X \circ F\eta_X$ is an F-algebra. (Read α with subscript $*$ as "unlift α freely".) We omit the check that the two directions are inverse and that φ^* is indeed a T_F-algebra. The dual claim follows by duality: for $\psi : X \longrightarrow BX$, $\psi^\propto = [\![\, \mathsf{id}_X \vartriangle \psi \,]\!]_{B^\ddagger_X}$; $\beta_\propto = B\epsilon_X \circ \xi_X \circ \beta$.

Example 9. Consider the functor ExprF. Then $T_{\mathsf{ExprF}}\, X$ is the type of expressions with (formal) variables from X. For $\mathsf{evalAlg} \equiv \mathsf{id} \triangledown (\mathsf{curry}\,(+)) : \mathsf{Int} + \mathsf{Int}^2 \longrightarrow \mathsf{Int}$, $\mathsf{evalAlg}^* : T_{\mathsf{ExprF}}\, \mathsf{Int} \longrightarrow \mathsf{Int}$ recursively evaluates an expression where the variables are integers. For $\mathsf{eval}' : T_{\mathsf{ExprF}}\, \mathsf{Int} \longrightarrow \mathsf{Int}$ that evaluates ("free") expressions, one can reproduce the function $\mathsf{evalAlg} \equiv \mathsf{eval}'_*$ by applying eval' to trivial expressions, provided that eval' is sufficiently uniform (in the sense of equations (19) and (20)).

Example 10. Remember that $D_{\mathsf{Id}} \cong \mathsf{Stream}$. For an Id-coalgebra $k : X \longrightarrow X$, $k^\propto = \mathsf{iterate} : X \longrightarrow \mathsf{Stream}\, X$. And obviously, from a function $\mathsf{itf} : X \longrightarrow \mathsf{stream}\, X$ producing a stream of iterated results of a function, one can reproduce the original function $\mathsf{itf}_\propto$ by taking the second element of the stream.

The theorem, its proof, and the examples illustrate the key operational intuition about algebras of free monads: any algebra of a free monad works essentially in an iterative fashion. It is equivalent to the iteration of a plain algebra, which processes (deep) terms by induction on their structure. If we consider a T_F-algebra $\alpha : T_F X \longrightarrow X$ as a function reducing a tree with leaves from X into a single X, α_* is the corresponding one-layer function which prescribes how each layer of the tree is collapsed given an operator and a collection of collapsed subtrees. This intuition is essential for building an intuition about generalized distributive laws, which are to come shortly, and the programs induced by them.

We continue lifting concepts from § 3.

Lemma 2. *Let 0 be the initial object in $\mathbf{C}$. Then μ_0 is the initial T-algebra in $\mathbf{C}^T$. Dually, δ_1 is the terminal D-coalgebra in $\mathbf{C}_D$.*

Proof. Omitted.

We can now lift the definitions of folds and unfolds.

Definition 10. *Let α be a T-algebra. Then $(\!|\, \alpha \,|\!)_T$ denotes the unique arrow in* **C** *from the initial T-algebra to α. Dually, for a D-coalgebra β and $[\!(\, \beta \,)\!]_D$.*

$$h = (\!|\, \alpha \,|\!)_{T_F} \Leftrightarrow h \circ \mu_0 = \alpha \circ Fh, \quad \textit{and } \alpha \textit{ is a } T\textit{-algebra} \tag{21}$$

$$h = [\!(\, \beta \,)\!]_F \Leftrightarrow \delta_1 \circ h = Fh \circ \beta, \quad \textit{and } \beta \textit{ is a } D\textit{-coalgebra} \tag{22}$$

5.3 Distributive Laws of Monads over Comonads

Distributive laws of monads over comonads, due to J. Beck [2], are liftings of the plain distributive laws of § 3, which had a straightforward computational interpretation, to monads and comonads. Again, they are natural transformations on the functors, but they respect the additional structure of the monad and comonad in question.

The following is standard, here taken from [25].

Definition 11. *Let $\langle T, \eta, \mu\rangle$ be a monad and $\langle D, \varepsilon, \delta\rangle$ be a comonad in a category $\mathcal{C}$. A* ***distributive law*** *of T over D is a natural transformation*

$$\Lambda : TD \longrightarrow DT$$

satisfying the following:

$$\Lambda \circ \eta_D = D\eta \tag{23}$$

$$\Lambda \circ \mu_D = D\mu \circ \Lambda_T \circ T\Lambda \tag{24}$$

$$\varepsilon_T \circ \Lambda = T\varepsilon \tag{25}$$

$$\delta_T \circ \Lambda = D\Lambda \circ \Lambda_D \circ T\delta \tag{26}$$

In the following, we relate "programming" to distributive laws, where we further emphasize the view that programs are represented as natural transformations. First, we show that natural transformations with uses of monads and comonads give rise to distributive laws of monads over comonads; see the following theorem. In this manner, we go beyond the simple natural transformations and distributive laws of § 3, and hence we can dualize more programs.

Theorem 3 (Plotkin and Turi, 1997). *Let F and B be endofunctors. Natural transformations of type*

$$F(\mathsf{Id} \times B) \longrightarrow BT_F \tag{27}$$

or

$$FD_B \longrightarrow B(\mathsf{Id} + F) \tag{28}$$

give rise to distributive laws $\Lambda : T_F D_B \longrightarrow D_B T_F$.

Proof. See [25]. □

The theorem originated in the context of categorical operational semantics [25]; see the related work discussion in § 6. However, the theorem directly applies to our situation of "programming with natural transformations". In the Haskell-based illustrations of § 4, we encountered such natural transformations as arguments of the enhanced fold operations. We did not exhaust the full generality of the typing scheme that is admitted by the theorem, but we did have occurrences of a free monad and a cofree comonad. Here we note that the general types of natural transformations in the above theorem admit paramorphisms [18] (c.f. $F(\mathsf{Id} \times B)$ in (27)) and their duals, *apomorphisms* [27] (c.f. $B(\mathsf{Id} + F)$ in (28)).

Example 11. The type $\mathsf{FExprArg}$ in §4.3 models natural transformations of type $FB \longrightarrow BT_F$ for $F \equiv \mathsf{ExprF}$ and $B \equiv \mathsf{Id}$. Likewise, The type $\mathsf{SExprArg}$ in §4.4 models natural transformations of type $FD_B \longrightarrow BF$, for $F \equiv \mathsf{ExprF}$ and $B \equiv \mathsf{Id}$.

Natural transformations $\lambda : FB \longrightarrow BF$ can be lifted so that Theorem 3 applies:

$$B\tau \circ \lambda_{T_F} \circ FB\eta \circ F\pi_2 : F(\mathsf{Id} \times B) \longrightarrow BT_F \tag{29}$$

$$B\iota_2 \circ BF\epsilon \circ \lambda_{D_B} \circ F\xi : FD_B \longrightarrow B(\mathsf{Id} + F) \tag{30}$$

The following fact is useful:

Lemma 3. *Given a natural transformation $\lambda : FB \longrightarrow BF$, the distributive law constructed by Theorem 3 from (29) is equal to the one constructed from (30).*

Proof. Omitted. See the full report.

The following definition is therefore well-formed.

Definition 12. *For a natural transformation $\lambda : FB \longrightarrow BF$, we denote by $\bar{\lambda}$ the distributive law $T_F D_B \longrightarrow D_B T_F$ given by Theorem 3 either from (29) or (30).*

5.4 Conservativeness of Free Distributive Laws

We can lift simple distributive laws of § 3 to distributive laws of monads over comonads. It remains to establish that such a lifting of distributive laws is semantics-preserving. The gory details follow.

In the simple development of § 3, we constructed algebras and coalgebras from distributive laws by simple projections and injections; c.f. equations (10) and (7). These constructions are lifted as follows:

Lemma 4. *For all X in* **C***, the arrow*

$$D\mu_X \circ \Lambda_{TX} \; : \; TDTX \longrightarrow DTX \tag{31}$$

is a T-algebra. Dually, the arrow

$$\Lambda_{DX} \circ T\delta_X \; : \; TDX \longrightarrow DTDX \tag{32}$$

is a D-coalgebra.

Proof. We must verify that the two T-algebra laws (19) and (20) hold. This can be done by a simple calculation involving just naturality and definitions. □

Now (31) is a T-algebra, and thus by Lemma 2 and Def. 10 it induces an arrow:

$$(\!|\, D\mu_0 \circ \varLambda_{T0} \,|\!)_T \;:\; T0 \longrightarrow DT0 \tag{33}$$

Moreover, when T and D are free on F and B respectively, this is equivalent to

$$(\!|\, D\mu_0 \circ \varLambda_{T0} \,|\!)_T \;:\; \mu F \longrightarrow D\mu F$$

which is by Theorem 2 isomorphic to a B-coalgebra

$$(\!|\, D\mu_0 \circ \varLambda_{T0} \,|\!)_{T_F \propto} \;:\; \mu F \longrightarrow B\mu F \tag{34}$$

Dually for (32):

$$[\!(\, \varLambda_{D1} \circ T\delta_1 \,)\!]_D \;:\; TD1 \longrightarrow D1 \tag{35}$$

$$[\!(\, \varLambda_{D1} \circ T\delta_X \,)\!]_{D_B *} \;:\; F\nu B \longrightarrow \nu B \tag{36}$$

Compare with equations (11) and (8). This shows that *any* distributive law of a free monad over a cofree comonad also gives rise to an algebra for a catamorphisms and a coalgebra for object construction.

Example 12. The functions sFoldExpr in § 4.4, and fFoldExpr in § 4.3 are examples of (34) where B in (34) is fixed to be the identity functor.

The following theorem establishes the essential property that the (co)monadic development of the present section entails the development of § 3.

Theorem 4. *Let F and B be endofunctors on a category* $\mathbf{C}$. *Let $\lambda \,:\, FB \longrightarrow BF$ be a natural transformation. Then the following holds.*

$$[\!(\, \lambda_{\nu B} \circ F\mathsf{out}_B \,)\!]_B = [\!(\, \bar{\lambda}_{D_B 1} \circ T_F\delta_X \,)\!]_{D_B *} \;:\; F\nu B \longrightarrow \nu B \tag{37}$$

$$(\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F = (\!|\, D_B\mu_0 \circ \bar{\lambda}_{T_F 0} \,|\!)_{T_F \propto} \;:\; \mu F \longrightarrow B\mu F \tag{38}$$

Proof. Omitted. See the full report.

5.5 The Generalized Expression Lemma

It remains to lift Theorem 1 (the "simple expression lemma"). As a preparation, we need an analog of the fusion rule.

Lemma 5. *For D-coalgebras α and β:*

$$[\!(\, \alpha \,)\!]_D \circ f \;=\; [\!(\, \beta \,)\!]_D \quad \Leftarrow \quad \alpha \circ f = Df \circ \beta \tag{39}$$

Proof. Immediate by uniqueness of the terminal morphism, as before. □

Theorem 5 ("Generalized expression lemma"). *Let $\langle T, \eta, \mu \rangle$ be a monad and $\langle D, \eta, \delta \rangle$ be a comonad. Let $\varLambda : TD \longrightarrow DT$ be a distributive law of the monad T over D. Then the following holds:*

$$(\!|\, [\!(\, \varLambda_{D1} \circ T\delta_1 \,)\!]_D \,|\!)_T \quad = \quad [\!(\, (\!|\, D\mu_0 \circ \varLambda_{T0} \,|\!)_T \,)\!]_D$$

Proof. The proof has exactly the same structure as that of Theorem 1 except that we have to check at all relevant places that the algebras and coalgebras in question satisfy the additional properties (19) and (20) or their duals, subject to straightforward applications of the monad laws, properties (23) - (26) of distributive laws, and by naturality. We give an outline of the proof of the theorem while omitting the routine checks.

$$\begin{aligned}
& \llbracket\, \Lambda_D \circ T\delta \,\rrbracket_D \circ T\llbracket\, \llparenthesis\, D\mu \circ \Lambda_T \,\rrparenthesis_T \,\rrbracket_D \\
= \quad & \{ \text{ By (39) } \} \\
& \llbracket\, \Lambda_D \circ T\llparenthesis\, D\mu \circ \Lambda_T \,\rrparenthesis_T \,\rrbracket_D \\
= \quad & \{ \text{ By (39) } \} \\
& \llbracket\, \llparenthesis\, D\mu \circ \Lambda_T \,\rrparenthesis_T \,\rrbracket_D \circ \mu
\end{aligned}$$

The conclusion follows by (21). □

6 Related Work

Functional OO Programming

We are not aware of any similar treatment of the correspondence between functional and OO programming. Initially, one would expect some previous work on functional OO programming to be relevant here, such as Moby [8] (an ML-like language with a class mechanism), C# 3.0/VB 9.0/LINQ [3] (the latest .NET languages that incorporate higher-order list-processing functions and more type inference), F# (an ML/OCaml-inspired language that is married with .NET objects), Scala [20] (a Java-derived language with support for functional programming), ML-ART or OCaml [23] (ML with OO-targeting type extensions) — just to mention a few. However, all such work has not revealed the expression lemma. *When functional OO efforts start from a functional language*, then the focus is normally on type-system extensions for subtyping, self, and inheritance, while OO programs are essentially encoded as functional programs, without though relating the encoding results to any "native" functional counterparts. *Dually, when functional OO efforts start from an OO language*, then the focus is normally on translations that eliminate functional idioms, without though relating the translation results to any "native" OO counterpart. (For instance, Scala essentially models a function as a special kind of object.) Our approach specifically leverages the correspondence between functional folds and OO designs based on an idealized composite pattern.

The Expression Problem

Previous work on the expression problem [28] has at best assumed the expression lemma implicitly. The lemma may have been missing because the expression problem classically assumes only very little structure: essentially, there are supposed to be multiple data variants as well as multiple operations on these variants. In contrast, the proposed expression lemma requires more structure, i.e., it requires functional folds or OO designs based on an idealized composite pattern, respectively.

Programming vs. Semantics

We have demonstrated how distributive laws of a functor over a functor (both possibly with additional structure) arise naturally from programming practice with disciplined folds and an idealized composite pattern. By abstraction, we have ultimately arrived at the same notion of adequacy that Turi and Plotkin originally coined for denotational and operational semantics [24, 25]. There, our functional programming side of the picture corresponds to *denotational semantics* and the OO programming side corresponds to *operational semantics*. Our functional/OO programming correspondence corresponds to *adequacy of denotational and operational semantics*. We have provided a simple, alternative, calculational proof geared towards functional programming intuitions. Any further correspondence, for instance of their *operational rules*, is not straightforward. We hypothesize that an elaborated expression lemma may eventually incorporate additional structure that has no direct correspondence in Turi and Plotkin's sense.

More on Distributive Laws

Distributive laws of a functor over a functor (both possibly with additional structure) [2] have recently enjoyed renewed interest. We mention a few of the more relevant contributions. In the context of bialgebraic semantics, Fiore, Plotkin and Turi have worked on languages with binders in a presheaf category [7], and Bartek Klin has worked on recursive constructs [16]. Both theoretical contributions may inspire a model of object structures with cycles and sharing in our interpretation. Modular constructions on distributive laws, including those we leveraged towards the end of § 3.3 have been investigated by Bart Jacobs [13]. More advanced modular constructions may be helpful in the further exploration of the modularity of dualizable programs. Alberto Pardo, Tarmo Uustalu, Varmo Vene, and collaborators have been using distributive laws for recursion and corecursion schemes. For instance, in [26], a generalized coinduction scheme is delivered where a distributive law specifies the pattern of mutual recursion between several functions defined by coinduction. This work seems to be related to coalgebraic OO programming where methods in an interface are (possibly) mutually recursive.

7 Concluding Remarks

We have revealed the expression lemma — a correspondence between OO and functional programs, subject to the assumption that both kinds of programs share a certain decomposition based on structural recursion. The decomposition requirement for functional programs is equivalent to a class of natural folds. The decomposition requirement for OO programs is equivalent to the concept of object structures with part-whole relationships and methods that are in alignment with an idealized composite design pattern. The formal development for comparing functional and OO programs relies on a coalgebraic model of functional objects.

While our development already covers some non-trivial idioms in "dualizable" programming, e.g., iteration of term construction and recursive function application, it still leaves many open questions — in particular, if we wanted to leverage the duality for real-world programs. Hence, one challenge is to generalize the expression lemma and the associated constructions of distributive laws so that the class of dualizable programs is extended. For instance, histomorphisms [15, 26] are not just useful in devising efficient encodings for non-linearly recursive problems, they also generalize access to intermediate results for non-immediate components — very much in the sense of "dotting" into objects and invoking methods on non-immediate components. Another challenge is to admit data structures with sharing and cycles. The use of sharing and cycles is common in OO programming — even without the additional complication of mutable objects. Yet another challenge is to complete the current understanding of the functional/OO correspondence into effective bidirectional refactorings. For instance, in the objects-to-functions direction, such a refactoring would involve non-trivial preconditions on the shape of the OO code, e.g., preconditions to establish absence of sharing, cycles, and mutations, where we may be able to leverage related OO type-system extensions, e.g., for immutability and ownership [4, 6].

Acknowledgments. Ondrej Rypacek would like to thank the CALCO-jnr 2007 referees for feedback that could be leveraged for the present paper. This research has been funded by EPSRC grant EP/D502632/1. The authors are grateful for the constructive and detailed reviews by the MPC 2008 program committee.

References

1. Awodey, S.: Category Theory. Clarendon Press (2006)
2. Beck, J.: Distributive laws. Lecture Notes in Mathematics 80, 119–140 (1969)
3. Bierman, G.M., Meijer, E., Torgersen, M.: Lost in translation: formalizing proposed extensions to C#. In: OOPSLA 2007: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications, pp. 479–498. ACM Press, New York (2007)
4. Birka, A., Ernst, M.D.: A practical type system and language for reference immutability. In: OOPSLA 2004: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 35–49. ACM Press, New York (2004)
5. Chin, W.-N.: Towards an automated tupling strategy. In: PEPM 1993: Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pp. 119–132. ACM Press, New York (1993)
6. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA 1998: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 48–64. ACM Press, New York (1998)
7. Fiore, M., Plotkin, G., Turi, D.: Abstract Syntax and Variable Binding. In: LICS 1999: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science, pp. 193–202. IEEE Press, Los Alamitos (1999)
8. Fisher, K., Reppy, J.: Object-oriented aspects of Moby. Technical report, University of Chicago Computer Science Department Technical Report (TR-2003-10) (July 2003)
9. Fokkinga, M.M.: Tupling and Mutumorphisms. Appeared in: The Squigollist 1(4), 81–82 (1990)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1994)

11. Hu, Z., Iwasaki, H., Takeichi, M.: Deriving structural hylomorphisms from recursive definitions. In: ICFP 1996: Proceedings of the first ACM SIGPLAN international conference on Functional programming, pp. 73–82. ACM Press, New York (1996)
12. Hu, Z., Iwasaki, H., Takeichi, M., Takano, A.: Tupling calculation eliminates multiple data traversals. In: ICFP 1997: Proceedings of the second ACM SIGPLAN international conference on Functional programming, pp. 164–175. ACM Press, New York (1997)
13. Jacobs, B.: Distributive laws for the coinductive solution of recursive equations. Information and Computation 204(4), 561–587 (2006)
14. Jacobs, B.P.F.: Objects and classes, coalgebraically. In: Freitag, B., Jones, C.B., Lengauer, C., Schek, H.J. (eds.) Object-Orientation with Parallelism and Persistence, pp. 83–103. Kluwer Academic Publishers, Dordrecht (1996)
15. Kabanov, J., Vene, V.: Recursion Schemes for Dynamic Programming. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 235–252. Springer, Heidelberg (2006)
16. Klin, B.: Adding recursive constructs to bialgebraic semantics. Journal of Logic and Algebraic Programming 60-61, 259–286 (2004)
17. Launchbury, J., Sheard, T.: Warm fusion: deriving build-catas from recursive definitions. In: FPCA 1995: Proceedings of the seventh international conference on Functional programming languages and computer architecture, pp. 314–323. ACM Press, New York (1995)
18. Meertens, L.G.L.T.: Paramorphisms. Formal Aspects of Computing 4(5), 413–424 (1992)
19. Meijer, E., Fokkinga, M.M., Paterson, R.: Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 124–144. Springer, Heidelberg (1991)
20. Odersky, M.: The Scala Language Specification, Version 2.6, DRAFT, Programming Methods Laboratory, EPFL, Switzerland (December 19, 2007)
21. Pierce, B.C., Turner, D.N.: Simple type-theoretic foundations for object-oriented programming. Journal of Functional Programming 4(2), 207–247 (1994)
22. Reichel, H.: An approach to object semantics based on terminal co-algebras. Mathematical Structures in Computer Science 5(2), 129–152 (1995)
23. Rémy, D.: Programming Objects with ML-ART: An extension to ML with Abstract and Record Types. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 321–346. Springer, Heidelberg (1994)
24. Turi, D.: Functorial Operational Semantics and its Denotational Dual. PhD thesis, Free University, Amsterdam (June 1996)
25. Turi, D., Plotkin, G.D.: Towards a mathematical operational semantics. In: Proceedings 12th Annual IEEE Symposium on Logic in Computer Science, LICS 1997, Warsaw, Poland, 29 June – 2 July 1997, pp. 280–291. IEEE Press, Los Alamitos (1997)
26. Uustalu, T., Vene, V., Pardo, A.: Recursion schemes from comonads. Nordic Journal of Computing 8(3), 366–390 (2001)
27. Vene, V., Uustalu, T.: Functional programming with apomorphisms (corecursion). Proceedings of the Estonian Academy of Sciences: Physics, Mathematics 47(3), 147–161 (1998)
28. Wadler, P.: The expression problem. Message to java-genericity electronic mailing list (November 1998), `http://www.daimi.au.dk/~madst/tool/papers/expression.txt`

Nested Datatypes with Generalized Mendler Iteration: Map Fusion and the Example of the Representation of Untyped Lambda Calculus with Explicit Flattening

Ralph Matthes

Institut de Recherche en Informatique de Toulouse (IRIT)
C. N. R. S. et Université Paul Sabatier (Toulouse III)
118 route de Narbonne, F-31062 Toulouse Cedex 9

Abstract. Nested datatypes are families of datatypes that are indexed over all types such that the constructors may relate different family members. Moreover, the argument types of the constructors refer to indices given by expressions where the family name may occur. Especially in this case of true nesting, there is no direct support by theorem provers to guarantee termination of functions that traverse these data structures.

A joint article with A. Abel and T. Uustalu (TCS 333(1–2), pp. 3–66, 2005) proposes iteration schemes that guarantee termination not by structural requirements but just by polymorphic typing. They are generic in the sense that no specific syntactic form of the underlying datatype "functor" is required. In subsequent work (accepted for the Journal of Functional Programming), the author introduced an induction principle for the verification of programs obtained from Mendler-style iteration of rank 2, which is one of those schemes, and justified it in the Calculus of Inductive Constructions through an implementation in the theorem prover Coq.

The new contribution is an extension of this work to generalized Mendler iteration (introduced in Abel et al, cited above), leading to a map fusion theorem for the obtained iterative functions. The results and their implementation in Coq are used for a case study on a representation of untyped lambda calculus with explicit flattening. Substitution is proven to fulfill two of the three monad laws, the third only for "hereditarily canonical" terms, but this is rectified by a relativisation of the whole construction to those terms.

1 Introduction

Nested datatypes [1] are families of datatypes that are indexed over all types and where different family members are related by the datatype constructors. Let κ_0 stand for the universe of (mono-)types that will be interpreted as sets of computationally relevant objects. Then, let κ_1 be the kind of type transformations, hence $\kappa_1 := \kappa_0 \to \kappa_0$. A typical example would be *List* of kind κ_1, where *List A* is the type of finite lists with elements from type A. But *List*

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 220–242, 2008.

is not a nested datatype since the recursive equation for *List*, i. e., $\mathit{List}\,A = 1 + A \times \mathit{List}\,A$, does not relate lists with different indices. A simple example of a nested datatype where an invariant is guaranteed through its definition are the powerlists [2] (or perfectly balanced, binary leaf trees [3]), with recursive equation $\mathit{PList}\,A = A + \mathit{PList}(A \times A)$, where the type $\mathit{PList}\,A$ represents trees of 2^n elements of A with some $n \geq 0$ (that is not fixed) since, throughout this article, we will only consider the least solutions to these equations. The basic example where variable binding is represented through a nested datatype is a typeful de Bruijn representation of untyped lambda calculus, following ideas of [4,5,6]. The lambda terms with free variables taken from A are given by $\mathit{Lam}\,A$, with recursive equation $\mathit{Lam}\,A = A + \mathit{Lam}\,A \times \mathit{Lam}\,A + \mathit{Lam}(\mathit{option}\,A)$. The first summand gives the variables, the second represents application of lambda terms and the interesting third summand stands for lambda abstraction: An element of $\mathit{Lam}(\mathit{option}\,A)$ (where $\mathit{option}\,A$ is the type that has exactly one more element than A, namely *None*, while the injection of A into $\mathit{option}\,A$ is called *Some*) is seen as an element of $\mathit{Lam}\,A$ through lambda abstraction of that designated extra variable that need not occur freely in the body of the abstraction.

Programming with nested datatypes is possible in the functional programming language Haskell, but this article is concerned with frameworks that guarantee termination of all expressible programs, such as the Coq theorem prover [7] that is based on the Calculus of Inductive Constructions (CIC), presented with details in [8], which only recently (since version 8.1 of Coq) evolved towards a direct support for many nested datatypes that occur in practice, e. g., *PList* and *Lam* are fully supported with recursion and induction principles. Although Coq is officially called the “Coq proof assistant”, it is already in itself[1] a functional programming language. This is certainly not surprising since it is based on an extension of polymorphic lambda calculus (system F^ω), although the default type-theoretic system of Coq since version 8.0 is “pCIC”, namely the Predicative Calculus of (Co)Inductive Constructions. System F^ω is also the framework of the article with Abel and Uustalu [10] that presents a variety of terminating iteration principles on nested datatypes for a notion of nested datatypes that also allows true nesting, which is not supported by the aforementioned recent extension of CIC. A nested datatype will be called “truly nested” (non-linear [11]) if the intuitive recursive equation for the inductive family has at least one summand with a nested call to the family name, i. e., the family name appears somewhere inside the type argument of a family name occurrence of that summand. Our example throughout this article is lambda terms with explicit flattening [12], with the recursive equation

$$\mathit{LamE}\,A = A + \mathit{LamE}\,A \times \mathit{LamE}\,A + \mathit{LamE}(\mathit{option}\,A) + \mathit{LamE}(\mathit{LamE}\,A).$$

The last summand qualifies *LamE* as truly nested datatype: $\mathit{LamE}\,A$ is the type argument to *LamE*.

[1] Not to speak of the program extraction facility of Coq that allows to obtain programs in OCaml, Scheme and Haskell from Coq developments in an automatic way [9].

Even without termination guarantees, the algebra of programming [13] shows the benefits of programming recursive functions in a structured fashion, in particular with iterators: there are equational laws that allow a calculational way of verification. Also for nested datatypes, laws have been important from the beginning [1]. However, no reasoning principles, in particular no induction principles, were studied in [10] on terminating iteration (and coiteration) principles. Newer work by the author [14] integrates rank-2 Mendler iteration into CIC and also justifies an induction principle for them. This is embodied in the system *LNMIt*, the "logic for natural Mendler-style iteration", defined in Section 3.1. This system integrates termination guarantees and calculational verification in one formalism and would also allow dependently-typed programming on top of nested datatypes. Just to recall, termination is also of practical concern with dependent types, namely that type-checking should be decidable: If types depend on object terms, object terms have to be evaluated in order to verify types, as expressed in the convertibility rule. Note, however, that this only concerns evaluation within the definitional equality (i. e., convertibility), henceforth denoted by $\simeq$. Except from the above intuitive recursive equations, $=$ will denote propositional equality throughout: this is the equality type that requires proof and that satisfies the Leibniz principle, i. e., that validity of propositions is not affected by replacing terms by equal (w. r. t. $=$) terms.

The present article is concerned with an extension of *LNMIt* to a system *LNGMIt* that has generalized Mendler-iteration *GMIt*, introduced in [10], in addition to plain Mendler-iteration that is provided by *LNMIt*. Generalized Mendler-iteration is a scheme encompassing generalized folds [11,3,15]. In particular, the efficient folds of [15] are demonstrated to be instances of *GMIt* in [10], and the relation to the gfolds of [11] is discussed there. Perhaps surprisingly, *GMIt* could be explained within F^ω through *MIt*. In a sense, this all boils down to the use of a syntactic form of right Kan extensions as the target constructor G^{κ_1} of the polymorphic iterative functions of type $\forall A^{\kappa_0}.\,\mu F A \to GA$, where μF denotes the nested datatype [10, Section 4.3]. (These Kan extension ideas are displayed in more detail using Haskell in [16], but only in a setting that excludes truly nested datatypes although the type system of current Haskell implementations has no problems with them.)

The main theorem of [14] is trivially carried over to the present setting, i. e., just by the Kan extension trick, the justification of *LNMIt* within CIC with impredicative universe $Set =: \kappa_0$ and propositional proof irrelevance is carried over to *LNGMIt*. Impredicativity of κ_0 is needed here since syntactic Kan extensions use impredicative means for κ_0 in order to stay within κ_1. However, *LNMIt* and *LNGMIt* are formulated as extensions of pCIC with its predicative *Set* as κ_0.

The functions that are defined by a direct application of *GMIt* are uniquely determined (up to pointwise propositional equality) by their recursive equation, under a reasonable extensionality assumption. It is shown when these functions are themselves extensional and when they are "natural", and what natural has to mean for them.

By way of the example of lambda terms with explicit flattening—the truly nested datatype *LamE*—the merits of the general theorems about *LNGMIt* will be studied, mainly by a representation of parallel substitution on *LamE* using *GMIt* and a proof of the monad laws for it. One of the laws fails in general, but it can be established for the hereditarily canonical terms. Their inductive definition (using the inductive definition mechanism of pCIC) refers to the notion of free variables that is obtained from the scheme *MIt*. The whole development for *LamE* can be interpreted within the hereditarily canonical terms, and for those, parallel substitution is shown to be a monad.

All the concepts and results have been formalised in the Coq system, also using module functors having as parameter a module type with the abstract specification of *LNGMIt*, in order to separate the impredicative justification from the predicative formulation and its general consequences that do not depend on an implementation/justification. The Coq code is available [17] and is based on [18].

The following section 2.1 introduces to the Mendler style of obtaining terminating recursive programs and develops the notions of free variables and renaming in the case study. It also discusses extensionality and naturality. Section 2.2 presents *GMIt* and defines a representation of substitution for the case study, leading to a list of properties one would like to prove about it. In Section 3.1, the already existing system *LNMIt* with the logic for *MIt* is properly defined, while Section 3.2 defines the new extension *LNGMIt* as a logic for *GMIt* and proves some general results. The question of naturality for functions that are defined through *GMIt* is addressed in Section 4. General results about proving naturality are presented, one of them is map fusion. Section 5 problematizes the results obtained so far in the case study. Hereditary canonicity is the key notion that allows to pursue that case study. Section 6 concludes.

Acknowledgements: To Andreas Abel for all the joint work in this field and some of his LaTeX macros and the figure I reused from earlier joint papers, and to the referees for their helpful advice that I could only partially integrate in view of the length of this article. In an early stage of the present results, I have benefitted from support by the European Union FP6-2002-IST-C Coordination Action 510996 “Types for Proofs and Programs”.

2 Mendler-Style Iteration

Mendler-style iteration schemes, originally proposed for positive inductive types [19], come with a termination guarantee, and termination is not based on syntactic criteria (that all recursive calls are done with “smaller” arguments) but just on types (called “type-based termination” in [20]).

2.1 Plain Mendler-Style Iteration *MIt*

In order to fit the above intuitive definition of *LamE* into the setting of Mendler-style iteration, the notion of rank-2 functor is needed. Their kind is defined as

$\kappa_2 := \kappa_1 \to \kappa_1$. Any constructor F of kind κ_2 qualifies as rank-2 functor for the moment, and $\mu F : \kappa_1$ denotes the generated family of datatypes. For our example, set

$$LamEF := \lambda X^{\kappa_1} \lambda A^{\kappa_0}.\, A + XA \times XA + X(option\, A) + X(XA)$$

and $LamE := \mu\, LamEF$. In general, there is just one datatype constructor for μF, namely $in : F(\mu F) \subseteq \mu F$, using $X \subseteq Y := \forall A^{\kappa_0}.\, XA \to YA$ for any $X, Y : \kappa_1$ as abbreviation for the respective polymorphic function space. For *LamE*, more clarity comes from the four derived datatype constructors

$$\begin{aligned} varE &: \forall A^{\kappa_0}.\, A \to LamE\, A, \\ appE &: \forall A^{\kappa_0}.\, LamE\, A \to LamE\, A \to LamE\, A, \\ absE &: \forall A^{\kappa_0}.\, LamE(option\, A) \to LamE\, A, \\ flatE &: \forall A^{\kappa_0}.\, LamE(LamE\, A) \to LamE\, A, \end{aligned}$$

where, for example, *flatE* is defined as $\lambda A^{\kappa_0} \lambda e^{LamE(LamE\, A)}.\, in\, A\, (inr\, e)$, with right injection *inr* (here, we assume that + associates to the left), and the other datatype constructors are defined by the respective sequence of injections (see [12] or [10, Example 8.1]).[2] From the explanations of *Lam* in the introduction, it is already clear that *varE*, *appE* and *absE* represent the construction of terms from variable names, application and lambda abstraction in untyped lambda calculus (their representation via a nested datatype has been introduced by [5,6]).

A simple example can be given as follows: Consider the untyped lambda term $\lambda z.\, z\, x_1$ with the only free variable x_1. For future extensibility, think of the allowed set of variable names as *option A* with type variable A. The designated element *None* of *option A* shall be the name for variable x_1. $\lambda z.\, z\, x_1$ is represented by

$$absE(appE\ (varE\ None)\ (varE\ (Some\ None))),$$

with *None* and *Some None* of type *option*(*option A*), hence with the shift that is characteristic of de Bruijn representation. Obviously, the representation is of type $\forall A^{\kappa_0}.\, LamE\, (option\, A)$, and it could have been done in a similar way with *Lam* instead of *LamE*.

In [4], a lambda-calculus interpretation of monad multiplication of *Lam* is given that has the type of *flatE* (with *LamE* replaced by *Lam*), but *here*, this is just a formal (non-executed) form of an integration of the lambda terms that constitute its free variable occurrences into the term itself. We call *flatE explicit flattening.* It does not do anything to the term but is another means of constructing terms.

For an example, consider $t := \lambda y.\, y\, \{\lambda z.\, z\, x_1\}\, \{x_2\}$, where the braces shall indicate that the term inside is considered as the *name of a variable.* If these terms-as-variables were integrated into the term, i. e., if t were "flattened", one would obtain $\lambda y.\, y\, (\lambda z.\, z\, x_1)\, x_2$. This is a trivial operation in this example. In

[2] In Haskell 98, one would define *LamE* through the types of its datatype constructors whose names would be fixed already in the definition of *LamE*.

[14], it is recalled that parallel substitution can be decomposed into renaming, followed by flattening. Under the assumption that substitution is a non-trivial operation, flattening and renaming cannot both be considered trivial. Through the explicit form of flattening, its contribution to the complexity of substitution can be studied in detail.

We want to represent t as term of type $\forall A^{\kappa_0}.\, \mathit{LamE}\,(\mathit{option}(\mathit{option} A))$, in order to accommodate the two free variables x_1, x_2. We instantiate the representation above for $\lambda z.\, z\, x_1$ by $\mathit{option}\, A$ in place of A and get a representation as term $t_1 : \mathit{LamE}\,(\mathit{option}(\mathit{option} A))$. x_2 is represented by

$$t_2 := \mathit{varE}\,(\mathit{Some}\,\mathit{None}) : \mathit{LamE}\,(\mathit{option}(\mathit{option} A)).$$

Now, t shall be represented as the term

$$\mathit{flatE}(\mathit{absE}\, t_3) : \mathit{LamE}\,(\mathit{option}(\mathit{option} A)),$$

hence with $t_3 : \mathit{LamE}\,(\mathit{option}\,(\mathit{LamE}\,(\mathit{option}(\mathit{option} A))))$, defined as

$$t_3 := \mathit{appE}\,\Big(\mathit{appE}\,(\mathit{varE}\,\mathit{None})\,(\mathit{varE}\,(\mathit{Some}\, t_1))\Big)\,(\mathit{varE}\,(\mathit{Some}\, t_2)),$$

that stands for $y\,\{\lambda z.\, z\, x_1\}\,\{x_2\}$. Finally, we can quantify over the type A.

Mendler iteration of rank 2 [10] can be described as follows: There is a constant

$$\mathit{MIt} : \forall G^{\kappa_1}.\, (\forall X^{\kappa_1}.\, X \subseteq G \to FX \subseteq G) \to \mu F \subseteq G$$

and the iteration rule

$$\mathit{MIt}\, G\, s\, A\, (\mathit{in}\, A\, t) \simeq s\,(\mu F)\,(\mathit{MIt}\, G\, s)\, A\, t.$$

In a properly typed left-hand side, t has type $F(\mu F)A$ and s is of type

$$\forall X^{\kappa_1}.\, X \subseteq G \to FX \subseteq G.$$

The term s is called the *step term* of the iteration since it provides the inductive step that extends the function from the type transformation X that is to be viewed as approximation to μF, to a function from FX to G.

Our first example of an iterative function on LamE is the function $\mathit{EFV} : \mathit{LamE} \subseteq \mathit{List}$ (EFV is a shorthand for $\mathit{LamEToFV}$) that gives the list of the names of the free variables (with repetitions in case of multiple occurrences). We want to have the following definitional equations that describe the recursive behaviour (we mostly write type arguments as indices in the sequel):

$$\begin{array}{lcl}
\mathit{EFV}_A\,(\mathit{varE}_A\, a) & \simeq & [a],\\
\mathit{EFV}_A\,(\mathit{appE}_A\, t_1\, t_2) & \simeq & \mathit{EFV}_A\, t_1 + \mathit{EFV}_A\, t_2,\\
\mathit{EFV}_A\,(\mathit{absE}_A\, r) & \simeq & \mathit{filterSome}_A\,(\mathit{EFV}_{\mathit{option}\, A}\, r),\\
\mathit{EFV}_A\,(\mathit{flatE}_A\, e) & \simeq & \mathit{flatten}(\mathit{map}\, \mathit{EFV}_A\,(\mathit{EFV}_{\mathit{LamE}\, A}\, e)).
\end{array}$$

Here, we denoted by $[a]$ the singleton list that only has a as element and by $+$ list concatenation. Moreover, $\mathit{filterSome} : \forall A^{\kappa_0}.\, \mathit{List}(\mathit{option}\, A) \to \mathit{List}\, A$ removes all the occurrences of *None* from its argument and also removes the injection *Some* from A to *option* A from the others. This is nothing but saying that the extra element *None* of *option* A is the variable name that is considered bound in $\mathit{absE}_A\, r$, and that therefore all its occurrences have to be removed from the list of free variables. The set of free variables of $\mathit{flatE}_A\, e$ is the union of the sets of free variables of the free variables of e, which are still elements of *LamE* A. This is expressed by the usual mapping function

$$\mathit{map} : \forall A^{\kappa_0} \forall B^{\kappa_0}.\, (A \to B) \to \mathit{List}\, A \to \mathit{List}\, B$$

for lists and the operation *flatten* : $\forall A^{\kappa_0}.\, \mathit{List}(\mathit{List}\, A) \to \mathit{List}\, A$ that concatenates all the lists in its argument to a single list, and we did not mention the types with which the type arguments of *map* and *flatten* are instantiated.[3] We now argue that there is such a function *EFV*, by showing that it is directly definable as $\mathit{MIt}\, \mathit{List}\, s_{EFV}$ for some closed term

$$s_{EFV} : \forall X^{\kappa_1}.\, X \subseteq \mathit{List} \to \mathit{LamEF} X \subseteq \mathit{List},$$

and therefore, we have the termination guarantee (in [10], a definition of *MIt* within F^ω is given that respects the iteration rule even as reduction from left to right, hence this *is* iteration as is the iteration over the Church numerals of which this is still a generalization). Using an intuitive notion of pattern matching, we define

$$\begin{array}{rll} s_{EFV} := \lambda X^{\kappa_1} \lambda it^{X \subseteq \mathit{List}} \lambda A^{\kappa_0} \lambda t^{\mathit{LamEF}\, X\, A}. \, \mathrm{match}\, t\, \mathrm{with} \\ \mid \mathit{inl}(\mathit{inl}(\mathit{inl}\, a^A)) & \mapsto [a] \\ \mid \mathit{inl}(\mathit{inl}(\mathit{inr}(t_1^{XA}, t_2^{XA}))) & \mapsto it_A\, t_1 + it_A\, t_2 \\ \mid \mathit{inl}(\mathit{inr}\, r^{X(\mathit{option}\, A)}) & \mapsto \mathit{filterSome}(it_{\mathit{option}\, A}\, r) \\ \mid \mathit{inr}\, e^{X(XA)} & \mapsto \mathit{flatten}(\mathit{map}\, it_A\, (it_{XA}\, e)). \end{array}$$

For $\mathit{EFV} := \mathit{MIt}\, \mathit{List}\, s_{EFV}$, the required equational specification is obviously satisfied (since the pattern-matching mechanism behaves properly with respect to definitional equality $\simeq$).[4]

The visible reason why Mendler's style can guarantee termination without any syntactic descent (in which way can the mapping over EFV_A be seen as "smaller"?) is the following: the recursive calls come in the form of uses of *it*, which does not have type $\mathit{LamEF} \subseteq \mathit{List}$ but just $X \subseteq \mathit{List}$, and the type arguments of the datatype constructors are replaced by variants that only mention X instead of *LamE*. So, the definitions have to be uniform in that type transformation variable X, but this is already sufficient to guarantee termination (for

[3] It would have been cleaner to use just one function instead, namely the function *flat_map* : $\forall A^{\kappa_0} \forall B^{\kappa_0}.\, (A \to \mathit{List}\, B) \to \mathit{List}\, A \to \mathit{List}\, B$, where $\mathit{flat\_map}_{A,B}\, f\, \ell$ is the concatenation of all the B-lists $f\, a$ for the elements a of the A-list ℓ. Note that *flatten* is monad multiplication for the list monad and could also be made explicit by a truly nested datatype.

[4] In Haskell 98, our specification of *EFV*, together with its type, can be used as a definition, but no termination guarantee is obtained.

the rank-1 case of inductive *types*, this has been discovered in [21] by syntactic means and, independently, by the author with a semantic construction [22]).

A first interesting question about the results of $\mathit{EFV}_A\, t$ is how they behave with respect to renaming of variables. First, define for any type transformation $X : \kappa_1$ the type of its map term as (from now, omit the kind κ_0 from A and B)

$$\mathit{mon}\, X := \forall A \forall B. (A \to B) \to X A \to X B.$$

Clearly, $\mathit{map} : \mathit{mon}\, \mathit{List}$, but also renaming lamE will have a type of this form, more precisely, $\mathit{lamE} : \mathit{mon}\, \mathit{LamE}$, and $\mathit{lamE}\, f\, t$ has to represent t after renaming every free variable occurrence a in t by $f a$. It would be possible to define lamE by help of *GMIt* introduced in the next section, but it will automatically be available in the systems *LNMIt* and *LNGMIt* that will be described in Section 3. Therefore, we content ourselves in displaying its recursive behaviour (we omit the type arguments to lamE):

$$\begin{array}{lcl} \mathit{lamE}\, f\, (\mathit{varE}_A\, a) & \simeq & \mathit{varE}_B (f a), \\ \mathit{lamE}\, f\, (\mathit{appE}_A\, t_1\, t_2) & \simeq & \mathit{appE}_B\, (\mathit{lamE}\, f\, t_1)\, (\mathit{lamE}\, f\, t_2), \\ \mathit{lamE}\, f\, (\mathit{absE}_A\, r) & \simeq & \mathit{absE}_B (\mathit{lamE}\, (\mathit{option\_map}\, f)\, r), \\ \mathit{lamE}\, f\, (\mathit{flatE}_A\, e) & \simeq & \mathit{flatE}_B \Big(\mathit{lamE}\, \big(\lambda t^{\mathit{LamE}\, A}. \mathit{lamE}\, (\lambda x^A. f x)\, t \big)\, e \Big). \end{array}$$

Here, in the second clause, yet another map term occurs, namely the canonical $\mathit{option\_map} : \mathit{mon}\, \mathit{option}$, so that lamE is called with type arguments $\mathit{option}\, A$ and $\mathit{option}\, B$. In the final clause, the outer call to lamE is with type arguments $\mathit{LamE}\, A$ and $\mathit{LamE}\, B$, while the inner one stays with A and B. The right-hand side in the last case is unpleasantly η-expanded, and one would have liked to see $\mathit{flatE}_B (\mathit{lamE}\, (\mathit{lamE}\, f)\, e)$ instead. However, these two terms are not definitionally equal.

For any $X : \kappa_1$ and map term $m : \mathit{mon}\, X$, define the following proposition

$$\mathit{ext}\, m := \forall A \forall B \forall f^{A \to B} \forall g^{A \to B}. (\forall a^A. f a = g a) \to \forall r^{X A}. m\, A\, B\, f\, r = m\, A\, B\, g\, r.$$

It expresses that m only depends on the extension of its functional argument, which will be called *extensionality* of m in the sequel. In intensional type theory such as CIC, it does not hold in general.[5] In *LNMIt* and *LNGMIt*, the canonical map term $\mathit{map}_{\mu F}$ that comes with μF is extensional. Hence, lamE of our example will be extensional, and the right-hand side in the last case is propositionally equal to the simpler form considered above.

We can now state the "interesting question", mentioned before: Can one prove

$$\forall A \forall B \forall f^{A \to B} \forall t^{\mathit{LamE}\, A}. \mathit{EFV}_B (\mathit{lamE}\, f\, t) = \mathit{map}\, f\, (\mathit{EFV}_A\, t)\ \ ?$$

This is an instance of the question for polymorphic functions j of type $X \subseteq Y$ whether they behave propositionally as a natural transformation from (X, mX)

[5] There are deep studies [23,24,25] on a reconciliation of intensional type theory with extensionality for function spaces. However, we will stick with CIC.

to (Y, mY), given map functions $mX : mon\,X$ and $mY : mon\,Y$. Here, the pair (X, mX) is seen as a functor although no functor laws are required (for the moment). The proposition that defines j to be such a natural transformation is

$$j \in \mathcal{N}(mX,\, mY) := \forall A \forall B \forall f^{A \to B} \forall t^{X\,A}.\, j_B\,(mX\,A\,B\,f\,t) = mY\,A\,B\,f\,(j_A\,t).$$

The system *LNMIt*, described in Section 3.1, allows to answer the above question by showing $\mathit{EFV} \in \mathcal{N}(\mathit{lamE},\, \mathit{map})$. This is in contrast to pure functional programming, where, following [26], naturality is seen as free, namely as a specific instance of parametricity for parametric equality. In intensional type theory such as our *LNMIt* and *LNGMIt* (see Section 3.2), naturality has to be proven on a case by case basis.

By (plain) Mendler iteration *MIt*, one can also define a function $\mathit{eval}{:}\mathit{LamE}{\subseteq}\mathit{Lam}$ that evaluates all the explicit flattenings and thus yields the representation of a usual lambda term [14]. In [14], also *eval* is seen in *LNMIt* to be a natural transformation.

2.2 Generalized Mendler-Style Iteration *GMIt*

We would like to define a representation of substitution on *LamE*. As for *Lam*, the most elegant solution is to define a parallel substitution

$$\mathit{substE} : \forall A \forall B.\, (A \to \mathit{LamE}\,B) \to \mathit{LamE}\,A \to \mathit{LamE}\,B,$$

where for a *substitution rule* $f : A \to \mathit{LamE}\,B$, the term $\mathit{substE}_{A,B}\,f\,t : \mathit{LamE}\,B$ is the result of substituting every variable $a : A$ in the term representation $t : \mathit{LamE}\,A$ by the term $f\,a : \mathit{LamE}\,B$. The operation *substE* would then qualify as Kleisli extension operation of a monad in Kleisli form (a. k. a., bind operation in Haskell).

Evidently, the desired type of *substE* is not of the form $\mathit{LamE} \subseteq G$ for any $G : \kappa_1$. However, it is equivalent (just move the universal quantification over B across an implication) to $\mathit{LamE} \subseteq \mathit{Ran}_{\mathit{LamE}}\,\mathit{LamE}$, with

$$\mathit{Ran}_H\,G := \lambda A \forall B.\, (A \to H B) \to G B$$

for any $H, G : \kappa_1$, which is a syntactic form of a right Kan extension of G along H. This categorical notion has been introduced into the research on nested datatypes in [5], while in [12], it was first used to justify termination of iteration schemes, and in [10], it served as justification of *generalized* Mendler iteration, to be defined next. Its motivation was better efficiency (it covers the efficient folds of [15], see [10]), but visually, this is just hiding of the Kan extension from the user. Technically, this also means a formulation that does not need impredicativity of the universe κ_0 because, only with impredicative κ_0, we have $\mathit{Ran}_H\,G : \kappa_1$. Hence, we stay within pCIC.

The trick is to use the notion of *relativized refined containment* [10]: given $X, H, G : \kappa_1$, define the abbreviation

$$X \leq_H G := \forall A \forall B.\, (A \to H B) \to X A \to G B.$$

Generalized Mendler iteration consists of a constant (the iterator)

$$\mathit{GMIt} : \forall H^{\kappa_1}\forall G^{\kappa_1}.\,(\forall X^{\kappa_1}.\,X \leq_H G \to F X \leq_H G) \to \mu F \leq_H G$$

and the generalized iteration rule

$$\mathit{GMIt}\,H\,G\,s\,A\,B\,f\,(\mathit{in}\,A\,t) \simeq s\,(\mu F)\,(\mathit{GMIt}\,H\,G\,s)\,A\,B\,f\,t.$$

As mentioned before, GMIt can again be justified within F^ω, hence ensuring termination of the rewrite system underlying $\simeq$.

Coming back to substE, we note that its desired type is $\mathit{LamE} \leq_{\mathit{LamE}} \mathit{LamE}$, and in fact, we can define $\mathit{substE} := \mathit{GMIt}\,\mathit{LamE}\,\mathit{LamE}\,s_{\mathit{substE}}$ with

$$s_{\mathit{substE}} : \forall X^{\kappa_1}.\,X \leq_{\mathit{LamE}} \mathit{LamE} \to \mathit{LamEF}\,X \leq_{\mathit{LamE}} \mathit{LamE},$$

given by (note that we start omitting the type parameters at many places)

$$\begin{array}{ll}
\multicolumn{2}{l}{\lambda X^{\kappa_1}\lambda it^{X \leq_{\mathit{LamE}} \mathit{LamE}}\lambda A\lambda B\lambda f^{A \to \mathit{LamE}\,B}\lambda t^{\mathit{LamEF}\,X\,A}.\,\mathrm{match}\,t\,\mathrm{with}} \\
\quad \mid \mathit{inl}(\mathit{inl}(\mathit{inl}\,a^A)) & \mapsto f\,a \\
\quad \mid \mathit{inl}(\mathit{inl}(\mathit{inr}(t_1^{XA}, t_2^{XA}))) & \mapsto \mathit{appE}\,(it_{A,B}\,f\,t_1)\,(it_{A,B}\,f\,t_2) \\
\quad \mid \mathit{inl}(\mathit{inr}\,r^{X(\mathit{option}\,A)}) & \mapsto \mathit{absE}(it_{\mathit{option}\,A,\mathit{option}\,B}\,(\mathit{liftE}\,f)\,r) \\
\quad \mid \mathit{inr}\,e^{X(XA)} & \mapsto \mathit{flatE}(it_{XA,\mathit{LamE}\,B}(\mathit{varE}_{\mathit{LamE}\,B} \circ (it_{A,B}\,f))\,e)).
\end{array}$$

Here, we used an analogue of lifting for Lam in [6],

$$\mathit{liftE} : \forall A \forall B.\,(A \to \mathit{LamE}\,B) \to \mathit{option}\,A \to \mathit{LamE}(\mathit{option}\,B),$$

definable by pattern-matching with properties

$$\begin{array}{ll}
\mathit{liftE}_{A,B}\,f\,\mathit{None} & \simeq \mathit{varE}_{\mathit{option}\,B}\mathit{None}, \\
\mathit{liftE}_{A,B}\,f\,(\mathit{Some}\,a) & \simeq \mathit{lamE}\,\mathit{Some}\,(f a),
\end{array}$$

where renaming lamE is essential.

Note that $\mathit{varE}_{\mathit{LamE}\,B} \circ (it_{A,B}\,f)$ has type $XA \to \mathit{LamE}(\mathit{LamE}\,B)$ (the infix operator $\circ$ denotes composition of functions). From the point of view of clarity of the definition, we would have much preferred $\mathit{flatE}\,(\mathit{lamE}\,(it_{A,B}\,f)\,e)$ to the term in the last clause of the definition of s_{substE}. It would only type-check *after* instantiating X with LamE, hence generalized Mendler iteration cannot accept this alternative. However, a system of sized nested datatypes [27] could assign more informative types to lamE in order to solve this problem, but there do not yet exist systematic means of program verification for them.

Our definition only satisfies

$$\mathit{substE}\,f\,(\mathit{flatE}\,e) \simeq \mathit{flatE}(\mathit{substE}(\mathit{varE} \circ (\mathit{substE}\,f))\,e),$$

to be seen immediately from the generalized iteration rule (assuming again proper $\simeq$-behaviour of pattern matching). Note that $\mathit{substE}\,f\,(\mathit{varE}\,a) \simeq f a$ is already the verification of the first of the three monad laws for the purported monad $(\mathit{LamE}, \mathit{varE}, \mathit{substE})$ in Kleisli form (where varE is the unit of the monad).

The following will be provable about substE in the system LNGMIt, where we mean the universal (and well-typed) closure of all statements:

1. $(\forall a^A.\, f a = g a) \to \mathit{substE}\, f\, t = \mathit{substE}\, g\, t$
2. $(\forall a^A.\, a \in \mathit{EFV}\, t \to f a = g a) \to \mathit{substE}\, f\, t = \mathit{substE}\, g\, t$
3. $\mathit{lamE}\, g\, (\mathit{substE}\, f\, t) = \mathit{substE}\, (\, (\mathit{lamE}\, g) \circ f)\, t$
4. $\mathit{substE}\, g\, (\mathit{lamE}\, f\, t) = \mathit{substE}\, (g \circ f)\, t$
5. $\mathit{substE}\, g\, (\mathit{substE}\, f\, t) = \mathit{substE}\, ((\mathit{substE}\, g) \circ f)\, t$
6. $\mathit{EFV}(\mathit{substE}\, f\, t) = \mathit{flatten}(\mathit{map}\, (\mathit{EFV} \circ f)\, (\mathit{EFV}\, t))$

The first is extensionality, the second refined extensionality, the third and fourth are the two halves of naturality (number 4 appears to be an instance of map fusion, as studied in [15]), the fifth is one of the other two monad laws, and the last a means to express that *EFV* is a monad morphism from *LamE* (that does *not* satisfy the last remaining monad law) to *List*. An easy consequence from it is $b \in \mathit{EFV}(\mathit{substE}\, f\, t) \to \exists a.\, a \in \mathit{EFV}\, t \wedge b \in \mathit{EFV}(f a)$. This consequence and the first five statements are all intuitively true for substitution, renaming and the enumeration of free variables, and they were all known for *Lam*, hence without explicit flattening. The point here is that also the truly nested datatype *LamE* can be given a logic that allows such proofs within intensional type theory, hence in a system with static termination guarantee, interactive program construction (in implementations such as Coq) and no need to *represent* the programs in a programming logic: the program's behaviour with respect to $\simeq$ is directly available.

3 Logic for Natural Generalized Mendler-Style Iteration

First, we recall *LNMIt* from [14], then we extend it by *GMIt* and its definitional rules in order to obtain its extension *LNGMIt*.

3.1 *LNMIt*

In *LNMIt*, for a nested datatype μF, we require that $F : \kappa_2$ preserves *extensional functors*. In pCIC, we may form for $X : \kappa_1$ the dependently-typed record $\mathcal{E}X$ that contains a map term $m : \mathit{mon}\, X$, a proof e of extensionality of m, i. e., of $\mathit{ext}\, m$, and proofs f_1, f_2 of the first and second functor laws for (X, m), defined by the propositions

$$\mathit{fct}_1\, m := \forall A \forall x^{XA}.\, m\, A\, A\, (\lambda y.y)\, x = x,$$
$$\mathit{fct}_2\, m := \forall A \forall B \forall C\, \forall f^{A \to B}\, \forall g^{B \to C}\, \forall x^{XA}.\, m\, A\, C\, (g \circ f)\, x = m\, B\, C\, g\, (m\, A\, B\, f\, x).$$

Given a record *ef* of type $\mathcal{E}X$, Coq's notation for its field m is $m\, \mathit{ef}$, and likewise for the other fields. We adopt this notation instead of the more common $\mathit{ef}.m$. Preservation of extensional[6] functors for F is required in the form of a term of type $\forall X^{\kappa_1}.\, \mathcal{E}\, X \to \mathcal{E}(FX)$, and *LNMIt* is defined to be pCIC with $\kappa_0 :=$ *Set*, extended by the constants and rules of Figure 1, adopted from [14]. In

[6] While the functor laws are certainly an important ingredient of program verification, the extensionality requirement is more an artifact of our intensional type theory, as discussed in Section 2.1.

Parameters:

$$\begin{array}{ll} F & : \kappa_2 \\ Fp\mathcal{E} & : \forall X^{\kappa_1}.\, \mathcal{E}X \to \mathcal{E}(FX) \end{array}$$

Constants:

$$\begin{array}{ll} \mu F & : \kappa_1 \\ map_{\mu F} & : mon(\mu F) \\ In & : \forall X^{\kappa_1}\, \forall ef^{\mathcal{E}X} \forall j^{X \subseteq \mu F}.\, j \in \mathcal{N}(m\, ef,\, map_{\mu F}) \to FX \subseteq \mu F \\ MIt & : \forall G^{\kappa_1}.\, (\forall X^{\kappa_1}.\, X \subseteq G \to FX \subseteq G) \to \mu F \subseteq G \\ \mu FInd & : \forall P : \forall A.\, \mu F A \to Prop.\, \Big(\forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall j^{X \subseteq \mu F} \forall n^{j \in \mathcal{N}(m\, ef,\, map_{\mu F})}. \\ & \qquad \big(\forall A \forall x^{XA}.\, P_A(j_A\, x)\big) \to \forall A \forall t^{FXA}.\, P_A(In\, ef\, j\, n\, t)\Big) \\ & \qquad\qquad \to \forall A \forall r^{\mu F A}.\, P_A\, r \end{array}$$

Rules:

$$\begin{array}{l} map_{\mu F}\, f\, (In\, ef\, j\, n\, t) \simeq In\, ef\, j\, n\, (m(Fp\mathcal{E}\, ef)\, f\, t) \\ MIt\, s\, (In\, ef\, j\, n\, t) \simeq s\, (\lambda A.\, (MIt\, s)_A \circ j_A)\, t \\ \lambda A \lambda x^{\mu F A}.\, (MIt\, s)_A\, x \simeq MIt\, s \end{array}$$

Fig. 1. Specification of *LNMIt* as extension of pCIC

LNMIt, one can show the following theorem [14, Theorem 3] about canonical elements: There are terms $ef_{\mu F} : \mathcal{E}\mu F$ and $InCan : F(\mu F) \subseteq \mu F$ (the *canonical* datatype constructor that constructs canonical elements) such that the following convertibilities hold:

$$\begin{aligned} m\, ef_{\mu F} &\simeq map_{\mu F}, \\ map_{\mu F}\, f\, (InCan\, t) &\simeq InCan(m\, (Fp\mathcal{E}\, ef_{\mu F})\, f\, t), \\ MIt\, s\, (InCan\, t) &\simeq s\, (MIt\, s)\, t. \end{aligned}$$

(The proof of this theorem needs the induction rule $\mu FInd$ in order to show that $map_{\mu F}$ is extensional and satisfies the functor laws. These proofs enter $ef_{\mu F}$, and In can then be instantiated with $X := \mu F$, $ef := ef_{\mu F}$ and j the identity on μF with its trivial proof of naturality, to yield the desired *InCan*.)

This will now be related to the presentation in Section 2.1: The datatype constructor *In* is way more complicated than our previous *in*, but we get back *in* in the form of *InCan* that only constructs the "canonical elements" of the nested datatype μF. The map term $map_{\mu F}$ for μF, which does renaming in our example of *LamE*, as demonstrated in Section 2.1, is an integral part of the system definition since it occurs in the type of *In*. This is a form of simultaneous induction-recursion [28], where the inductive definition of μF is done simultaneously with the recursive definition of $map_{\mu F}$. The Mendler iterator *MIt* has not been touched at all; there is just a more general iteration rule that also covers non-canonical elements, but for the canonical elements, we get the same behaviour, i. e., the same equation with respect to $\simeq$. The crucial part is the induction principle $\mu FInd$, where *Prop* denotes the universe of propositions (all our propositional equalities and their universal quantifications belong to it). Without access to the argument n that assumes naturality of j as a transformation from $(X, m\,ef)$ to $(\mu F, map_{\mu F})$, one would not be able to prove naturality of $MIt\,s$, i. e., of iteratively defined functions on the nested datatype μF. The author is not aware of ways how to avoid non-canonical elements and nevertheless have an induction principle that allows to establish naturality of $MIt\,s$ [14, Theorem 1].

The system *LNMIt* can be defined within CIC with impredicative *Set*, extended by the principle of proof irrelevance, i. e., by $\forall P : Prop\, \forall p_1^P \forall p_2^P .\, p_1 = p_2$. This is the main result of [14], and it is based on an impredicative construction of simultaneous inductive-recursive definitions by Capretta [29] that could be extended to work for this situation. It is also available in the form of a Coq module [18] that allows to benefit from the evaluation of terms in Coq. For this, it is crucial that convertibility in *LNMIt* implies convertibility in that implementation.

The “functor” *LamEF* is easily seen to fulfill the requirement of *LNMIt* to preserve extensional functors (using [14, Lemma 1 and Lemma 2]). As mentioned in Section 2.1, *LNMIt* allows to prove that $EFV \in \mathcal{N}(lamE,\, map)$, and this is an instance of [14, Theorem 1].

3.2 *LNGMIt*

Let *LNGMIt* be the extension of *LNMIt* by the constant *GMIt* from section 2.2,

$$GMIt : \forall H^{\kappa_1} \forall G^{\kappa_1} .\, (\forall X^{\kappa_1} .\, X \leq_H G \to F X \leq_H G) \to \mu F \leq_H G,$$

and the following two rules:

$$\begin{array}{l} GMIt_{H,G}\, s\, f\, (In\, ef\, j\, n\, t) \simeq s\, (\lambda A \lambda B \lambda f^{A \to HB} .\, (GMIt_{H,G}\, s\, A\, B\, f) \circ j_A)\, f\, t, \\ \lambda A \lambda B \lambda f^{A \to HB} \lambda x^{\mu F A} .\, GMIt_{H,G}\, s\, A\, B\, f\, x \simeq GMIt_{H,G}\, s. \end{array}$$

Theorem [14, Theorem 3] about $ef_{\mu F}$ and *InCan* for *LNMIt* immediately extends to *LNGMIt* and yields the following additional convertibility:

$$GMIt\, s\, f\, (InCan\, t) \simeq s\, (GMIt\, s)\, f\, t,$$

which has this concise form only because of the η-rule for *GMIt* that was made part of *LNGMIt*. Thus, we get back the original behaviour of *GMIt* described in Section 2.2, but with the derived datatype constructor *InCan* instead of the *defining* datatype constructor *in*.

Lemma 1. *The system LNGMIt can be defined within LNMIt if the universe κ_0 of computationally relevant types is impredicative.*

Proof. The proof is nothing but the observation that the embedding of GMIt^ω into MIt^ω of [10, Section 4.3] extends for our situation of a rank-2 inductive constructor μF to non-canonical elements, i. e., the full datatype constructor *In* instead of only *in*, considered in that work: define for $H, G : \kappa_1$ the terms

$$\begin{array}{ll} toGRan & := \lambda X^{\kappa_1} \lambda h^{X \leq_H G} \lambda A \lambda x^{X A} \lambda B \lambda f^{A \to HB} .\, h\, A\, B\, f\, x, \\ fromGRan & := \lambda X^{\kappa_1} \lambda h^{X \subseteq Ran_H\, G} \lambda A \lambda B \lambda f^{A \to HB} \lambda x^{X A} .\, h\, A\, x\, B\, f. \end{array}$$

These terms establish the logical equivalence of $X \leq_H G$ and $X \subseteq \mathit{Ran}_H\, G$:

$$\begin{array}{ll} \mathit{toGRan} & : \forall X^{\kappa_1}.\, X \leq_H G \to X \subseteq \mathit{Ran}_H\, G, \\ \mathit{fromGRan} & : \forall X^{\kappa_1}.\, X \subseteq \mathit{Ran}_H\, G \to X \leq_H G. \end{array}$$

Define for a step term $s : \forall X^{\kappa_1}.\, X \leq_H G \to FX \leq_H G$ for $\mathit{GMIt}_{H,G}$ the step term s' for $\mathit{MIt}_{\mathit{Ran}_H\, G}$ as follows:

$$s' := \lambda X^{\kappa_1} \lambda h^{X \subseteq \mathit{Ran}_H\, G}.\, \mathit{toGRan}_{FX}\,(s_X\,(\mathit{fromGRan}_X\, h)).$$

Then, we can define

$$\mathit{GMIt}_{H,G}\, s := \mathit{fromGRan}_{\mu F}\,(\mathit{MIt}_{\mathit{Ran}_H\, G}\, s')$$

and readily observe that the main definitional rule for *GMIt* in *LNGMIt* is inherited from that of *MIt* in *LNMIt* and that the other rule is immediate from the definition.[7] Impredicativity of κ_0 is needed to have $\mathit{Ran}_H\, G : \kappa_1$, as mentioned in Section 2.2. □

Corollary 1. *The system LNGMIt can be defined within CIC with impredicative Set, extended by the principle of propositional proof irrelevance, i. e., by* $\forall P : \mathit{Prop}\, \forall p_1^P \forall p_2^P.\, p_1 = p_2$.

Proof. Use the the previous lemma and the main theorem of [14] that states the same property of *LNMIt*.

[14] is more detailed about how much proof irrelevance is needed for the proof.

Lemma 2 (Uniqueness of ***GMIt s*****).** *Assume* $H, G : \kappa_1$, $s : \forall X^{\kappa_1}.\, X \leq_H G \to FX \leq_H G$ *and* $h : \mu F \leq_H G$ *(the candidate for being GMIt s). Assume further the following extensionality property of s (s only depends on the extension of its first function argument, but in a way adapted to the parameter f):*

$$\forall X^{\kappa_1} \forall g, h : X \leq_H G.\, (\forall A \forall B \forall f^{A \to HB} \forall x^{XA}.\, g\, f\, x = h\, f\, x) \to \\ \forall A \forall B \forall f^{A \to HB} \forall y^{FXA}.\, s\, g\, f\, y = s\, h\, f\, y.$$

Assume finally that h satisfies the equation for GMIt s:

$$\forall X^{\kappa_1} \forall \mathit{ef}^{\mathcal{E}X} \forall j^{X \subseteq \mu F} \forall n^{j \in \mathcal{N}(m\, \mathit{ef},\, \mathit{map}_{\mu F})} \forall A \forall B \forall f^{A \to HB} \forall t^{FXA}. \\ h_{A,B}\, f\,(\mathit{In}\, \mathit{ef}\, j\, n\, t) = s\,(\lambda A \lambda B \lambda f^{A \to HB}.\,(h_{A,B}\, f) \circ j_A)\, f\, t.$$

Then, $\forall A \forall B \forall f^{A \to HB} \forall r^{\mu F\, A}.\, h_{A,B}\, f\; r = \mathit{GMIt}\, s\, f\, r$.

Proof. By the induction principle $\mu \mathit{FInd}$, as for [14, Theorem 2].

Given type constructors X, H, G, the type $X \leq_H G$ has an embedded function space, so there is the natural question whether an inhabitant h of $X \leq_H G$ only

[7] Strictly speaking, we have to define *GMIt* itself, but this can be done just by abstracting over G, H and s that are only parameters of the construction.

depends on the extension of this function parameter. This is expressed by the proposition (*gext* stands for generalized extensionality)

$$gext\,h := \forall A \forall B \forall f, g : A \to HB.\, (\forall a^A.\, f a = g a) \to \forall r^{XA}.\, h_{A,B}\, f\, r = h_{A,B}\, g\, r.$$

The earlier definition of *ext* is the special instance where X and G coincide and where H is the identity type transformation $Id_{\kappa_0} := \lambda A.\, A$.

Given type constructors H, G and a term $s : \forall X^{\kappa_1}.\, X \leq_H G \to FX \leq_H G$, we say that *s preserves extensionality* if $\forall X^{\kappa_1} \forall h^{X \leq_H G}.\, gext\, h \to gext(s\, h)$ holds.

Lemma 3 (Extensionality of $GMIt\,s$**).** *Assume type constructors* H, G *and a term* $s : \forall X^{\kappa_1}.\, X \leq_H G \to FX \leq_H G$ *that preserves extensionality in the above sense. Then* $GMIt\,s : \mu F \leq_H G$ *is extensional, i. e.,* $gext(GMIt\,s)$ *holds.*

Proof. An easy application of $\mu FInd$.

Coming back to the representation *substE* of substitution on *LamE* from Section 2.2, straightforward reasoning shows that s_{substE} preserves extensionality, hence Lemma 3 yields *gext substE*, which proves the first item in the list on page 230. Its refinement, namely the second item in that list,

$$(\forall a^A.\, a \in EFV\, t \to f a = g a) \to substE\, f\, t = substE\, g\, t,$$

needs a direct proof by the induction principle $\mu FInd$, where the behaviour of EFV on non-canonical elements plays an important role, but is nevertheless elementary.

4 Naturality in *LNGMIt*

In order to establish an extension of the map fusion law of [15], a notion of naturality for functionals $h : X \leq_H G$ has to be introduced. We first treat the case where H is the identity Id_{κ_0}. In this case, we omit the argument for H from $X \leq_H G$ and only write $X \leq G$. Assume a function $h : X \subseteq G$ and map terms $m_X : mon\,X$ and $m_G : mon\,G$. Figure 2, which is strongly inspired by [12, Figure 1], recalls naturality, i. e., $h \in \mathcal{N}(m_X, m_G)$ is displayed in the form of a commuting diagram (where commutation means pointwise propositional equality of the compositions) for any A, B and $f : A \to B$. The diagonal marked by $h\,f$ in Figure 2 can then be defined by either $(m_G\, f) \circ h_A$ or $h_B \circ (m_X\, f)$, and this yields a functional of type $\forall A \forall B.\, (A \to B) \to\, XA \to GB$, again called h in [30, Exercise 5 on page 19]. Its type is more concisely expressed as $X \leq G$. The exercise in [30] (there expressed in pure category-theoretic terms) can be seen to establish a naturality-like diagram of the functional h. Namely, also the diagram in Figure 3 commutes for all A, B, C, $f : A \to B$ and $g : B \to C$. Moreover, from a functional h for which the second diagram commutes, one obtains in a unique way a natural transformation h from X to G with h_A being $h\, id_A$. In category theory, this is a simple exercise, but in our intensional setting, this allows to define naturality for any $X, G : \kappa_1$, $m_X : mon\,X$, $m_G : mon\,G$ and $h : X \leq G$.

$$\begin{tikzcd}
A \arrow[r, "f"] & B \\
X \arrow[r, "h", "\subseteq"'] & G
\end{tikzcd}
\qquad
\begin{tikzcd}
X\,A \arrow[r, "h\,A"] \arrow[d, "m_X\,f"'] \arrow[dr, dotted, "h\,f"] & G\,A \arrow[d, "m_G\,f"] \\
X\,B \arrow[r, "h\,B"] & G\,B
\end{tikzcd}$$

Fig. 2. Naturality of $h : X \subseteq G$

$$\begin{tikzcd}
A \arrow[r, "f"] & B \\
B \arrow[r, "g"] & C \\
X \arrow[r, "h", "\leq"'] & G
\end{tikzcd}
\qquad
\begin{tikzcd}
X\,A \arrow[r, "h\,f"] \arrow[d, "m_X\,f"'] \arrow[dr, "h(g\circ f)"] & G\,B \arrow[d, "m_G\,g"] \\
X\,B \arrow[r, "h\,g"] & G\,C
\end{tikzcd}$$

Fig. 3. Naturality of $h : X \leq G$

Definition 1 (Naturality of $h : X \leq G$). *Given $X, G : \kappa_1$, $m_X : mon\,X$, $m_G : mon\,G$ and $h : X \leq G$, the functional h is called natural with respect to m_X and m_G if it satisfies the following two laws:*

1. $\forall A \forall B \forall C \forall f^{A\to B} \forall g^{B\to C} \forall x^{XA}.\, m_G\,g\,(h_{A,B}\,f\,x) = h_{A,C}\,(g\circ f)\,x$
2. $\forall A \forall B \forall C \forall f^{A\to B} \forall g^{B\to C} \forall x^{XA}.\, h_{B,C}\,g\,(m_X\,f\,x) = h_{A,C}\,(g\circ f)\,x$

Mac Lane's exercise [30] can readily be extended to the generality of $X \leq_H G$, with arbitrary H, and a function $h : X \circ H \subseteq G$, but with less pleasing diagrams. We therefore content ourselves with an algebraic description of the parts we need for *LNGMIt*.

Definition 2 (Naturality of $h : X \leq_H G$). *Given $X, H, G : \kappa_1$ and $h : X \leq_H G$, define the two parts of naturality of h as follows: If $m_H : mon\,H$ and $m_G : mon\,G$, define the first part $gnat_1\ m_H\,m_G\,h$ by*

$$\forall A \forall B \forall C \forall f^{A\to HB} \forall g^{B\to C} \forall x^{XA}.\, m_G\,g\,(h_{A,B}\,f\,x) = h_{A,C}\,((m_H\,g)\circ f)\,x \ .$$

If $m_X : mon\,X$, define the second part $gnat_2\ m_X\,h$ by

$$\forall A \forall B \forall C \forall f^{A\to B} \forall g^{B\to HC} \forall x^{XA}.\, h_{B,C}\,g\,(m_X\,f\,x) = h_{A,C}\,(g\circ f)\,x \ .$$

Since Id_{κ_0} has the map term $\lambda A \lambda B \lambda f^{A\to B} \lambda x^A.\, f x$, Definition 1 is an instance of Definition 2.

The backwards direction of Mac Lane's exercise for our generalization is now mostly covered by the following lemma.

Lemma 4. *Given $X, H, G : \kappa_1$, $m_X : mon\, X$, $m_H : mon\, H$, $m_G : mon\, G$ and $h : X \leq_H G$ such that $gnat_1\ m_H\, m_G\, h$ and $gnat_2\ m_X\, h$ hold, the function $h^{\subseteq} := \lambda A \lambda x^{X(HA)}.\, h_{HA,A}\, (\lambda y^{HA}.\, y)\, x : X \circ H \subseteq G$ is natural: $h^{\subseteq} \in \mathcal{N}(m_X \star m_H,\, m_G)$. Here, $m_X \star m_H$ denotes the canonical map term for $X \circ H$, obtained from m_X and m_H.*

Proof. Elementary.

Thus, finally, one can define and argue about functions of type $(\mu F) \circ H \subseteq G$ through $(GMIt\, s)^{\subseteq}$.

Lemma 5 (First part of naturality of *GMIt s***).** *Given $H, G : \kappa_1$, map terms $m_H : mon\, H$, $m_G : mon\, G$ and a term $s : \forall X^{\kappa_1}.\, X \leq_H G \to FX \leq_H G$ that preserves extensionality. Assume further*

$$\forall X^{\kappa_1} \forall h^{X \leq_H G}.\, \mathcal{E}\, X \to gext\, h \to gnat_1\ m_H\, m_G\, h \to gnat_1\ m_H\, m_G\, (s\, h).$$

Then, GMIt s satisfies the first part of naturality, i. e., $gnat_1\ m_H\, m_G\, (GMIt\, s)$.

Proof. Induction with *μFInd*. The proof does *not* use the naturality of argument j, provided by the context of the induction step. Preservation of extensionality is used in order to apply Lemma 3 for the function representing the recursive calls, because that function becomes the h of the main assumption on s.

As an instance of this lemma, one can prove the third item in the list on page 230 on properties of *substE*.

Theorem 1 (Second part of naturality of *GMIt s***—map fusion).** *Given $H, G : \kappa_1$ and a term $s : \forall X^{\kappa_1}.\, X \leq_H G \to FX \leq_H G$ that preserves extensionality. Assume further*

$$\forall X^{\kappa_1} \forall h^{X \leq_H G} \forall ef^{\mathcal{E}\, X}.\, gext\, h \to gnat_2\, (m\, ef)\, h \to gnat_2\, (m\, (Fp\mathcal{E}\, ef))\, (s\, h)\ .$$

Then, GMIt s satisfies the second part of naturality, i. e., $gnat_2\ map_{\mu F}\, (GMIt\, s)$.

Proof. Induction with *μFInd*. Again, we have to use Lemma 3 for the function

$$h := \lambda A \lambda B \lambda f^{A \to HB}.\, (GMIt_{H,G}\, s\, A\, B\, f) \circ j_A$$

representing the recursive calls in the right-hand side of the rule for *GMIt* in the definition of *LNGMIt*. Since we also have to provide a proof of $gnat_2\, (m\, ef)\, h$, we crucially need naturality of j that comes with the induction principle.

Although the proof is quite simple (again, see the full proof in the Coq development [17]), this is the main point of the complicated system *LNGMIt* with its inductive-recursive nature: ensure naturality to be available for j inside the inductive step of reasoning on μF. One might wonder whether this theorem could be an instance of [14, Theorem 1], using the definition of *GMIt* in Lemma 1 for impredicative κ_0. This is not true, due to problems with extensionality: Proving propositional equality between functions rarely works in intensional type theory such as CIC, and the use of $Ran_H\, G$ in the construction of Lemma 1 introduces values of function type.

As an instance of this theorem, one can prove the fourth item in the list on page 230 on properties of *substE*. The fifth item (the interchange law for substitution that is one of the monad laws) can then be proven by the induction principle $\mu FInd$, using extensionality and both parts of naturality (hence, the items 1, 3 and 4 that are based on Lemma 3, Lemma 5 and Theorem 1) in the case for the representation of lambda abstraction (recall that *liftE* is defined by help of *lamE*).

5 Completion of the Case Study on Substitution

The last item on page 230 in the list of properties of *substE* can be proven by the induction principle $\mu FInd$ without any results about *LamE*, just with several preparations about lists, also using naturality of *EFV* in the proof of the case for the representation of lambda abstraction. Thus, that property list can be considered as finished.

We are not yet fully satisfied: The last monad law is missing, namely

$$\forall A \forall t^{LamE\, A}.\, substE\, varE_A\; t = t.$$

Any proof attempt breaks due to the presence of non-canonical terms in *LNGMIt*. We call any term of the form $InCan\, t$ with $t : F(\mu F)A$ a canonical term in $\mu F A$, but since this notion is not recursively applied to the subterms, we cannot hope to prove the above monad law for all the canonical terms in the family *LamE* either.

The following is an ad hoc notion for our example. For the truly nested datatype *Bush* of "bushes" with $Bush\, A = 1 + A \times Bush(Bush\, A)$, a similar notion has been studied by the author in [14, Section 4.2], also introducing a "canonization" function that transforms any bush into a hereditarily canonical bush and that does not change hereditarily canonical bushes with respect to propositional equality.

Definition 3 (Hereditarily canonical term). *Define the notion of hereditarily canonical elements of the nested datatype LamE, the predicate can :* $\forall A.\, LamE\, A \to Prop$*, inductively by the following four closure rules:*

- $\forall A \forall a^A.\, can\, (varE\, a)$
- $\forall A \forall t_1^{LamE\, A} \forall t_2^{LamE\, A}.\, can\, t_1 \to can\, t_2 \to can(appE\, t_1\, t_2)$
- $\forall A \forall r^{LamE(option\, A)}.\, can\, r \to can\, (absE\, r)$
- $\forall A \forall e^{LamE(LamE\, A)}.\, can\, e \to (\forall t^{LamE\, A}.\, t \in EFV e \to can\, t) \to can\, (flatE\, e)$

This definition is strictly positive and, formally, infinitely branching. However, there are always only finitely many t that satisfy $t \in EFV e$. System pCIC does not need this latter information for having induction principles for *can*, and *LNGMIt* comprises pCIC, but this is not the part that is under study here. Therefore, all proofs by induction on *can* are not considered to be of real interest for this article. Except for the information which results are used in these proofs.

Note once again the simultaneous inductive-recursive structure that is avoided here: If only hereditarily canonical elements were to be considered from the beginning, one would have to define their free variables simultaneously since the last clause of the definition refers to them at a negative position.

5.1 Results for Hereditarily Canonical Terms

Using refined extensionality of *substE* (property number 2 in the list on page 230) in the induction step for *flatE* e, induction on *can* provides the relativization of the missing monad law to hereditarily canonical terms:

$$\forall A \forall t^{LamE\,A}.\, can\, t \to substE\; varE_A\; t = t.$$

Renaming *lamE* preserves hereditary canonicity:

$$\forall A \forall B \forall f^{A \to B} \forall t^{LamE\,A}.\, can\, t \to can(lamE\, f\, t).$$

This is proven by induction on *can*, and the crucial *flatE* case needs the following identification of free variables of *lamE* $f\,t$:

$$\forall A \forall B \forall f^{A \to B} \forall t^{LamE\,A} \forall b^B.\, b \in EFV(lamE\, f\, t) \to \exists a^A.\, a \in EFV\, t \wedge b = fa,$$

which is nearly an immediate consequence of naturality of *EFV*.

Analogously, *substE* preserves hereditary canonicity:

$$\forall A \forall B \forall f^{A \to LamE\,B} \forall t^{LamE\,A}.$$
$$(\forall a^A.\, a \in EFV\, t \to can(fa)) \to can\, t \to can(substE\, f\, t).$$

Again, this is proven by induction on *can*, and again, the crucial case is with *flatE* e, for which free variables of *substE* $f\,t$ have to be identified, but this has already been mentioned as a consequence of property number 6 in the list on page 230.

As an immediate consequence of the last monad law, preservation of hereditary canonicity by *lamE* and the second part of naturality of *substE* (item 4 of the list, proven by map fusion), one can see *lamE* as a special instance of *substE* for hereditarily canonical elements:

$$\forall A \forall B \forall f^{A \to B} \forall t^{LamE\,A}.\, can\, t \to lamE\, f\, t = substE\,(varE_B \circ f)\, t.$$

From this, evidently, we get the more perspicuous equation for *substE* f (*flatE* e), discussed on page 229, but only for hereditarily canonical e and only with propositional equality:

$$\forall A \forall B \forall f^{A \to LamE\,B} \forall e^{LamE(LamE\,A)}.\, can\, e \to$$
$$substE\, f\,(flatE\, e) = flatE(lamE\,(substE\, f)\, e).$$

5.2 Hereditarily Canonical Terms as a Nested Datatype

Define $LamEC := \lambda A.\, \{t : LamE\, A \mid can\, t\} : \kappa_1$. The set comprehension notation stands for the inductively defined `sig` of Coq (definable within pCIC, hence within *LNGMIt*) which is a strong sum in the sense that the first projection $\pi_1 : \mathit{LamEC} \subseteq \mathit{LamE}$ yields the element t and the second projection the proof of $\mathit{can}\, t$.

Thus, we encapsulate hereditary canonicity already in the family *LamEC*. We will present *LamEC* as a truly nested datatype, but not one that comes as a μF from *LNGMIt*.

It is quite trivial to define datatype constructors

$$\begin{aligned} &\mathit{varEC} \;: \forall A.\, A \to \mathit{LamEC}\, A,\\ &\mathit{appEC} : \forall A.\, \mathit{LamEC}\, A \to \mathit{LamEC}\, A \to \mathit{LamEC}\, A,\\ &\mathit{absEC} \;: \forall A.\, \mathit{LamEC}(\mathit{option}\, A) \to \mathit{LamEC}\, A \end{aligned}$$

from their analogues in *LamE*. For the construction of

$$\mathit{flatEC} : \forall A.\, \mathit{LamEC}(\mathit{LamEC}\, A) \to \mathit{LamEC}\, A,$$

the problem is as follows: Assume $e : \mathit{LamEC}(\mathit{LamEC}\, A)$. Then, its first projection, $\pi_1 e$, is of type $\mathit{LamE}(\mathit{LamEC}\, A)$. Therefore, the first projection of $\mathit{flatEC}\, e$ has to be

$$t := \mathit{flatE}(\mathit{lamE}\ (\pi_1)_A\ (\pi_1 e)) : \mathit{LamE}\, A,$$

with the renaming with $(\pi_1)_A : \mathit{LamEC}\, A \to \mathit{LamE}\, A$ inside. Thanks to the preservation of hereditary canonicity by *lamE* and the identification of the variables of renamed terms, canonicity of t can be established.

Since *flatEC* is doing something with its argument, we cannot think of *LamEC* as being generated from the four datatype constructors. We see this more as a semantical construction whose properties can be studied. However, there is still the operational kernel available in the form of the definitional equality $\simeq$.

From preservation of hereditary canonicity by *lamE* and *substE*, one can easily define $\mathit{lamEC} : \mathit{mon}\, \mathit{LamEC}$ and

$$\mathit{substEC} : \forall A \forall B.\, (A \to \mathit{LamEC}\, B) \to \mathit{LamEC}\, A \to \mathit{LamEC}\, B.$$

The list of free variables is obtained through $\mathit{ECFV} : \mathit{LamEC} \subseteq \mathit{List}$, defined by composing *EFV* with π_1, which is then also natural. Therefore, one can immediately transfer the identification of free variables of $\mathit{lamE}\, f\, t$ and $\mathit{substE}\, f\, t$ to *lamEC* and *substEC*.

In order to have “real” results, proof irrelevance has to be assumed for the proofs of hereditary canonicity. From propositional proof irrelevance, as used in Corollary 1, it immediately follows that π_1 is injective:

$$\forall A \forall t_1, t_2 : \mathit{LamEC}\, A.\, \pi_1\, t_1 = \pi_1\, t_2 \to t_1 = t_2\ \ .$$

This is the only addition to *LNGMIt* that we adopt here. Then, all the properties of the list in Section 2.2 can be transferred to *substEC*, the recursive description (now only with propositional equality) of *lamE* can be carried over to *lamEC* that makes *LamEC* an extensional functor, and also the results of Section 5.1 that were relativized to hereditarily canonical terms now hold unconditionally for *lamEC* and *substEC*. Finally, a monad structure has been obtained. Once again, all the proofs are to be found in the Coq scripts [17].

6 Conclusions and Future Work

Recursive programming with Mendler-style iteration is able to cover intricate nested datatypes with functions whose termination is far from being obvious. But termination is not the only property of interest. A calculational style of verification that is based on generic results such as naturality criteria is needed on top of static analysis. The system *LNGMIt* and the earlier system *LNMIt* from which it is derived are an attempt to combine the benefits from both paradigms: the rich dependently-typed language secured by decidable type-checking and termination guarantees on one side and the laws that are inspired from category theory on the other side.

LNGMIt can prove naturality in many cases, with a notion of naturality that encompasses map fusion. However, the system is heavily based on the unintuitive non-canonical datatype constructor *In* which makes reasoning on paper somewhat laborious. This can be remedied by intensive use of computer aided proof development. The ambient system for the development of the metatheory and the case study is the Calculus of Inductive Constructions that is implemented by the Coq system. Proving and programming can both be done interactively. Therefore, *LNGMIt*, through its implementation in Coq, can effectively aid in the construction of terminating programs on nested datatypes and to establish their equational properties.

Certainly, the other laws in, e. g., [15] should be made available in our setting as well. Clearly, not only (generalized) iteration should be available for programs on nested datatypes. The author experiments with primitive recursion in Mendler style, but does not yet have termination guarantees [31].

An alternative to *LNGMIt* with its non-canonical elements could be a dependently-typed approach from the very beginning. This could be done by indexing the nested datatypes additionally over the natural numbers as with sized nested datatypes [27] where the size corresponds to the number of iterations of the datatype “functor” over the constantly empty family. But one could also try to define functions directly for all powers of the nested datatype (suggested to me by Nils Anders Danielsson) or even define all powers of it simultaneously (suggested to me by Conor McBride). The author has presented preliminary results at the TYPES 2004 meeting about yet another approach where the indices are finite trees that branch according to the different arguments that appear in the recursive equation for the nested datatype (based on ideas by Anton Setzer and Peter Aczel).

References

1. Bird, R., Meertens, L.: Nested Datatypes. In: Jeuring, J. (ed.) MPC 1998. LNCS, vol. 1422, pp. 52–67. Springer, Heidelberg (1998)
2. Bird, R., Gibbons, J., Jones, G.: Program optimisation, naturally. In: Davies, J., Roscoe, B., Woodcock, J. (eds.) Millenial Perspectives in Computer Science, Proceedings of the 1999 Oxford-Microsoft Symp. in Honour of Professor Sir Anthony Hoare, Palgrave (2000)

3. Hinze, R.: Efficient generalized folds. In: Jeuring, J. (ed.) Proceedings of the Second Workshop on Generic Programming, WGP 2000, Ponte de Lima, Portugal (2000)
4. Bellegarde, F., Hook, J.: Substitution: A formal methods case study using monads and transformations. Science of Computer Programming 23, 287–311 (1994)
5. Bird, R.S., Paterson, R.: De Bruijn notation as a nested datatype. Journal of Functional Programming 9(1), 77–91 (1999)
6. Altenkirch, T., Reus, B.: Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In: Flum, J., Rodríguez-Artalejo, M. (eds.) CSL 1999. LNCS, vol. 1683, pp. 453–468. Springer, Heidelberg (1999)
7. Coq Development Team: The Coq Proof Assistant Reference Manual Version 8.1. Project LogiCal, INRIA (2006), `http://coq.inria.fr`
8. Paulin-Mohring, C.: Définitions Inductives en Théorie des Types d'Ordre Supérieur. Habilitation à diriger les recherches, Université Claude Bernard Lyon I (1996)
9. Letouzey, P.: A New Extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003)
10. Abel, A., Matthes, R., Uustalu, T.: Iteration and coiteration schemes for higher-order and nested datatypes. Theoretical Computer Science 333(1–2), 3–66 (2005)
11. Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects of Computing 11(2), 200–222 (1999)
12. Abel, A., Matthes, R. (Co-)Iteration for Higher-Order Nested Datatypes. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 1–20. Springer, Heidelberg (2003)
13. Bird, R., de Moor, O.: Algebra of Programming. International Series in Computer Science, vol. 100. Prentice-Hall, Englewood Cliffs (1997)
14. Matthes, R.: An induction principle for nested datatypes in intensional type theory. Journal of Functional Programming (to appear, 2008)
15. Martin, C., Gibbons, J., Bayley, I.: Disciplined, efficient, generalised folds for nested datatypes. Formal Aspects of Computing 16(1), 19–35 (2004)
16. Johann, P., Ghani, N.: Initial Algebra Semantics is enough! In: Ronchi Della Rocca, S. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 207–222. Springer, Heidelberg (2007)
17. Matthes, R.: Coq development for Nested datatypes with generalized Mendler iteration: map fusion and the example of the representation of untyped lambda calculus with explicit flattening (January 2008), `http://www.irit.fr/~Ralph.Matthes/Coq/MapFusion/`
18. Matthes, R.: Coq development for An induction principle for nested datatypes in intensional type theory (January 2008), `http://www.irit.fr/~Ralph.Matthes/Coq/InductionNested/`
19. Mendler, N.P.: Recursive types and type constraints in second-order lambda calculus. In: Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, Ithaca, pp. 30–36. IEEE Computer Society Press (1987)
20. Barthe, G., Frade, M.J., Giménez, E., Pinto, L., Uustalu, T.: Type-based termination of recursive definitions. Mathematical Structures in Computer Science 14, 97–141 (2004)
21. Uustalu, T., Vene, V.: A cube of proof systems for the intuitionistic predicate μ-, ν-logic. In: Haveraaen, M., Owe, O. (eds.) Selected Papers of the 8th Nordic Workshop on Programming Theory (NWPT 1996). Research Reports, Department of Informatics, University of Oslo, vol. 248, pp. 237–246 (May 1997)

22. Matthes, R.: Naive reduktionsfreie Normalisierung (translated to English: naive reduction-free normalization). Slides of talk on December 19, 1996, given at the Bern Munich meeting on proof theory and computer science in Munich, available at the author's homepage (December 1996)
23. Hofmann, M.: Extensional concepts in intensional type theory. PhD thesis, University of Edinburgh, Available as report ECS-LFCS-95-327 (1995)
24. Altenkirch, T.: Extensional equality in intensional type theory. In: 14th Annual IEEE Symposium on Logic in Computer Science (LICS 1999), pp. 412–420. IEEE Computer Society, Los Alamitos (1999)
25. Oury, N.: Extensionality in the Calculus of Constructions. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 278–293. Springer, Heidelberg (2005)
26. Wadler, P.: Theorems for free? In: Proceedings of the fourth international conference on functional programming languages and computer architecture, Imperial College, pp. 347–359. ACM Press, London (1989)
27. Abel, A.: A Polymorphic Lambda-Calculus with Sized Higher-Order Types. Doktorarbeit (PhD thesis), LMU München (2006)
28. Dybjer, P.: A general formulation of simultaneous inductive-recursive definitions in type theory. The Journal of Symbolic Logic 65(2), 525–549 (2000)
29. Capretta, V.: A polymorphic representation of induction-recursion. Note of 9 pages available on the author's web page (a second 15 pages version of May 2005 has been seen by the present author) (March 2004)
30. Mac Lane, S.: Categories for the Working Mathematician, 2nd edn. Graduate Texts in Mathematics, vol. 5. Springer, Heidelberg (1998)
31. Matthes, R.: Recursion on nested datatypes in dependent type theory. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) Logic and Theory of Algorithms. LNCS, vol. 5028. Springer, Heidelberg (to appear, 2008)

Probabilistic Choice in Refinement Algebra

Larissa Meinicke[1] and Ian J. Hayes[2,⋆]

[1] Department of Computer Science,
Åbo Akademi, Finland
[2] School of Information Technology and Electrical Engineering,
The University of Queensland, Australia
larissa.meinicke@abo.fi, ianh@itee.uq.edu.au

Abstract. The term refinement algebra refers to a set of abstract algebras, similar to Kleene algebra with tests, that are suitable for reasoning about programs in a total-correctness framework. Abstract algebraic reasoning also works well when probabilistic programs are concerned, and a general refinement algebra that is suitable for such programs has been defined previously. That refinement algebra *does not* contain features that are specific to probabilistic programs. For instance, it does not include a probabilistic choice operator, or *probabilistic* assertions and guards (tests), which may be used to represent correctness properties for probabilistic programs. In this paper we investigate how these features may be included in a refinement algebra. That is, we propose a new refinement algebra in which probabilistic choice, and probabilistic guards and assertions may be expressed. Two operators for modelling probabilistic enabledness and termination are also introduced.

1 Introduction

Abstract algebras, for example Kleene algebra with tests [5], have been shown to be useful tools for reasoning about programs. They provide us with a convenient way to describe similarities and dissimilarities between different program models, and to verify transformation theorems in a model-independent way, which is potentially simple to automate.

The term refinement algebra refers to a set of abstract algebras that are suitable for reasoning about programs in a total-correctness framework. These algebras are similar to Kleene algebra (the algebra of regular languages), and its variations, which may be used for reasoning about the partial-correctness of programs [5].

Perhaps the best known refinement algebra is the demonic refinement algebra of von Wright [15,16]. The demonic refinement algebra is equipped with operators which may be used to represent sequential composition, demonic nondeterministic choice, and two kinds of iteration: weak iteration, which iterates its

[⋆] This research was supported by Australian Research Council (ARC) Discovery Grant DP0558408, *Analysing and generating fault-tolerant real-time systems.* We would like to thank Kim Solin for feedback on earlier drafts of this paper, and the anonymous reviewers for their thoughtful and helpful comments.

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 243–267, 2008.

argument any finite number of times, and a strong iteration, which may iterate any finite or possibly infinite number of times. Like Kleene algebra, guards (or tests) may be defined in this refinement algebra, as can assertions. These elements represent predicates at the level of programs. Among other things, they may be used to define conditional statements, and to reason algebraically about correctness properties. For example, if $[g]$ represents a guard, where g is a predicate with complement $\neg g$, then a conditional statement can be defined in terms of guards and nondeterministic choice ($\sqcap$) by

$$\mathsf{if}\ g\ \mathsf{then}\ S\ \mathsf{else}\ T\ \mathsf{fi} \triangleq [g]; S \sqcap [\neg g]; T. \tag{1}$$

Also, the equivalence

$$[\mathsf{False}] = [p]; S; [\neg g]$$

states that from an initial state in which predicate p holds, program S must terminate and establish post-condition g [15]. Enabledness (domain) and termination operators have also been defined and explored in extensions to the algebra [14].

The demonic refinement algebra is sound with respect to program models like the conjunctive predicate transformers, in which only one choice operator is expressible. However, it is not sound with respect to program models that also allow for another form of choice, such as angelic, or discrete probabilistic choice: for these models a weaker axiomatisation is required. In [16] von Wright proposed a relaxation of demonic refinement algebra, the general refinement algebra, for which the monotonic predicate transformers form a model. In further work, Meinicke and Solin explored (and extended) this algebra and showed that it was also sound with respect to two probabilistic program models: one in which discrete probabilistic choice and demonic nondeterministic choice coexist, and another in which discrete probabilistic, demonic and angelic nondeterministic choice can be expressed [9]. As well as being a useful way to identify commonalities between these different program models, the general refinement algebra was shown to be capable of deriving a number of useful transformation theorems for probabilistic – and angelic – programs.

Although it is possible to derive a number of useful transformation theorems for probabilistic programs in the general refinement algebra, it doesn't facilitate reasoning about some properties which are specific to probabilistic programs. For example, it contains neither a probabilistic choice operator, nor generalised (*probabilistic*) guards and assertions, which may be used to represent correctness properties for probabilistic programs. It is the purpose of this paper to propose and investigate extensions to the general refinement algebra in which discrete probabilistic choice, and probabilistic guards and assertions are treated abstractly.

We take a novel approach in the definition of our new algebras. Instead of introducing a probabilistic choice operator in the signature, we introduce a more general *plus* operator ("+"), which we then use to define probabilistic choice. Our algebra is equipped with both non-probabilistic and probabilistic guards and assertions, as well as probabilistic enabledness and termination operators. Probabilistic assertions are used in conjunction with the plus operator to define probabilistic choice, in much the same way that guards are used in conjunction with nondeterministic choice to define conditional statements in the existing refinement algebras (see (1)). In the standard refinement calculus an assertion, $\{p\}$, is defined in terms of a predicate p over the state space. Probabilistic assertions generalise this so that p is a mapping from the state space to the closed real interval $[0..1]$. Probabilistic choice is then defined using the plus operator as

$$S\ {}_p{\oplus}\ T \triangleq \{p\}; S + \{1-p\}; T.$$

This provides us with a simple notation for expressing multi-way probabilistic choices. For example, a three-way probabilistic choice may be represented symmetrically by $\{\frac{1}{3}\}; S + \{\frac{1}{3}\}; T + \{\frac{1}{3}\}; U$.

In Sect. 2, we outline the probabilistic program model we use to motivate our refinement algebra extensions. Then we specify the general refinement algebra (Sect. 3) and propose our new extensions for a probabilistic refinement algebra (Sect. 4). In Sect. 5 we introduce both probabilistic and non-probabilistic assertions and guards into the new algebras and we show how probabilistic choice may be defined in terms of the new operator, "+", and probabilistic assertions. Algebraic properties of the probabilistic choice operator and probabilistic loops are then described in Sect. 6, and in Sect. 7 we show how probabilistic guards and assertions can be used to express correctness conditions. Sect. 8 then discusses the probabilistic enabledness and termination operators, and some observations about the algebras are made in Sect. 9.

2 Probabilistic Model

We use *one-bounded expectation transformers* [10,11] to represent probabilistic programs using a *weakest expectation* semantics. Expectation transformers [6,10,12] may be seen as a generalisation of *predicate transformers* [2,3] which may be used to describe the semantics of non-probabilistic programs. We use them here in favour of a relational model because, like predicate transformers, they are more expressive, elegant and simple to reason with.

First we briefly define the one-bounded expectations and expectation transformers, and then the semantics of the commands are defined. Last we describe the healthiness conditions used to classify interesting subsets of the one-bounded expectation transformers.

2.1 Expectations and Expectation Transformers

The set of *expectations* on a state space Σ, $\mathcal{E}\Sigma \triangleq \Sigma \rightarrow \mathbb{R}_{\geq 0}$, are a generalisation of the set of *predicates* on Σ, $\mathcal{P}\Sigma \triangleq \Sigma \rightarrow \{0,1\}$; and the one-bounded expectations, $\mathcal{E}_1\Sigma \triangleq \Sigma \rightarrow [0..1]$, are the set of expectations on Σ which are bounded above by one. The operator used to order expectations, $\leq$, along with some other basic operators we use throughout the paper are defined in Fig. 1.

Let ϕ and ψ be of type $\mathcal{E}\Sigma$; ϕ_1 be of type $\mathcal{E}_1\Sigma$; p and q be of type $\mathcal{P}\Sigma$; c be a constant of type $\mathbb{R}_{\geq 0}$; and r be a probability in [0..1]. When applied to real numbers, $\sqcap$ is the minimum operator (meet), $\sqcup$ is the maximum operator (join), and $\times$ denotes multiplication. For a probability $r \in [0..1]$, we often write $\neg r$ to mean $1 - r$.

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
\phi \leq \psi & \triangleq & (\forall \sigma \in \Sigma \bullet \phi.\sigma \leq \psi.\sigma) & p \Rightarrow q & \triangleq & p \leq q \\
\phi \sqcap \psi & \triangleq & (\lambda \sigma \in \Sigma \bullet \phi.\sigma \sqcap \psi.\sigma) & p \wedge q & \triangleq & p \sqcap q \\
\phi \sqcup \psi & \triangleq & (\lambda \sigma \in \Sigma \bullet \phi.\sigma \sqcup \psi.\sigma) & p \vee q & \triangleq & p \sqcup q \\
\phi \times \psi & \triangleq & (\lambda \sigma \in \Sigma \bullet \phi.\sigma \times \psi.\sigma) & \mathsf{True} & \triangleq & (\lambda \sigma \in \Sigma \bullet 1) \\
\phi + \psi & \triangleq & (\lambda \sigma \in \Sigma \bullet \phi.\sigma + \psi.\sigma) & \mathsf{False} & \triangleq & (\lambda \sigma \in \Sigma \bullet 0) \\
\neg \phi_1 & \triangleq & (\lambda \sigma \in \Sigma \bullet 1 - \phi_1.\sigma) & & & \\
c * \phi & \triangleq & (\lambda \sigma \in \Sigma \bullet c \times \phi.\sigma) & & & \\
\phi \ominus c & \triangleq & (\lambda \sigma \in \Sigma \bullet (\phi.\sigma - c) \sqcup 0) & & & \\
\phi \ {}_r\!\oplus\ \psi & \triangleq & r * \phi + (1 - r) * \psi & & &
\end{array}
$$

Fig. 1. Expectation notation

Given a fixed state space Σ, a *one-bounded expectation transformer* on Σ is simply a function from the set of one-bounded expectations on Σ to the set of one-bounded expectations on Σ. Like predicate transformers, expectation transformers are ordered with respect to the underlying order on expectations. That is, for $S, T : \mathcal{E}_1\Sigma \rightarrow \mathcal{E}_1\Sigma$, $S \sqsubseteq T \triangleq (\forall \phi \in \mathcal{E}_1\Sigma \bullet S.\phi \leq T.\phi)$.

2.2 Commands

In Fig. 2 we present the *weakest-expectation* semantics [6,12] of some basic operators and commands. Informally, the *weakest expectation* of a command S to achieve a post-expectation ϕ from an initial state σ, $S.\phi.\sigma$, represents the least mean value of ϕ (a *random variable*) that may be observed by executing S (an *experiment*) from σ.

Let $\phi, \psi \in \mathcal{E}_1\Sigma$; $S, T : \mathcal{E}_1\Sigma \rightarrow \mathcal{E}_1\Sigma$; x be a variable in state space Σ and E be an expression on Σ.

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
\mathsf{abort}.\phi & \triangleq & \mathsf{False} & (S \sqcap T).\phi & \triangleq & S.\phi \sqcap T.\phi \\
\mathsf{magic}.\phi & \triangleq & \mathsf{True} & (S \sqcup T).\phi & \triangleq & S.\phi \sqcup T.\phi \\
\mathsf{skip}.\phi & \triangleq & \phi & (S; T).\phi & \triangleq & S.(T.\phi) \\
\{\psi\}.\phi & \triangleq & \psi \times \phi & S^{*}.\phi & \triangleq & (\nu X \bullet S; X \sqcap \mathsf{skip}).\phi \\
[\psi].\phi & \triangleq & (\neg\psi) \times \mathsf{True} + \psi \times \phi & S^{\omega}.\phi & \triangleq & (\mu X \bullet S; X \sqcap \mathsf{skip}).\phi \\
(x := E).\phi & \triangleq & (\lambda \sigma \in \Sigma \bullet \phi.(\sigma[x \backslash E.\sigma])) & S^{\circledast}.\phi & \triangleq & (\mu X \bullet S; X + \mathsf{skip}).\phi \\
(S + T).\phi & \triangleq & (S.\phi + T.\phi) \sqcap \mathsf{True} & S^{\ddagger}.\phi & \triangleq & (\nu X \bullet S; X + \mathsf{skip}).\phi \\
(S \ {}_{\psi}\!\oplus\ T).\phi & \triangleq & (\{\psi\}; S + \{\neg\psi\}; T).\phi & & &
\end{array}
$$

Fig. 2. Weakest expectation semantics of commands

The least mean value of any ϕ that may be observed by executing the least program abort from some initial state σ is 0. Intuitively, this is because the *experiment* abort may fail to terminate in any state, and so the least mean value of the *random variable* ϕ – recorded after a large number of trials of abort – is 0. The greatest program magic is not implementable[1], and therefore has no intuitive program interpretation: the least mean value of any ϕ which may be observed by executing magic from σ is, miraculously, one. The program skip, which does not modify the state in any way, simply produces a pre-expectation which is the same as any given post-expectation.

Standard guards and assertions are generalised to *probabilistic* guards and assertions in the model. Given a one-bounded expectation ϕ, probabilistic assertion $\{\phi\}$ skips with probability $\phi.\sigma$ from some initial state σ, and aborts with probability $\neg(\phi.\sigma)$. Similarly, probabilistic guard $[\phi]$ skips with probability $\phi.\sigma$ from σ and performs magic with probability $\neg(\phi.\sigma)$. These commands are novel. Together with a new binary operator, $+$, probabilistic assertions may be used to define the discrete probabilistic choice operator $\oplus$. In our probabilistic algebras we use this observation to define probabilistic choice abstract-algebraically without explicitly referring to the real numbers, or real-valued functions. Real-valued functions (where the real-values are bounded between zero and one) are embedded in our carrier set as probabilistic assertions, in the same way that predicates are embedded as assertions in the existing refinement algebras. These probabilistic assertions are then used in conjunction with the plus operator, which is included in the signature of the algebra, to represent probabilistic choice.

In addition to discrete probabilistic choice, demonic nondeterministic choice, $\sqcap$, and angelic nondeterministic choice, $\sqcup$, are defined, as are sequential composition ";", and assignment.

The iterative constructs *, $^\omega$, $^\circledast$ and ‡ are defined using greatest (ν) and least (μ) fixpoints. They are discussed in detail later in the paper. By the Knaster-Tarski Theorem, these statements are well defined since the one-bounded expectation transformers form a complete lattice with respect to their refinement ordering, $\sqsubseteq$.

The iteration operators are assigned highest precedence, followed by sequential composition, ";", and then with equal precedence plus, $+$, demonic, $\sqcap$, angelic, $\sqcup$, and probabilistic choice, $\oplus$. Ambiguity between the choice operators is resolved by bracketing.

2.3 Healthiness Conditions

Like predicate transformers, the set of one-bounded expectation transformers is very large and expressive (allowing for the expression of programs which may have no obvious real-world interpretation), and so we find it necessary to define useful subsets of the transformers using certain *healthiness conditions* [6].

In our earlier work (e.g., [7,9]) we were concerned with two main classes of the one-bounded expectation transformers: the *nondeterministic* and the

[1] Although it is useful for reasoning *about* implementable programs!

Let $S : \mathcal{E}_1\Sigma \to \mathcal{E}_1\Sigma$; $\beta_1, \beta_2 \in \mathcal{E}_1\Sigma$; p be a probability in $[0..1]$; c and c' be constants such that $0 \leq c, c'$ and $c - c' \leq 1$; and $\mathcal{B}$ and $\mathcal{B}'$ be non-empty directed and codirected sets respectively, of expectations of type $\mathcal{E}_1\Sigma$. A set Y is directed with respect to an ordering $\leq$, iff $(\forall y_1, y_2 \in Y \bullet (\exists y_3 \in Y \bullet y_1 \leq y_3 \wedge y_2 \leq y_3)$; and it is codirected iff $(\forall y_1, y_2 \in Y \bullet (\exists y_3 \in Y \bullet y_3 \leq y_1 \wedge y_3 \leq y_2)$.

$$\beta_1 \leq \beta_2 \Rightarrow S.\beta_1 \leq S.\beta_2 \qquad \text{(monotonicity)}$$

$$c * S.\beta_1 \ominus c' \leq S.(c * \beta_1 \ominus c') \qquad \text{(semi-sublinearity)}$$

$$c * S.\beta_1 \leq S.(c * \beta_1) \qquad \text{(sub-scaling)}$$

$$S.\beta_1 \ominus c' \leq S.(\beta_1 \ominus c') \qquad \text{(subtraction)}$$

$$S.\beta_1 \;{}_p\!\oplus\; S.\beta_2 = S.(\beta_1 \;{}_p\!\oplus\; \beta_2) \qquad \text{($\oplus$-distributivity)}$$

$$S.\beta_1 \;{}_p\!\oplus\; S.\beta_2 \leq S.(\beta_1 \;{}_p\!\oplus\; \beta_2) \qquad \text{($\oplus$-subdistributivity)}$$

$$S.\beta_1 \sqcap S.\beta_2 = S.(\beta_1 \sqcap \beta_2) \qquad \text{(conjunctivity)}$$

$$(S.(\sqcup\beta \in \mathcal{B} \bullet \beta) = (\sqcup\beta \in \mathcal{B} \bullet S.\beta)) \qquad \text{(semi-continuity)}$$

$$(S.(\sqcap\beta \in \mathcal{B}' \bullet \beta) = (\sqcap\beta \in \mathcal{B}' \bullet S.\beta)) \qquad \text{(semi-cocontinuity)}$$

Fig. 3. Healthiness conditions for one-bounded expectation transformers

dually nondeterministic expectation transformers. These sets are the probabilistic analog of the (finitely) *conjunctive* and the *monotonic* predicate transformers, respectively.

The *nondeterministic* one-bounded expectation transformers are those which satisfy *monotonicity*, *semi-sublinearity* and $\oplus$-*subdistributivity* (Fig. 3) [10]. For finite state spaces, the nondeterministic one-bounded expectation transformers uniquely characterise a relational model for probabilistic programs, the *Lamington relational model* [10]: a model which is capable of expressing programs that may include discrete probabilistic choices and demonic nondeterministic choices. The set of nondeterministic transformers contains the primitive commands in Fig. 2 and is closed under the operators ";", $\oplus$, $\sqcap$, * and $^\omega$.

The *dually nondeterministic* one-bounded expectation transformers are those which satisfy monotonicity and semi-sublinearity alone [7]. This set is capable of representing probabilistic programs which may include both angelic *and* demonic choices[2], in addition to discrete probabilistic choice. The set is closed under all the commands in Fig. 2, other than $+$, and the two iteration operators that are defined using plus: $^{\circledast}$ and ‡.

Neither the nondeterministic or dually nondeterministic transformers are closed under the more exotic plus operator – although they *are* closed under probabilistic choice – and so cannot be taken as carrier sets of an algebra for which the plus operator is defined. Take for example nondeterministic expectation transformer skip. We have that skip + skip does not satisfy semi-sublinearity:

[2] Which is why it is referred to as *dually* nondeterministic.

$$\begin{aligned}
&(\mathsf{skip}+\mathsf{skip}).(1 * (\lambda\sigma \bullet \tfrac{1}{2}) \ominus \tfrac{1}{2})\\
=\ &\{\text{expectation definitions}\}\\
&(\mathsf{skip}+\mathsf{skip}).\mathsf{False}\\
=\ &\{\text{definition of plus}\}\\
&\mathsf{skip}.\mathsf{False}+\mathsf{skip}.\mathsf{False}\\
=\ &\{\text{definition of } \mathsf{skip}\}\\
&\mathsf{False}\\
\ngeq\ &(\lambda\sigma \bullet \tfrac{1}{2})\\
=\ &\{\text{definition of } \mathsf{skip} \text{ and expectations}\}\\
&(\mathsf{skip}.(\lambda\sigma \bullet \tfrac{1}{2}) + \mathsf{skip}.(\lambda\sigma \bullet \tfrac{1}{2})) \ominus \tfrac{1}{2}\\
=\ &\{\text{definition of plus}\}\\
&1 * (\mathsf{skip}+\mathsf{skip}).(\lambda\sigma \bullet \tfrac{1}{2}) \ominus \tfrac{1}{2}.
\end{aligned}$$

For this reason, we are mainly interested in the two slightly more general sets of one-bounded expectation transformers that *are* closed under plus, and the iteration constructs which contain this operator, $^{\circledast}$ and ‡. These are the *nondeterministic*$_{+}$ and the *dually nondeterministic*$_{+}$ one-bounded expectation transformers, respectively. The nondeterministic$_{+}$ transformers satisfy monotonicity, $\oplus$-subdistributivity, and sub-scaling (which is actually implied by $\oplus$-subdistributivity), while the dually nondeterministic$_{+}$ transformers satisfy monotonicity and sub-scaling alone. We can see that the part of semi-sublinearity that is lost is a property called *subtraction*.

Healthiness Conditions, Algebraically. The healthiness conditions used to describe expectation transformers are usually expressed at the level of expectations (as they appear in Fig. 3), however, we observe that they may be *equivalently* expressed in a *point-free* form, as algebraic properties.[3] For example, the following proposition holds.

Proposition 1. A one-bounded expectation transformer S is $\oplus$-distributive *if and only if* for all constant one-bounded expectations c, and one-bounded transformers R and T,

$$S;R \;{}_{c}\!\oplus\; S;T = S;(R \;{}_{c}\!\oplus\; T). \tag{2}$$

Note that c is required to be a constant expectation since S may modify the state.

Aside: In order to prove that (2) holds for all one-bounded expectations R and T, it is sufficient to verify that it holds for all nondeterministic one-bounded expectation transformers R and T.

Proof. It is simple to verify that the healthiness condition implies the point-free property (McIver and Morgan make this observation in [6]), so we verify implication in the other direction.

[3] In [13] Solin showed that monotonicity and conjunctivity may be characterised in a point-free way for the predicate transformers.

For any one-bounded expectation transformer S on state space Σ, the following give the (algebraic) conditions for S to satisfy the corresponding healthiness properties. The conditions must hold for all constant one-bounded expectations, c; one-bounded expectation transformers, R and T; and non-empty directed, and co-directed sets of one-bounded expectation transformers, X and X'.
Aside: In order to prove that the first five properties hold for all R and T it is sufficient to verify that they hold for all nondeterministic one-bounded transformers R, T.

$$S; (R \sqcap T) \sqsubseteq S; R \sqcap S; T \qquad \text{(monotonicity)}$$
$$\{c\}; S \sqsubseteq S; \{c\} \qquad \text{(sub-scaling)}$$
$$S; R \,{}_c\!\oplus\; S; T = S; (R \,{}_c\!\oplus\; T) \qquad \text{($\oplus$-distributivity)}$$
$$S; R \,{}_c\!\oplus\; S; T \sqsubseteq S; (R \,{}_c\!\oplus\; T) \qquad \text{($\oplus$-subdistributivity)}$$
$$S; R \sqcap S; T = S; (R \sqcap T) \qquad \text{(conjunctivity)}$$
$$S; (\sqcup T_i \in X \bullet T_i) = (\sqcup T_i \in X \bullet S; T_i) \qquad \text{(semi-continuity)}$$
$$S; (\sqcap T_i \in X' \bullet T_i) = (\sqcap T_i \in X' \bullet S; T_i) \qquad \text{(semi-cocontinuity)}$$

Fig. 4. Equivalent algebraic formulation of healthiness conditions

Take any expectation transformer S, and constant probability function P which has value p everywhere, and expectations ϕ and ψ. Let R and T be one-bounded expectation transformers such that $R.\theta = \phi$ and $T.\theta = \psi$, for some expectation θ. We have that if S satisfies (2), then

$$\begin{aligned}
& S.(\phi \,{}_p\!\oplus\; \psi) \\
= & \{\text{definition of } R,\ T \text{ and } \theta\} \\
& S.(R.\theta \,{}_p\!\oplus\; T.\theta) \\
= & \{\text{definition of probabilistic choice and sequential composition}\} \\
& (S; (R \,{}_P\!\oplus\; T)).\theta \\
= & \{\text{assumption (2) on } S\} \\
& (S; R \,{}_P\!\oplus\; S; T).\theta \\
= & \{\text{definition of probabilistic choice and sequential composition}\} \\
& S.(R.\theta) \,{}_p\!\oplus\; S.(T.\theta) \\
= & \{\text{definition of } R \text{ and } T \text{ and } \theta\} \\
& S.\phi \,{}_p\!\oplus\; S.\psi.
\end{aligned}$$

To verify that the aside holds, observe that for any given ϕ and ψ, R, T and θ can always be chosen such that R and T are nondeterministic transformers. For example we have that

$$\begin{aligned}
R & \triangleq \{(\lambda\sigma \in \Sigma \bullet 1 \times (\phi.\sigma \geq \psi.\sigma \wedge \phi.\sigma \neq 0) + \tfrac{\phi.\sigma}{\psi.\sigma} \times (\phi.\sigma < \psi.\sigma))\} \\
T & \triangleq \{(\lambda\sigma \in \Sigma \bullet \tfrac{\psi.\sigma}{\phi.\sigma} \times (\phi.\sigma \geq \psi.\sigma \wedge \phi.\sigma \neq 0) + 1 \times (\phi.\sigma < \psi.\sigma))\} \\
\theta & \triangleq (\lambda\sigma \in \Sigma \bullet \phi.\sigma \times (\phi.\sigma \geq \psi.\sigma \wedge \phi.\sigma \neq 0) + \psi.\sigma \times (\phi.\sigma < \psi.\sigma))
\end{aligned}$$

satisfy the requirement that $R.\theta = \phi$ and $T.\theta = \psi$. For example, $T.\theta$ equals

$$\begin{array}{ll} & (\lambda\sigma \in \Sigma \bullet \quad \frac{\psi.\sigma}{\phi.\sigma} \times (\phi.\sigma \geq \psi.\sigma \wedge \phi.\sigma \neq 0) + \quad 1 \times (\phi.\sigma < \psi.\sigma))\times \\ & (\lambda\sigma \in \Sigma \bullet \quad \phi.\sigma \times (\phi.\sigma \geq \psi.\sigma \wedge \phi.\sigma \neq 0) + \quad \psi.\sigma \times (\phi.\sigma < \psi.\sigma)) \\ = & (\lambda\sigma \in \Sigma \bullet \ \frac{\psi.\sigma}{\phi.\sigma} \times \phi.\sigma \times (\phi.\sigma \geq \psi.\sigma \wedge \phi.\sigma \neq 0) + 1 \times \psi.\sigma \times (\phi.\sigma < \psi.\sigma)) \\ = & \psi \end{array}$$

(Recall that probabilistic assertions are nondeterministic transformers.) □

In Fig. 4 we list the equivalent algebraic form of some of the other healthiness conditions.[4] These may be verified in a similar way to Prop. 1. We will see how these algebraic conditions play an important role in the abstract algebras.

3 General Refinement Algebra

In this section we provide a brief introduction to von Wright's general refinement algebra [16]. In the coming sections we then propose probabilistic extensions to this algebra. Like the demonic refinement algebra [15], the general refinement algebra has the following signature

$$(;, \sqcap, ^*, ^\omega, \top, \bot, 1),$$

where, in a program interpretation, constants $\top$, $\bot$, and 1 may be taken to represent the greatest program, magic, the least program, abort and the program which performs "no action", skip; ";" may be taken to represent sequential composition; $\sqcap$ represents demonic nondeterministic choice; x^* is the program $(\nu X \bullet x; X \sqcap 1)$, which iterates its argument x any finite number of times; and x^ω, $(\mu X \bullet x; X \sqcap 1)$, is a recursive statement which executes its argument any finite or infinite number of times.[5]

Aptly named, the general refinement algebra is very general indeed, and captures an interesting set of similarities between a large number of different program models (with a total-correctness semantics). For instance, it is sound with respect to the set of monotonic predicate transformers (where the operators are given their usual weakest-precondition semantics) [16], as well as both the sets of nondeterministic and dually nondeterministic one-bounded expectation transformers [9]. It is also sound with respect to the sets of nondeterministic$_+$ and dually nondeterministic$_+$ expectation transformers.

The general refinement algebra is a generalisation of the demonic refinement algebra (dRA). A discussion of the axiomatisation and the differences between

[4] Note that we have not given subtraction a point-free representation, since this would require the definition of a subtraction operator, $(S \ominus T).\phi \triangleq (S.\phi - T.\phi) \sqcup \mathsf{False}$. Given such a definition, we could then give the equivalent form as $S \ominus [\neg c]; \mathsf{abort} \sqsubseteq S; (\mathsf{skip} \ominus [\neg c]; \mathsf{abort})$.

[5] As noted by von Wright [15], the constant $\bot$ could have been removed from the signature, and defined syntactically as 1^ω. We include it explicitly in the signature to simplify the presentation.

these two algebras can be found in [16][6] and [9]. As a notational shorthand, we omit ";" and use juxtaposition to represent sequential composition when no confusion can arise.

Definition 1. A *general refinement algebra* (gRA) is a structure over the signature $(;, \sqcap, ^*, ^\omega, \top, \bot, 1)$ satisfying the following axioms and rules

$$\begin{aligned}
x \sqcap (y \sqcap z) &= (x \sqcap y) \sqcap z, && (3)\\
x \sqcap y &= y \sqcap x, && (4)\\
x \sqcap \top &= x, && (5)\\
x \sqcap \bot &= \bot, && (6)\\
x \sqcap x &= x, && (7)\\
x(yz) &= (xy)z, && (8)\\
1x = x &= x1, && (9)\\
\top x &= \top, && (10)\\
\bot x &= \bot, && (11)\\
x(y \sqcap z) &\sqsubseteq xy \sqcap xz, && (12)\\
(x \sqcap y)z &= xz \sqcap yz, && (13)\\
x^* &= 1 \sqcap xx^*, && (14)\\
x \sqsubseteq yx \sqcap z &\Rightarrow x \sqsubseteq y^* z, && (15)\\
x^\omega &= 1 \sqcap xx^\omega \text{ and} && (16)\\
yx \sqcap z \sqsubseteq x &\Rightarrow y^\omega z \sqsubseteq x, && (17)
\end{aligned}$$

where the order $\sqsubseteq$ is defined by $x \sqsubseteq y \triangleq x \sqcap y = x$. □

It is easy to prove that all the operators are monotonic in all their arguments with respect to $\sqsubseteq$ and that $\sqsubseteq$ is a partial order [9].

For some program models satisfying semi-cocontinuity[7], an extra weak iteration induction axiom

$$x \sqsubseteq x(y \sqcap 1) \sqcap z \Rightarrow x \sqsubseteq zy^*, \tag{18}$$

may be shown to be sound [9]. (For example, we do have that the nondeterministic$_+$ and dually nondeterministic$_+$ expectation transformers on *finite* state spaces satisfy this property.) We refer to a gRA extended with axiom (18) as a *continuous general refinement algebra* (cont. gRA).

Healthiness Conditions. It can be seen that the healthiness condition, monotonicity, is included in the axiomatisation of gRA (axiom (12)). Conjunctivity is not included – it does not hold for probabilistic programs – but can be used to

[6] Note that a typographical error appears in [16]: the distributivity axioms (12) and (13) are mistakenly written as $x(y \sqcap z) = xy \sqcap xz$ and $(x \sqcap y)z \sqsubseteq xz \sqcap yz$.

[7] See Fig. 4.

identify useful subsets of the carrier set. An element x is said to be *conjunctive* if it satisfies

$$x(y \sqcap z) = xy \sqcap xz \tag{19}$$

for all y and z in the carrier set [9,13].

4 Probabilistic Refinement Algebra

The probabilistic algebras we now define extend the signature of gRA with three extra operators: $+$, $^{\circledast}$ and ‡.

The binary operator $+$ may be given a probabilistic program interpretation as our $+$ operator from Sect. 2. This (slightly exotic) operator shall be used to define probabilistic choice. Unary operators $^{\circledast}$ and ‡ are our weak and strong plus iteration operators, respectively, where

$$x^{\circledast} \triangleq (\mu X \bullet x; X + 1) \text{ and } x^{\ddagger} \triangleq (\nu X \bullet x; X + 1)$$

in the model. These iteration operators are the "plus" equivalent of the angelic iteration operators included in Solin's angelic and demonic refinement algebra [13]: an extension of gRA in which angelic choice is axiomatised. They are new (as are the probabilistic while-loops below).

Like the plus operator, the plus iteration operators have a useful interpretation in the model when they are restricted to iterations of a particular form. For any probabilistic assertion $\{\phi\}$ and expectation transformer S, we have that $(\{\phi\}; S)^{\circledast}; \{\neg\phi\}$ and $(\{\phi\}; S)^{\ddagger}; \{\neg\phi\}$ are *probabilistic* while-loops. For example, using unfolding, we can see that program $U \triangleq (\{\frac{3}{4}\}; S)^{\circledast}; \{\frac{1}{4}\}$

$$\begin{aligned} U &= (\{\tfrac{3}{4}\}; S)^{\circledast}; \{\tfrac{1}{4}\} \\ &= (1 + \{\tfrac{3}{4}\}; S; (\{\tfrac{3}{4}\}; S)^{\circledast}); \{\tfrac{1}{4}\} \\ &= \{\tfrac{1}{4}\} + \{\tfrac{3}{4}\}; S; (\{\tfrac{3}{4}\}; S)^{\circledast}; \{\tfrac{1}{4}\} \\ &= \{\tfrac{1}{4}\} + \{\tfrac{3}{4}\}; S; U \end{aligned}$$

is a loop which, on each iteration, has a probability of $\frac{1}{4}$ of ceasing execution and a probability of $\frac{3}{4}$ of performing S and continuing execution. In the expectation transformer model we have that infinite iterations of the loop body in $x^{\circledast}$ are associated with aborting behaviour, but infinite iterations of the loop body of $x^{\ddagger}$ are associated with magical behaviour. For example, using induction we can show $(\{\mathsf{True}\})^{\circledast}; \{\mathsf{False}\} = \mathsf{abort}$ and $(\{\mathsf{True}\})^{\ddagger}; \{\mathsf{False}\} = \mathsf{magic}$.

First we introduce a very general extension to gRA, one which is sound with respect to both the nondeterministic$_+$ and dually nondeterministic$_+$ one-bounded expectation transformer models from Sect. 2. After this we extend the algebra with assertions (probabilistic, constant and standard).

Definition 2. A *general probabilistic refinement algebra* (gpRA) is a structure over the signature

$$(;, \sqcap, +, ^{*}, ^{\circledast}, ^{\omega}, ^{\ddagger}, \top, \bot, 1),$$

such that $(;, \sqcap, ^*, ^\omega, \top, \bot, 1)$ is a gRA, and

$$\begin{aligned} x + (y + z) &= (x + y) + z, & (20)\\ x + y &= y + x, & (21)\\ x + \top &= \top & (22)\\ x + \bot &= x & (23)\\ x \sqcap (y + z) &\sqsubseteq (x \sqcap y) + (x \sqcap z) & (24)\\ x + (y \sqcap z) &= (x + y) \sqcap (x + z) & (25)\\ (x + y)z &= xz + yz, & (26)\\ x^{\circledast} &= 1 + xx^{\circledast}, & (27)\\ yx + z \sqsubseteq x &\Rightarrow y^{\circledast}z \sqsubseteq x, & (28)\\ x^{\ddagger} &= 1 + xx^{\ddagger} \text{ and} & (29)\\ x \sqsubseteq yx + z &\Rightarrow x \sqsubseteq y^{\ddagger}z, & (30) \end{aligned}$$

hold. □

As for the other operators from gRA, operators $+$, $^{\circledast}$ and ‡ may easily be shown to be monotonic in their arguments with respect to the refinement ordering $\sqsubseteq$. For example, since the operator $+$ is commutative (21), to verify monotonicity of $+$ it is sufficient to show that

$$\begin{aligned} &x + y \sqsubseteq x + z\\ \Leftrightarrow &\{\text{refinement ordering}\}\\ &(x + y) \sqcap (x + z) = x + y\\ \Leftrightarrow &\{\text{axiom (25)}\}\\ &x + (y \sqcap z) = x + y\\ \Leftarrow &\{\text{refinement ordering}\}\\ &y \sqsubseteq z. \end{aligned}$$

A comparison to the axiomatisation of angelic choice in the angelic and demonic refinement algebra (adRA) [14] reveals expected similarities: the plus operator satisfies all of the axioms of angelic choice ($\sqcup$) other than:

$$\begin{aligned} x \sqcup x &= x,\\ x(y \sqcup z) &\sqsupseteq xy \sqcup xz \text{ and}\\ x \sqcap (y \sqcup z) &\sqsupseteq (x \sqcap y) \sqcup (x \sqcap z). \end{aligned}$$

Operators $^{\circledast}$ and ‡ also share the same axiomatisation as angelic iteration operators $^{\phi}$ and † from adRA.

5 Probabilistic Refinement Algebra with Assertions

Guards (sometimes known as tests) and assertions have been defined in the refinement algebras (see for example [9,15]). In gpRA we not only define guards and assertions, but probabilistic guards and assertions, and also constant probabilistic guards and assertions. We start by adding assertions; then for every probabilistic assertion, a, we can define a corresponding guard $a^\diamond$ by $a^\diamond \triangleq 1 \;{}_a\!\oplus\; \top$, which behaves like 1 with probability a and like $\top$ with probability $\bar{a}$.

Definition 3. A *general probabilistic refinement algebra with assertions* (gpRAa) is a structure

$$(\mathsf{X}, \mathsf{PA}, \mathsf{CA}, \mathsf{A}, ^{-}, ;, \sqcap, +, ^{*}, ^{\circledast}, ^{\omega}, ^{\ddagger}, \top, \bot, 1),$$

such that $(\mathsf{X}, ;, \sqcap, +, ^{*}, ^{\circledast}, ^{\omega}, ^{\ddagger}, \top, \bot, 1)$ *is a* gpRA*, and the set of* probabilistic assertions*,* $\mathsf{PA} \subseteq \mathsf{X}$*, satisfies the following properties. Every element* a *of* PA *has a unique complement* $\bar{a} \in \mathsf{PA}$ *satisfying* $a + \bar{a} = 1$*.* PA *is closed under nondeterministic choice, sequential composition, and probabilistic choice, where, given any elements* $x, y \in \mathsf{X}$*,* $a \in \mathsf{PA}$*, a statement of the form*

$$x \;{}_a\!\oplus\; y \triangleq ax + \bar{a}y$$

represents a probabilistic choice statement. Elements of PA *commute over sequential composition, are conjunctive, and distribute over probabilistic choices. That is, we require that for all* $a, b \in \mathsf{PA}$*, and* $x, y \in \mathsf{X}$*,* $ab = ba, a(x \sqcap y) = ax \sqcap ay$ *and* $a(x \;{}_b\!\oplus\; y) = ax \;{}_b\!\oplus\; ay$*. For all* $a, b \in \mathsf{PA}$*, we also require* $a\top = b\top \Rightarrow a = b$*, and* $a^\diamond \sqcap b^\diamond = (\overline{\bar{a} \sqcap \bar{b}})^\diamond$*.*

The set of constant assertions*,* $\{1, \bot\} \subseteq \mathsf{CA} \subseteq \mathsf{PA}$*, is closed under nondeterministic choice, sequential composition, complement, and constant probabilistic choice, where a constant probabilistic choice,* $x \;{}_c\!\oplus\; y$ *is a probabilistic choice where* $c \in \mathsf{CA}$*. For each element* c *of* CA*, and* $x \in \mathsf{X}$*,* $cx \sqsubseteq xc$ *(cf. sub-scaling).*

The (standard) assertions $\{1, \bot\} \subseteq \mathsf{A} \subseteq \mathsf{PA}$ *are closed under nondeterministic choice, sequential composition and complement. For any two assertions* a *and* a' *in* A *we require* $a\bar{a} = \bar{a}a = \bot$ *and* $aa' = a \sqcap a'$*.* □

For every probabilistic assertion $a \in \mathsf{PA}$, we define its corresponding *probabilistic guard* $a^\diamond$ to be $1 \;{}_a\!\oplus\; \top$, and we refer to the sets of probabilistic, constant, and standard guards as PG, CG and G, respectively. The complement of a guard $g = a^\diamond$ is defined by $\tilde{g} \triangleq (\bar{a})^\diamond$, (i.e., $\tilde{g} = 1 \;{}_{\bar{a}}\!\oplus\; \top$).[8]

In the expectation transformer model, standard guards and assertions correspond to the guards and assertions which are specified using predicates, while probabilistic guards and assertions are the more general commands which are defined using expectations. Similarly constant probabilistic assertions and guards are those probabilistic guards and assertions, respectively, defined using constant probability functions. The complement of a probabilistic assertion $\{\phi\}$, is simply $\{\neg\phi\}$, and $\{\phi\}^\diamond = [\phi]$.

A number of useful properties of the three different sets of guards may be derived from the definition of the algebra. In particular, the set of probabilistic guards PG satisfies the following proposition.

[8] We use different symbols to represent guard and assertion complement, since 1 is both a guard and an assertion, but $\bar{1} = \bot \neq \top = \tilde{1}$ unless we have a one-point model.

Proposition 2. PG *is closed under probabilistic choice, nondeterministic choice, complement, and sequential composition. Elements of* PG *commute over sequential composition, are conjunctive, and distribute over probabilistic choices. For all* $g, p \in \mathsf{PG}$, $g\bot = p\bot \Rightarrow g = p$.

It may be shown that the set of constant guards, $\mathsf{CG} \subseteq \mathsf{PG}$, is closed under constant probabilistic choice, nondeterministic choice, complement, and sequential composition; and the set of standard guards, $\mathsf{G} \subseteq \mathsf{PG}$, forms a Boolean algebra.

Proposition 3. $(\mathsf{G}, \sqcap, ;, \tilde{\ }, 1, \top)$ *is a Boolean algebra.*

Note that probabilistic assertion axiom $a^{\diamond} \sqcap b^{\diamond} = (\overline{\bar{a} \sqcap \bar{b}})^{\diamond}$ is useful (if not necessary) for proving closure of the guard sets under nondeterministic choice.

Since the guards are conjunctive, and they form a Boolean algebra (as above), they satisfy the guard axioms from gRA [9]. We also have that the set of assertions satisfy the assertion axioms from gRA, although why this is the case may be less immediately apparent. In gRA, assertions are defined in terms of guards, and not the other way around: each guard g is defined to have a corresponding assertion $g^{\circ} \triangleq \tilde{g}\bot \sqcap 1$. In our definition we can verify that this relationship between guards and assertions holds. That is, for each guard $g = a^{\diamond}$, $g^{\circ} \triangleq \tilde{g}\bot \sqcap 1 = a$.

This observation means that it is possible to reuse basic assertion and guard properties from earlier work. For example, the following proposition holds.

Proposition 4. *For standard assertion* $a \in \mathsf{A}$,

$$aa^{\diamond} = a \text{ and} \tag{31}$$

$$a\bar{a}^{\diamond} = a\top. \tag{32}$$

Note that we require probabilistic assertions to satisfy the property $a\top = b\top \Rightarrow a = b$. This property seems to be important, but not derivable from the other constraints.[9] In particular it is useful for deriving the enabledness and termination operator properties listed in Sect. 8. The following generalisation of this property can be shown in the algebra:

$$\begin{aligned} & a\top \sqsubseteq b\top \\ \Leftrightarrow\ & a\top \sqcap b\top = a\top \\ \Leftrightarrow\ & (a \sqcap b)\top = a\top \\ \Leftrightarrow\ & a \sqcap b = a \\ \Leftrightarrow\ & a \sqsubseteq b. \end{aligned}$$

Similarly, we can show that for probabilistic guards g and p,

$$g\bot \sqsubseteq p\bot \Leftrightarrow g \sqsubseteq p. \tag{33}$$

Healthiness Conditions. The algebraic version of healthiness condition *subscaling* is used in the definition of constant assertions: for every element x of

[9] Interestingly, this property may be derived in gRA for standard assertions a and b (Kim Solin, personal correspondence January 2008).

the carrier set, we require x to satisfy sub-scaling, i.e., $cx \sqsubseteq xc$, for all constant assertions c.

The definition of constant probabilistic assertions makes it possible to express healthiness conditions $\oplus$-distributivity and $\oplus$-subdistributivity in the algebra.

Definition 4. *An element x of the carrier set is* $\oplus$-distributive *if, for all constants $c \in \mathsf{CA}$ and elements y and z from the carrier set X, it satisfies*

$$xy \;{}_c\!\oplus\; xz = x(y \;{}_c\!\oplus\; z). \tag{34}$$

Definition 5. *Similarly, $x \in \mathsf{X}$ is* $\oplus$-subdistributive *if*

$$xy \;{}_c\!\oplus\; xz \sqsubseteq x(y \;{}_c\!\oplus\; z), \tag{35}$$

for all $c \in \mathsf{CA}$ and elements $y, z \in \mathsf{X}$.

These definitions should be compared with those in Fig. 4.

A Strengthened Algebra. The healthiness condition $\oplus$-subdistributivity can be used to define a stronger algebra, suitable for the nondeterministic$_+$, but not the dually nondeterministic$_+$ one-bounded expectation transformer model.

We define a *probabilistic refinement algebra with assertions* (pRAa) to be a gpRAa for which (35) also holds, for all constant assertions $c \in \mathsf{CA}$ and elements x, y and z from the carrier set, X.

When extended with the additional axiom (18), both gpRAa and pRAa are referred to as continuous.

6 Probabilistic Choice Statements and Loops

As in gRA [9], (standard) guards may be used to define conditional statements and while-loops. For any guard g and elements x and y from the carrier set

$$\mathsf{if}\ g\ \mathsf{then}\ x\ \mathsf{else}\ y\ \mathsf{fi} \triangleq gx \sqcap \tilde{g}y \tag{36}$$

represents a *conditional choice* between x and y, and

$$(gx)^\omega \tilde{g}$$

is a *while-loop* which iterates x until g ceases to hold.

Similarly, probabilistic assertions may be used to define probabilistic choices and probabilistic iteration statements: probabilistic choice statements have already been introduced, and iterations of the form

$$(ax)^{\circledast}\bar{a} \text{ or } (ax)^{\ddagger}\bar{a}$$

are referred to as *probabilistic while-loops*.

In [6] McIver and Morgan identify a number of properties satisfied by probabilistic choice statements in a non-angelic expectation transformer model (slightly different, but similar to the nondeterministic$_+$ one-bounded expectation transformers). All but one of these properties,[10] $\oplus$-subdistributivity (35), are easily derivable in gpRAa: recall that $\oplus$-subdistributivity does not hold for angelic probabilistic programs. Let a, b, c, d be probabilistic assertions, and x, y and z be elements of the carrier set. We have that

$$x \,{}_{a}\!\oplus\, x = x \tag{37}$$

$$x \,{}_{a}\!\oplus\, y = y \,{}_{\bar{a}}\!\oplus\, x \tag{38}$$

$$x \,{}_{1}\!\oplus\, y = x \tag{39}$$

$$(a = cd \wedge \bar{a}b = c\bar{d} \wedge \bar{a}\bar{b} = \bar{c}) \Rightarrow x \,{}_{a}\!\oplus\, (y \,{}_{b}\!\oplus\, z) = (x \,{}_{d}\!\oplus\, y) \,{}_{c}\!\oplus\, z \tag{40}$$

$$(x \,{}_{a}\!\oplus\, y)z = xz \,{}_{a}\!\oplus\, yz \tag{41}$$

$$(x \sqcap y) \,{}_{a}\!\oplus\, z = (x \,{}_{a}\!\oplus\, z) \sqcap (y \,{}_{a}\!\oplus\, z) \tag{42}$$

Note that for (40), if we write the left and right sides of the equality using "+", the equality becomes $ax + \bar{a}by + \bar{a}\bar{b}z = cdx + c\bar{d}y + \bar{c}z$, which holds trivially given the assumptions.

It is of interest to note that probabilistic choices may be viewed as a generalisation of conditional statements.[11] In particular we have that, for standard assertion a, and elements x and y

$$x \,{}_{a}\!\oplus\, y = \mathsf{if}\ a^{\diamond}\ \mathsf{then}\ x\ \mathsf{else}\ y\ \mathsf{fi}.$$

This is shown by writing the conditional using its definition (36), and transforming as follows,

$$\begin{aligned}
& a^{\diamond}x \sqcap \bar{a}^{\diamond}y \\
= & \{\text{as } a + \bar{a} = 1,\ 1 \text{ is unit (9)}\} \\
& (a + \bar{a})(a^{\diamond}x \sqcap \bar{a}^{\diamond}y) \\
= & \{\text{right }+\text{-distributivity (26)}\} \\
& a(a^{\diamond}x \sqcap \bar{a}^{\diamond}y) + \bar{a}(a^{\diamond}x \sqcap \bar{a}^{\diamond}y) \\
= & \{\text{as assertions } a \text{ and } \bar{a} \text{ are conjunctive}\} \\
& (aa^{\diamond}x \sqcap a\bar{a}^{\diamond}y) + (\bar{a}a^{\diamond}x \sqcap \bar{a}\bar{a}^{\diamond}y) \\
= & \{\text{as } aa^{\diamond} = a \text{ and } a\bar{a}^{\diamond} = a\top \text{ by (31) and (32)}\} \\
& (ax \sqcap a\top y) + (\bar{a}\top x \sqcap \bar{a}y) \\
= & \{\top \text{ is left-annihilating (10) and assertions are conjunctive}\} \\
& a(x \sqcap \top) + \bar{a}(\top \sqcap y) \\
= & \{\top \text{ is greatest element (5)}\} \\
& ax + \bar{a}y \\
= & \ x \,{}_{a}\!\oplus\, y.
\end{aligned}$$

From this we can see, for example, that under certain circumstances our probabilistic while-loops reduce to their non-probabilistic equivalents: when a is a standard assertion,

$$(ax)^{\circledast}\bar{a} = (a^{\diamond}x)^{\omega}\bar{a}^{\diamond} \text{ and } (ax)^{\ddagger}\bar{a} = (a^{\diamond}x)^{*}\bar{a}^{\diamond}.$$

[10] We have selected the independent properties from their work.

[11] In earlier work [4] Hehner also made the observation that probabilistic choices can be viewed as generalised conditional statements.

We show the first of these. Using $\circledast$-induction, $(ax)^{\circledast}\bar{a} \sqsubseteq (a^{\diamond}x)^{\omega}\bar{a}^{\diamond}$ provided

$$ax(a^{\diamond}x)^{\omega}\bar{a}^{\diamond} + \bar{a} \sqsubseteq (a^{\diamond}x)^{\omega}\bar{a}^{\diamond},$$

which holds because

$$\begin{aligned}
&ax(a^{\diamond}x)^{\omega}\bar{a}^{\diamond} + \bar{a}\\
=\;&\{\text{using the if-statement equivalent form above}\}\\
&a^{\diamond}x(a^{\diamond}x)^{\omega}\bar{a}^{\diamond} \sqcap \bar{a}^{\diamond}\\
=\;&(a^{\diamond}x(a^{\diamond}x)^{\omega} \sqcap 1)\bar{a}^{\diamond}\\
=\;&(a^{\diamond}x)^{\omega}\bar{a}^{\diamond}.
\end{aligned}$$

And the reverse direction holds by ω-induction provided

$$a^{\diamond}x(ax)^{\circledast}\bar{a} \sqcap \bar{a}^{\diamond} \sqsubseteq (ax)^{\circledast}\bar{a},$$

which holds because

$$\begin{aligned}
&a^{\diamond}x(ax)^{\circledast}\bar{a} \sqcap \bar{a}^{\diamond}\\
=\;&\{\text{using the if-statement equivalent form above}\}\\
&ax(ax)^{\circledast}\bar{a} + \bar{a}\\
=\;&(ax(ax)^{\circledast} + 1)\bar{a}\\
=\;&(ax)^{\circledast}\bar{a}.
\end{aligned}$$

Since demonic choices may be refined by probabilistic choices,

$$\begin{aligned}
&x \sqcap y\\
=\;&\{\text{idempotence of probabilistic choice (37)}\}\\
&(x \sqcap y) \;{}_{a}\!\oplus\; (x \sqcap y)\\
\sqsubseteq\;&\{\text{monotonicity of plus, and hence probabilistic choice}\}\\
&x \;{}_{a}\!\oplus\; y,
\end{aligned}$$

it is easy to show that strong and weak iterations may be refined by probabilistic while-loops in the following way,

$$x^{\omega} \sqsubseteq (ax)^{\circledast}\bar{a} \text{ and } x^{*} \sqsubseteq (ax)^{\ddagger}\bar{a},$$

where a is a probabilistic assertion and x is a member of the carrier set.

We show the first of these: $x^{\omega} \sqsubseteq (ax)^{\circledast}\bar{a}$; this holds by ω-induction provided

$$x(ax)^{\circledast}\bar{a} \sqcap 1 \sqsubseteq (ax)^{\circledast}\bar{a},$$

which holds because

$$\begin{aligned}
&x(ax)^{\circledast}\bar{a} \sqcap 1\\
\sqsubseteq\;&\{\text{as } x \sqcap y \sqsubseteq x \;{}_{a}\!\oplus\; y \text{ for any } x \text{ and } y\}\\
&ax(ax)^{\circledast}\bar{a} + \bar{a}1\\
=\;&(ax(ax)^{\circledast} + 1)\bar{a}\\
=\;&(ax)^{\circledast}\bar{a}.
\end{aligned}$$

It is possible to derive other simple transformation rules for probabilistic while-loops in gpRAa, in the same way it is simple to reason about while-loops in gRA. For example, we have the following data refinement theorem for strong probabilistic iteration.

Proposition 5. *For any constant probabilistic assertion c and elements x, y and z from the carrier set, if z is $\oplus$-subdistributive and*

$$xz \sqsubseteq zy$$

then $(cx)^{\circledast}\bar{c}z \sqsubseteq z(cy)^{\circledast}\bar{c}$.

Proof. The proposition may be shown to hold using the following argument.

$$
\begin{array}{l}
\quad (cx)^{\circledast}\bar{c}z \sqsubseteq z(cy)^{\circledast}\bar{c} \\
\Leftarrow \{\text{induction (28)}\} \\
\quad cxz(cy)^{\circledast}\bar{c} + \bar{c}z \sqsubseteq z(cy)^{\circledast}\bar{c} \\
\Leftarrow \{\text{assumption } xz \sqsubseteq zy \text{ and unfolding (27)}\} \\
\quad czy(cy)^{\circledast}\bar{c} + \bar{c}z \sqsubseteq z(cy(cy)^{\circledast} + 1)\bar{c} \\
\Leftrightarrow \{\text{right } +\text{-distributivity (26)}\} \\
\quad czy(cy)^{\circledast}\bar{c} + \bar{c}z \sqsubseteq z(cy(cy)^{\circledast}\bar{c} + \bar{c}) \\
\Leftrightarrow \{\text{assumption } z \text{ is } \oplus\text{-subdistributive (35)}\} \\
\quad \mathsf{true}.
\end{array}
$$

□

The above proposition requires the commuting program z to be $\oplus$-subdistributive. In the more general case, where z does not satisfy $\oplus$-subdistributivity, the theorem does not necessarily hold.

7 Correctness Assertions

In [15], von Wright observed that *total-correctness assertions* (Hoare triples) for standard programs could be encoded in dRA (and hence gRA) as properties of the form

$$\top = px\tilde{q} \quad \text{or equivalently} \quad \tilde{p}x \sqsubseteq x\tilde{q} \quad \text{or} \quad \tilde{p}\bot \sqsubseteq x\tilde{q}, \tag{43}$$

where p and q are guards, and x is an element of the carrier set. A total-correctness property of the form (43) can be taken to mean that from a state in which p holds, x can certainly terminate and reach a state in which q holds (i.e., for predicate transformer $x = S$, and predicates p', q' such that $p = [p']$ and $q = [q']$, $p' \Rightarrow S.q'$).

Since probabilistic guards and assertions are defined in our algebra, it is possible to describe total-correctness assertions (involving expectations) for probabilistic programs. For a one-bounded expectation transformer $x = S$, and probabilistic guards $p = [\phi]$, $q = [\psi]$, we have that $\phi \leq S.\psi$ if and only if

$$\tilde{p}\bot \sqsubseteq x\tilde{q}. \tag{44}$$

In the case that p and q are standard guards this property may be equivalently rewritten as the other two properties in (43).

In [15], von Wright also observed that in dRA the following equivalent properties,

$$px = pxq \quad \text{and} \quad xq \sqsubseteq px \quad \text{and} \quad p^{\circ}x = p^{\circ}xq^{\circ} \quad \text{and} \quad p^{\circ}x \sqsubseteq xq^{\circ}, \tag{45}$$

where p and q are guards, and x is an element from the carrier set, could be used to express *weak-correctness assertions*. Informally, they specify that from an initial state in which p holds, x must either abort, or reach a state in which q holds. The equivalences in (45) may also easily be shown to hold in our algebras. Although we are not aware of any generalisation of these properties to weak-correctness assertions involving expectations (which are not predicates), we find that they are also a useful way to express weak-correctness assertions (involving predicates) for probabilistic programs.

For example, we show how they may be used in the following theorem, which states that a $\oplus$-distributive element x may be shown to distribute over a probabilistic choice involving a probabilistic assertion a, provided a can be written as a disjoint sum of terms of the form $a_i c_i$, where a_i is a standard assertion, c_i is a constant probabilistic assertion, and a_i is (weakly) invariant over x: $a_i x = a_i x a_i$.

Proposition 6. *Given $\oplus$-distributive x and probabilistic assertion a, we have that for any y and z,*

$$axy + \bar{a}xz = x(ay + \bar{a}z)$$

if for some finite set of standard assertions $\{i \in I \bullet a_i\}$, and constant probabilistic assertions $\{i \in I \bullet c_i\}$,

$$1 = (\textstyle\sum i \in I \bullet a_i), \tag{46}$$

$$a = (\textstyle\sum i \in I \bullet a_i c_i) \text{ and} \tag{47}$$

$$a_i x = a_i x a_i, \tag{48}$$

where $(\sum i \in I \bullet a_i)$ represents the finite sum over the elements of I of the term a_i. The conditions (46-47) describe that for each pairwise disjoint set of states a_i, a takes the constant value c_i; and (48) requires x to preserve each of these sets of states: from a state in which a_i holds, either x must abort or terminate in a state in which a_i also holds.

Proof. First we show that $a_i a = a_i c_i$ and $a_i \bar{a} = a_i \bar{c_i}$ using (46) and (47). From (46) we can deduce that the elements a_i for $i \in I$ are pairwise disjoint:

$$\forall i, j \bullet i \neq j \Rightarrow a_i a_j \sqsubseteq a_i(\textstyle\sum j \in I - \{i\} \bullet a_j) = a_i \bar{a}_i = \bot.$$

And so using (47) we have that

$$a_i a = a_i(\textstyle\sum j \in I \bullet a_j c_i) = (\textstyle\sum j \in I \bullet a_i a_j c_i) = a_i a_i c_i = a_i c_i.$$

Since the negation of a may trivially be shown to be $(\sum i \in I \bullet a_i \bar{c_i})$, we also have that $a_i \bar{a} = a_i \bar{c_i}$.

Given the assumptions (46-48),

$$
\begin{aligned}
& x(ay+\bar{a}z)\\
= {} & \{1 \text{ is unit } (9), \text{ assumption } (46)\}\\
& (\textstyle\sum i \in I \bullet a_i)x(ay+\bar{a}z)\\
= {} & \{\text{right } +\text{-distributivity } (26),\ x \text{ preserves } a_i \ (48)\}\\
& (\textstyle\sum i \in I \bullet a_i x a_i(ay+\bar{a}z))\\
= {} & \{\text{distributivity of assertions, } (46\text{-}47) \text{ implies } a_i a = a_i c_i \text{ and } a_i\bar{a} = a_i\bar{c_i}\}\\
& (\textstyle\sum i \in I \bullet a_i x(a_i c_i y + a_i\bar{c_i}z))\\
= {} & \{\text{distributivity of assertions, } x \text{ preserves } a_i \ (48)\}\\
& (\textstyle\sum i \in I \bullet a_i x(c_i y + \bar{c_i}z))\\
= {} & \{\text{assumption } x \text{ is } \oplus\text{-distributive}\}\\
& (\textstyle\sum i \in I \bullet a_i(c_i xy + \bar{c_i}xz))\\
= {} & \{\text{distributivity of assertions, assumptions } (46\text{-}47)\}\\
& (\textstyle\sum i \in I \bullet a_i(axy + \bar{a}xz))\\
= {} & \{\text{right } +\text{-distributivity } (26)\}\\
& (\textstyle\sum i \in I \bullet a_i)(axy + \bar{a}xz)\\
= {} & \{1 \text{ is unit } (9), \text{ assumption } (46)\}\\
& (axy + \bar{a}xz). \qquad \square
\end{aligned}
$$

8 Termination and Enabledness

In [1], Desharnais, Möller and Struth observed that the addition of a domain (enabledness) operator to Kleene Algebra with tests considerably augments the expressiveness of the algebra. Inspired by these observations, Solin and von Wright have extended the demonic refinement algebra with operators for determining when elements are terminating or enabled [14]. Here we define the probabilistic counterparts to these operators.

8.1 Termination

The *termination operator* τ is a unary operator that maps a carrier-set element of a gpRAa to the unique probabilistic assertion which satisfies,

$$\tau x\top = x\top. \tag{49}$$

We will call a gpRAa with a termination operator a gpRAat. Given a program interpretation, the termination operator describes, for each initial state, the probability with which a given program will terminate (i.e., not abort). We define $\tau S \triangleq \{S.\mathsf{True}\}$ in the model.

Note that the introduction of the termination operator extends gpRAa with the extra condition that for all elements x from the carrier set, there must exist an assertion a such that $a\top = x\top$. This extension specifies an important algebraic property of our motivating program models: it describes the fact that magic right-annihilates non-aborting program behaviour. This could not be expressed without the aid of probabilistic assertions.

The following useful properties of τ follow immediately from the definition,

$$\tau a = a \tag{50}$$
$$\tau(ax) \sqsubseteq a \tag{51}$$
$$\tau(x\tau y) = \tau(xy) \tag{52}$$
$$\tau(x \sqcap y) = \tau x \sqcap \tau y \tag{53}$$
$$\tau(x \;{}_a\!\oplus\; y) = \tau x \;{}_a\!\oplus\; \tau y \tag{54}$$

where a is a probabilistic assertion and x and y are from the carrier set. If τx is more specifically a (standard) assertion, then we also have that

$$x = (\tau x)x \text{ and} \tag{55}$$
$$\tau x = x\top \sqcap 1. \tag{56}$$

Apart from the probabilistic choice property, (54), properties (49-56) are used (with probabilistic assertions replaced by standard assertions) to define the termination operator in dRA [14]. (In that algebra, (50) is derivable.)

The proofs of properties (50-56) are trivial since assertions satisfy the useful property $a\top = b\top \Leftrightarrow a = b$. For example, to show property (52), we have that,

$$\begin{aligned}
&\tau(x\tau y) = \tau(xy)\\
\Leftrightarrow\; &\tau(x\tau y)\top = \tau(xy)\top\\
\Leftrightarrow\; &x\tau y\top = xy\top\\
\Leftrightarrow\; &xy\top = xy\top\\
\Leftrightarrow\; &\mathsf{True}.
\end{aligned}$$

Also, uniqueness of τx trivially follows from $a\top = b\top \Leftrightarrow a = b$.

8.2 Enabledness

The enabledness operator is defined in a similar way to termination, but using probabilistic guards, instead of probabilistic assertions.

The *enabledness operator* ϵ is a unary operator that maps a carrier-set element of a gpRAa to the unique probabilistic guard which satisfies

$$\epsilon x\bot = x\bot. \tag{57}$$

We will call a gpRAa extended with an enabledness operator a gpRAae, and a gpRAa with termination and enabledness operators gpRAaet.

In the model we define $\epsilon S \triangleq [\neg S.\mathsf{False}]$, where from each initial state σ, $\neg S.\mathsf{False}.\sigma$ describes the probability with which S will not behave miraculously.

Like the termination operator, the enabledness operator carries with it an implicit assumption about the carrier set: it requires the least element $\bot$ to right-annihilate non-miraculous behaviour. This describes an important property of our motivating program model.

From the enabledness definition follows a set of useful properties,

$$\epsilon g = g \tag{58}$$
$$g \sqsubseteq \epsilon(gx) \tag{59}$$
$$\epsilon(x\epsilon y) = \epsilon(xy) \tag{60}$$
$$\epsilon(x \sqcap y) = \epsilon x \sqcap \epsilon y \tag{61}$$
$$\epsilon(x \;{}_a\!\oplus\; y) = \epsilon x \;{}_a\!\oplus\; \epsilon y \tag{62}$$

where g is a probabilistic guard, a is a probabilistic assertion, and $x, y \in \mathsf{X}$. When ϵx is a (standard) assertion, we also have,

$$x = (\epsilon x)x \text{ and} \tag{63}$$
$$\epsilon x = x\bot + 1. \tag{64}$$

As for termination, excluding probabilistic choice property (62), properties (57-63) are used in the definition of the enabledness operator in dRA [14] (where g is always taken to be a standard guard in that algebra). Properties (59-61,63) are also the axioms of the domain operator in Kleene algebra with domain [1]. (Property (58) is derivable in these other algebras.)

Enabledness properties (59-62) and uniqueness of ϵx are no less trivial to verify than the termination properties given the guard property $g\bot = p\bot \Leftrightarrow g = p$ holds (Prop. 2).

Note that using the enabledness operator, total-correctness property $\tilde{p}\bot \sqsubseteq x\tilde{q}$ (44) may be written as $\tilde{p} \sqsubseteq \epsilon(x\tilde{q})$:

$$\begin{aligned}
&\quad \tilde{p}\bot \sqsubseteq x\tilde{q}\\
&\Leftrightarrow \tilde{p}\bot \sqsubseteq x\tilde{q}\bot\\
&\Leftrightarrow \{\text{enabledness property (57)}\}\\
&\quad \tilde{p}\bot \sqsubseteq \epsilon(x\tilde{q})\bot\\
&\Leftrightarrow \{\text{guard property (33)}\}\\
&\quad \tilde{p} \sqsubseteq \epsilon(x\tilde{q}).
\end{aligned}$$

Although the termination and enabledness axioms (49) and (57) hold in our motivating program model, it is possible that they may not hold for additional probabilistic program models, such as reactive (trace) models (e.g., [8]). For such models, alternative definitions using some of the other properties we have listed (e.g., (51-54) for termination, and (59-62) for enabledness) could be considered.

9 Observations

We have seen how many of the healthiness conditions of the one-bounded expectation transformers have been instrumental in the definition of the probabilistic refinement algebras. What of the continuity conditions or of subtraction?

As for the general refinement algebra (and the demonic refinement algebra) the lack of infinite meet ($\sqcap$) and join ($\sqcup$) operators mean that it is not possible to express the continuity conditions, which may be useful for reasoning about certain loop transformations. (For example, $xy \sqsubseteq zx \Rightarrow xy^\omega \sqsubseteq z^\omega x$ is dependent on a continuity condition on x [15]).

How does our inability to express subtraction, or to derive properties of the one-bounded expectation transformers which assume this property, affect the reasoning capabilities of the algebras? Let us consider how it affects the relationship between total and weak correctness assertions discussed in Sect. 7.

For the conjunctive predicate transformer x and guards p and q we have that total-correctness implies weak-correctness, i.e.,

$$(\top = px\tilde{q}) \Rightarrow (px = pxq) \tag{65}$$

holds and may be verified in gRA given a conjunctivity assumption on x. For the more general set of monotonic predicate transformers, this property fails to hold, and cannot be verified in gRA [16].

For the nondeterministic one-bounded expectation transformers, the relationship between total and weak correctness specified by property (65) also holds [9], although it (predictably) does not hold for the more general set of dually nondeterministic one-bounded transformers. Despite this, (65) may *not* be shown to hold in gpRAa (where q and p are standard guards), even when healthiness property $\oplus$-subdistributivity is assumed (take for instance $x = (s := 1) + (s := 2)$, $q = p = [s = 1]$). This reveals an underlying dependence of (65) on the healthiness condition subtraction.

This observation seems to suggest that the inclusion of a subtraction operator, or further axioms, may be useful for reasoning algebraically (in a point-free way) about probabilistic programs. We believe that a further investigation into these possibilities may be worthwhile.

10 Conclusions

The general refinement algebra (gRA) lives up to its name in that it is general enough to include models such as the monotonic predicate transformers [16] (which can be used to represent standard programs) and the monotonic one-bounded expectation transformers [9] (which can be used to represent probabilistic programs). This allows gRA to be used to reason about properties common to these models. However, its very generality means that there are some aspects of probabilistic programs that cannot be expressed in gRA. To address this issue a number of extensions of gRA have been investigated. In this paper we have focussed on including properties of probabilistic choice, and probabilistic assertions and guards. We have done so by presenting a sequence of extensions of gRA:

- gpRA, which adds the "+" operator and associated iteration operators,
- gpRAa, which adds probabilistic assertions and guards (including constant and standard subsets),
- pRAa , which restricts its elements to satisfy $\oplus$-subdistributivity,
- gpRAat, which adds a termination operator, τ,

– gpRAae, which adds an enabledness operator, ϵ, and
– gpRAaet, which combines the previous two algebras.

There are many subtly different ways in which probabilistic choice may be introduced into the general refinement algebra. When introducing probabilistic choice, we have taken the novel approach of decomposing it into the "+" operator and probabilistic assertions. The "+" operator has simpler algebraic properties than "$\oplus$" — compare the axioms in Definition 2 with the properties of "$\oplus$" (37-42). However, the "+" operator is more exotic, as is demonstrated by the program "$\mathsf{skip} + \mathsf{skip}$" in Sect. 2.3, which doesn't satisfy semi-sublinearity.

As well as being able to express the properties of "$\oplus$" (see Sect. 6), the addition of probabilistic assertions and guards allows one to express other interesting properties, like total-correctness assertions (Sect. 7) via

$$\tilde{p}\bot \sqsubseteq x\tilde{q},$$

and probabilistic termination and enabledness operators.

In adding the termination and enabledness operators we have relied upon the requirement that probabilistic assertions, a and b, and guards, g and p, must satisfy

$$a\top = b\top \Rightarrow a = b, \quad \text{and}$$
$$g\bot = p\bot \Rightarrow g = p.$$

This greatly simplifies the definitions of the termination and enabledness operators, and allows the axioms used elsewhere [14] to be derived as properties. This simplification has wider implications for the general refinement algebra with termination and enabledness operators.

Throughout the paper we have demonstrated how algebraic expressions of healthiness conditions for expectation transformers are used in the algebras. Of the healthiness properties listed in Fig. 3, we are able to express monotonicity and conjunctivity (which are expressible in gRA) as well as $\oplus$-distributivity, $\oplus$-subdistributivity, and sub-scaling. Future work is to investigate adding a subtraction operator, which would allow the expression of the "subtraction" property in Fig. 3 and hence also semi-sublinearity. Given the importance of the healthiness properties in defining the subset of expectation transformers that correspond to probabilistic programs, it seems justified to investigate an algebra in which these properties can be expressed. The extensions in this paper have taken us an important step in that direction, and in general we believe that we have made a number of important observations that could be useful for any axiomatisation of probabilistic choice in refinement algebra.

References

1. Desharnais, J., Möller, B., Struth, G.: Kleene algebra with domain. ACM Transactions on Computational Logic 7(4), 798–833 (2006)
2. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Englewood Cliffs (1976)

3. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer, Heidelberg (1990)
4. Hehner, E.C.R.: Probabilistic Predicative Programming. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 169–185. Springer, Heidelberg (2004)
5. Kozen, D.: Kleene algebra with tests. ACM Transactions on Programming Languages and Systems 19(3), 427–443 (1997)
6. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science. Springer, Heidelberg (2005)
7. Meinicke, L., Hayes, I.J.: Algebraic reasoning for probabilistic action systems and while-loops. DOI: 10.1007/s00236-008-0073-4. Accepted to Acta Informatica (March 2008)
8. Meinicke, L., Solin, K.: Reactive probabilistic programs and refinement algebra. In: Berghammer, R., Möller, B., Struth, G. (eds.) Relations and Kleene Algebra in Computer Science. LNCS, vol. 4988, pp. 304–319. Springer, Heidelberg (2008)
9. Meinicke, L., Solin, K.: Refinement algebra for probabilistic programs. Electron. Notes Theor. Comput. Sci. 201, 177–195 (2008)
10. Morgan, C., McIver, A.: Cost analysis of games using program logic. In: APSEC 2001: Proceedings of the Eighth Asia-Pacific on Software Engineering Conference, p. 351. IEEE Computer Society, Washington (2001)
11. Morgan, C., McIver, A.: Cost analysis of games using program logic (2001), `http://www.cse.unsw.edu.au/~carrollm/probs/bibliography.html`
12. Morgan, C., McIver, A., Seidel, K.: Probabilistic predicate transformers. ACM Transactions on Programming Languages and Systems 18(3), 325–353 (1996)
13. Solin, K.: On Two Dually Nondeterministic Refinement Algebras. In: Schmidt, R.A. (ed.) RelMiCS/AKA 2006. LNCS, vol. 4136, pp. 373–387. Springer, Heidelberg (2006)
14. Solin, K., von Wright, J.: Refinement Algebra with Operators for Enabledness and Termination. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 397–415. Springer, Heidelberg (2006)
15. von Wright, J.: From Kleene Algebra to Refinement Algebra. In: Boiten, E.A., Möller, B. (eds.) MPC 2002. LNCS, vol. 2386, pp. 233–262. Springer, Heidelberg (2002)
16. von Wright, J.: Towards a refinement algebra. Science of Computer Programming 51, 23–45 (2004)

Algebra of Programming Using Dependent Types

Shin-Cheng Mu[1], Hsiang-Shang Ko[2], and Patrik Jansson[3]

[1] Institute of Information Science, Academia Sinica, Taiwan
[2] Department of Computer Science and Information Engineering
National Taiwan University, Taiwan
[3] Department of Computer Science and Engineering
Chalmers University of Technology & University of Gothenburg, Sweden

Abstract. Dependent type theory is rich enough to express that a program satisfies an input/output relational specification, but it could be hard to construct the proof term. On the other hand, squiggolists know very well how to show that one relation is included in another by algebraic reasoning. We demonstrate how to encode functional and relational derivations in a dependently typed programming language. A program is coupled with an algebraic derivation from a specification, whose correctness is guaranteed by the type system.

1 Introduction

Program derivation is the technique of successively applying formal rules to a specification to obtain a program that is correct by construction. On the other hand, modern programming languages deploy expressive type systems to guarantee compiler-verifiable properties. There has been a trend to explore the expressiveness of dependent types, which opens a whole new world of type-level programming techniques. As Altenkirch et al. [1] put it, dependently typed programs are, "by their nature, proof carrying code." This paper aims to illustrate their comment by showing, in the dependently typed language Agda [17], that programs can actually carry their derivations.

As a teaser, Fig. 1 shows a derivation of a sorting algorithm in progress. The type of *sort-der* is a proposition that there exists a program of type $[\mathit{Val}] \rightarrow [\mathit{Val}]$ that is contained in $\mathit{ordered?} \circ \mathit{permute}$, a relation mapping a list to one of its ordered permutations. The proof proceeds by derivation from the specification towards the algorithm. The first step exploits monotonicity of $\circ$ and that *permute* can be expressed as a fold. The second step makes use of relational fold fusion. The shaded areas denote *interaction points* — fragments of (proof) code to be completed. The programmer can query Agda for the expected type and the context of the shaded expression. When the proof is completed, an algorithm *isort* is obtained by extracting the witness of the proposition. It is an executable program that is backed by the type system to meet the specification.

We have developed a library for functional and relational program derivation, with convenient notation for algebraic reasoning. Our work aims to be a

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 268–283, 2008.

```
sort-der : ∃ ([Val] → [Val]) (\f → ordered? ∘ permute ⊒ fun f)
sort-der = exists _ (      ordered? ∘ permute
                     ⊒⟨   (\vs → _-monotonic ordered? (permute-is-fold vs))   ⟩
                           ordered? ∘ foldR combine nil
                     ⊒⟨   {foldR-fusion ordered? ins-step ins-base}0    ⟩
                           {  }1 )

isort : [Val] → [Val]
isort = witness sort-der
```

Fig. 1. A derivation of a sorting algorithm in progress (see Sect. 4 for the details)

co-operation between the *squiggolists* and dependently-typed programmers that may benefit both sides. On the one hand, a number of tools for program transformation [11,22,24] have been developed but few of them have been put into much use. Being able to express derivation *within* the programming language encourages its use and serves as documentation. This paper is a case study of using the Curry-Howard isomorphism which the squiggolists may appreciate: specification of the program is expressed in their types, whose proofs (derivations) are given as programs and checked by the type system. On the other hand, it is known among dependently-typed programmers that the expressiveness of dependent types is far beyond proving that *reverse* preserves the length of its input. We can reason about the full input/output specification, for example, that *fast-reverse* is pointwise equal to the quadratic-time *reverse*, or that insertion sort implements a relational specification of sort. The reason this is rarely done is probably because it appears difficult to construct the proof terms. The method we propose is to develop the proof by algebraic reasoning within Agda.

In Sect. 2 we give an introduction to the part of Agda we use. We present our encoding of relations and their operations in Sect. 3, which prepares us to discuss our primary example in Sect. 4 and conclude with related work in Sect. 5.

2 A Crash Course on Agda

By "Agda" we mean Agda version 2, a dependently typed programming language evolved from the theorem prover having the same name. In this section we give a crash course on Agda, focusing on the aspects we need. For a detailed documentation, the reader is referred to Norell [17] and the Agda wiki [21].

Agda has a Haskell-like syntax extended with a number of additional features. Dependent function types are written $(x : A) \to B$ where B may refer to the identifier x, while non-dependent functions are written $A \to B$. The identity function, for example, can be defined by:

```
id : (A : Set) → A → A
id A a  =  a,
```

where *Set* is the kind of types. To apply *id* we should supply both the type and the value parameters, e.g., $\mathit{id}\,\mathbb{N}\,3$ where $\mathbb{N}$ is the type of natural numbers. Dependently typed programming would be very verbose if we always had to explicitly mention all the parameters. In cases when some parameters are inferable from the context, the programmer may leave them out, as in $\mathit{id}\ \_\, 3$.

For brevity, Agda supports implicit parameters. In the definition below:

$$\begin{aligned}
&\mathit{id} : \{A : \mathit{Set}\} \to A \to A\\
&\mathit{id}\,a \;=\; a,
\end{aligned}$$

the parameter $\{A : \mathit{Set}\}$ in curly brackets is implicit and need not be mentioned when *id* is called, e.g., $\mathit{id}\,3$. Agda tries to infer implicit parameters whenever possible. In case the inference fails, they can be explicitly provided in curly brackets: $\mathit{id}\,\{\mathbb{N}\}\,3$.

Named parameters in the type signature can be collected in a *telescope*. For example, $\{x : A\} \to \{y : A\} \to (z : B) \to \{w : C\} \to D$ can be abbreviated to $\{x\,y : A\}(z : B)\{w : C\} \to D$.

As an example of a datatype definition, cons-lists can be defined by:

$$\begin{aligned}
&\mathbf{data}\,[\_]\,(A : \mathit{Set}) : \mathit{Set}\,\mathbf{where}\\
&\qquad [\,] \quad\;\; : [A]\\
&\qquad \_{::}\_ \;\; : A \to [A] \to [A].
\end{aligned}$$

In Agda's notation for dist-fix definitions, an underline denotes a location for a parameter. The type constructor $[\_]$ takes a type and yields a type. The parameter $(A : \mathit{Set})$, written on the left-hand side of the colon, scopes over the entire definition and is an implicit parameter of the constructors $\_{::}\_$ and $[\,]$.

2.1 First-Order Logic

In the Curry-Howard isomorphism, types are propositions and terms their proofs. Being able to construct a term of a particular type is to provide a proof of that proposition. Fig. 2 shows an encoding of first-order intuitionistic logic in Agda. Falsity is represented by $\bot$, a type with no constructors and therefore no inhabitants. Truth, on the other hand, can be represented by the type $\top$, having one unique term — a record with no fields. Disjunction is represented by disjoint sum, while conjunction is denoted by product as usual: a proof of $P \uplus Q$ can be deducted either from a proof of P or a proof of Q, while a proof of $P \times Q$ consists of proofs of both.

An implication $P \to Q$ is represented as a function taking a proof of P to a proof of Q. We do not introduce new notation for it. The quantifier $\forall$ is encoded as a dependent function which, given any $x : A$, must produce a proof of $P\,x$. Agda provides a short hand $\mathit{forall}\,x \to P\,x$ in place of $(x : A) \to P\,x$ when A can be inferred. To prove the proposition $\exists\,A\,P$, where P is a predicate on terms of type A, one has to provide a witness $w : A$ and a proof of $P\,w$. Given a term of type $\exists\,A\,P$, the two functions *witness* and *proof* extract the witness and the proof, respectively.

```
data ⊥ : Set where

record ⊤ : Set where

data _⊎_ (P Q : Set) : Set where
  inj₁ : P → P ⊎ Q
  inj₂ : Q → P ⊎ Q

data _×_ (P Q : Set) : Set where
  _,_ : P → Q → P × Q
```

```
data ∃ (A : Set) (P : A → Set) : Set
 where
  exists : (w : A) → P w → ∃ A P

witness : {A : Set}{P : A → Set} →
          ∃ A P → A
witness (exists w p) = w

proof : {A : Set}{P : A → Set} →
        (x : ∃ A P) → P (witness x)
proof (exists w p) = p
```

Fig. 2. An encoding of first-order intuitionistic logic in Agda

2.2 Identity Type

A term of type $x \equiv y$ is a proof that the values x and y are equal. The datatype $\_\equiv\_$ is defined by:

```
data _≡_ {A : Set}(x : A) : A → Set where
    ≡-refl : x ≡ x.
```

Agda has relaxed lexical rules allowing Unicode characters in identifiers. Therefore, ≡-*refl* (without space) is a valid name. Since the only constructor ≡-*refl* is of type $x \equiv x$, being able to type-check a term with type $x \equiv y$ means that the type checker is able to deduce that x and y are indeed equal[1].

For the rest of the paper, we will exploit Unicode characters to give telling names to constructors, arguments, and lemmas. For example, if a variable is a proof of $y \equiv z$, we may name it *y≡z* (without space).

The type $\_\equiv\_$ is reflexive by definition. It is also symmetric and transitive, meaning that given a term of type $x \equiv y$, one can construct a term of type $y \equiv x$, and that given $x \equiv y$ and $y \equiv z$, one can construct $x \equiv z$:

```
≡-sym : {A : Set}{x y : A} →  x ≡ y  →  y ≡ x
≡-sym ≡-refl  =  ≡-refl,

≡-trans : {A : Set}{x y z : A} →  x ≡ y  →  y ≡ z  →  x ≡ z
≡-trans ≡-refl y≡z  =  y≡z.
```

The type of the first explicit parameter in the type signature of ≡-*sym* is $x \equiv y$, while the constructor ≡-*refl* in the pattern has type $x \equiv x$. When the type checker tries to unify them, x is unified with y. Therefore, when we need to return a term of type $y \equiv x$ on the right-hand side, we can simply return ≡-*refl*. The situation with ≡-*trans* is similar. Firstly, x is unified with y, therefore the parameter *y≡z*, having type $y \equiv z$, can also be seen as having type $x \equiv z$ and

[1] Agda assume uniqueness of identity proofs (but not proof irrelevance).

$$\begin{array}{ll}\textbf{infixr} & 2 \;\_{\sim}\langle\_\rangle\_\\ \textbf{infix} & 2 \;\_{\sim}\Box\end{array}$$

$$\begin{array}{l}\_{\sim}\langle\_\rangle\_ : \{A : \mathit{Set}\}(x : A)\{y\, z : A\} \to\; x \sim y \;\to\; y \sim z \;\to\; x \sim z\\ x \sim\langle\; x{\sim}y \;\rangle\; y{\sim}z \;=\; {\sim}\text{-}\mathit{trans}\; x{\sim}y\; y{\sim}z\end{array}$$

$$\begin{array}{l}\_{\sim}\Box : \{A : \mathit{Set}\}(x : A) \to\; x \sim x\\ x \sim\Box \;=\; {\sim}\text{-}\mathit{refl}\end{array}$$

Fig. 3. Combinators for preorder reasoning

be returned. In general, pattern matching and inductive families (such as $\_{\equiv}\_$) is a very powerful combination.

The interactive feature of Agda[2] is helpful for constructing the proof terms. One may, for example, leave out the right-hand side as an interaction point. Agda would prompt the programmer with the expected type of the term to fill in, which also corresponds to the remaining proof obligations. The list of variables in the current context and their types after unification are also available to the programmer.

The lemma $\equiv$-*subst* states that Leibniz equality holds: if $x \equiv y$, they are interchangeable in all contexts. Given a context f and a proof that $x \equiv y$, the congruence lemma $\equiv$-*cong* returns a proof that $f\,x \equiv f\,y$.

$$\begin{array}{l}\equiv\text{-}\mathit{subst} : \{A : \mathit{Set}\}(P : A \to \mathit{Set})\{x\, y : A\} \to\; x \equiv y \;\to P\,x \to P\,y\\ \equiv\text{-}\mathit{subst}\;P \equiv\text{-}\mathit{refl}\;Px \;=\; Px,\end{array}$$

$$\begin{array}{l}\equiv\text{-}\mathit{cong} : \{A\, B : \mathit{Set}\}(f : A \to B)\{x\, y : A\} \to\; x \equiv y \;\to\; f\,x \equiv f\,y\\ \equiv\text{-}\mathit{cong}\;f \equiv\text{-}\mathit{refl} \;=\; \equiv\text{-}\mathit{refl}.\end{array}$$

2.3 Preorder Reasoning

To prove a proposition $e_1 \equiv e_2$ is to construct a term having that type. One can do so by the operators defined in the previous section. It can be very tedious, however, when the expressions involved get complicated. Luckily, for any binary relation $\_{\sim}\_$ that is reflexive and transitive (that is, for which one can construct terms $\sim$-*refl* and $\sim$-*trans* having the types as described in the previous section), we can induce a set of combinators, shown in Fig. 3, which allows one to construct a term of type $e_1 \sim e_n$ in algebraic style. These combinators are implemented in Agda by Norell [17] and improved by Danielsson in the Standard Library of Agda [21]. Augustsson [3] has proposed a similar syntax for equality reasoning, with automatic inference of congruences.

To understand the definitions, notice that $\_{\sim}\langle\_\rangle\_$, a dist-fix function taking three explicit parameters, associates to the right. Therefore, the algebraic proof:

[2] Agda has an Emacs mode and a command line interpreter interface.

$$\begin{array}{ll}
 & e_1 \\
\sim\langle & reason_1 \quad \rangle \\
 & \quad\vdots \\
 & e_{n-1} \\
\sim\langle & reason_{n-1} \quad \rangle \\
 & e_n \\
\sim\Box &
\end{array}$$

should be bracketed as $e_1 \sim\langle reason_1 \rangle \ldots (e_{n-1} \sim\langle reason_{n-1} \rangle\, (e_n \sim\Box))$. Each occurrence of $\_\!\sim\!\langle\_\rangle\_$ takes three arguments: e_i on the left, $reason_i$ (a proof that $e_i \sim e_{i+1}$) in the angle brackets, and a proof of $e_{i+1} \sim e_n$ on the right-hand side, and produces a proof of $e_i \sim e_n$ using $\sim$-*trans*. As the base case, $\sim\Box$ takes the value e_n and returns a term of type $e_n \sim e_n$.

2.4 Functional Derivation

The ingredients we have prepared so far already allow us to perform some functional program derivation. For brevity, however, we introduce an equivalence relation on functions:

```
_≐_ : {A B : Set} → (A → B) → (A → B) → Set
f ≐ g  =  forall a → f a ≡ g a.
```

Since $\_\dot{=}\_$ can be shown to be reflexive and transitive, it also induces a set of pre-order reasoning operators. Fig. 4 shows a proof of the foldr fusion theorem. The steps using ≡-*refl* are simple equivalences which Agda can prove by expanding the definitions. The inductive hypothesis *ih* is established by a recursive call.

```
foldr-fusion : {A B C : Set} → (h : B → C) → {f : A → B → B} →
   {g : A → C → C} → {z : B} → (push : forall a → h · f a ≐ g a · h) →
        h · foldr f z ≐ foldr g (h z)
foldr-fusion h {f} {g} {z} _      []          =  ≡-refl
foldr-fusion h {f} {g} {z} push  (a :: as) =
   let ih  =  foldr-fusion h push as in
      h (foldr f z (a :: as))
   ≡⟨   ≡-refl   ⟩
      h (f a (foldr f z as))
   ≡⟨   push a (foldr f z as)   ⟩
      g a (h (foldr f z as))
   ≡⟨   ≡-cong (g a) ih   ⟩
      g a (foldr g (h z) as)
   ≡⟨   ≡-refl   ⟩
      foldr g (h z) (a :: as)
   ≡□
```

Fig. 4. Proving the fusion theorem for *foldr*

```
scanr-der : {A B : Set} → (f : A → B → B) → (e : B) →
      ∃ ([A] → List⁺ B) (\prog → map⁺ (foldr f e) · tails ≐ prog)
scanr-der f e = exists _(   map⁺ (foldr f e) · tails
                          ≐⟨  foldr-fusion (map⁺ (foldr f e)) (push-map-til f)   ⟩
                            foldr (sc f) [e]⁺
                          ≐□)
  where sc : {A B : Set} → (A → B → B) → A → List⁺ B → List⁺ B
        sc f a [b]⁺       = f a b ::⁺ [b]⁺
        sc f a (b ::⁺ bs) = f a b ::⁺ b ::⁺ bs
        push-map-til : {A B : Set} → (f : A → B → B) → {e : B} → (a : A) →
                map⁺ (foldr f e) · til a ≐ sc f a · map⁺ (foldr f e)
        push-map-til f a [xs]⁺         = ≡-refl
        push-map-til f a (xs ::⁺ xss) = ≡-refl
```

Fig. 5. Derivation of *scanr*. The constructors $\_{::}^{+}\_$ and $[\_]^{+}$ build non-empty lists, while $\mathit{tails} = \mathit{foldr}\ \mathit{til}\ [[]]^{+}$, where $\mathit{til}\ a\ [xs]^{+} = (a{::}xs){::}^{+}[xs]^{+}$; $\mathit{til}\ a\ (xs\ {::}^{+} xss) = (a{::}xs)\ {::}^{+} xs\ {::}^{+} xss$.

Agda ensures that proofs by induction are well-founded. Fig. 5 derives *scanr* from its specification $\mathit{map}^{+}\ (\mathit{foldr}\ f\ e) \cdot \mathit{tails}$, where map^{+} is the map function defined for List^{+}, the type of non-empty lists. The *foldr-fusion* theorem is used to transform the specification to a fold. The derived program can be extracted by *scanr* = *witness scanr-der*, while *scanr-pf* = *proof scanr-der* is a proof that can be used elsewhere. Notice that the first argument to *exists* is left implicit. Agda is able to infer the witness because it is syntactically presented in the derivation.

We have reproduced a complete derivation for the maximum segment sum problem. The derivation proceeds in the standard manner [6], transforming the specification to $\mathit{max} \cdot \mathit{map}\ (\mathit{foldr}\ \_\otimes\_\ 0) \cdot \mathit{tails}$ for some $\_\otimes\_$, and exploiting *scanr-pf* to convert it to a *scanr*. The main derivation is about 220 lines long, plus 400 lines of library code proving properties about lists and 100 lines for properties about integers. The code is available online [16].

The interactive interface of Agda proved to be very useful. One could progress the derivation line by line, leaving out the unfinished part as an interaction point. One may also type in the desired next step but leave the "reason" part blank, and let Agda derive the type of the lemma needed.

3 Relational Derivation

During the 90's there was a trend in the program derivation community to move from functions to relations. For completeness, we give a quick introduction to relations in this section. The reader is referred to Backhouse et al. [4] and Backhouse and Hoogendijk [5] for a more rigorous treatment. Bird and de Moor [8] present program derivation from a more abstract, category-theoretical point of view, with many illustrative examples of program derivation.

A relation R to B from A, denoted by $R : B \leftarrow A$, is usually understood as a subset of the set of pairs $A \times B$.[3] A function f is seen as a special case where $(a, b) \in f$ and $(a, b') \in f$ implies $b = b'$. The use of relations allows non-determinism in the specification. Derivation proceeds by inclusion as well as equality: in each step, the specification may be refined to a more deterministic subset, often all the way until we reach a function.

The composition of two relations $R : C \leftarrow B$ and $S : B \leftarrow A$ is defined by: $R \circ S \;=\; \{(a, c) \mid \exists b : (a, b) \in S \wedge (b, c) \in R\}$. Given a relation $R : B \leftarrow A$, its *power transpose* ΛR is a function from A to $\mathsf{P}B$ (subsets of B): $\Lambda R\, a = \{b \mid (a, b) \in R\}$, while the relation $\in : A \leftarrow \mathsf{P}A$ maps a set to one of its arbitrary members.

The fold remains an important construct in the relational setting. While *foldr* takes a step function of type $A \rightarrow B \rightarrow B$ and a base case of type B, its relational counterpart, which we denote by *foldR*, takes an uncurried relation $R : B \leftarrow (A \times B)$, while the base cases, being non-deterministic, are recorded in a set $s : \mathsf{P}B$.[4] The relational fold can be defined in terms of functional fold:

$$\begin{array}{l} \mathit{foldR}\; R\; s \;:\; B \leftarrow [A] \\ \mathit{foldR}\; R\; s = \in \circ\, \mathit{foldr}\; \Lambda(R \circ (\mathit{id} \times \in))\; s. \end{array}$$

We will see in the next few sections how these concepts can be modelled in Agda.

3.1 Modelling Relations

A possibly infinite subset (of A) could be represented by its membership function of type $\mathsf{P}A = A \rightarrow \mathit{Bool}$. With dependent types, we can also represent the membership judgement at type level:

$$\begin{array}{l} \mathsf{P} : \mathit{Set} \rightarrow \mathit{Set}1 \\ \mathsf{P}A = A \rightarrow \mathit{Set}. \end{array}$$

A set $s : \mathsf{P}A$ is a function mapping $a : A$ to a type, which encodes a logic formula determining its membership. Agda maintains a hierarchy of universes, where *Set* denotes the universe of types, *Set*1 denotes the universe of *Set* and all types declared as being in *Set*1, etc. Since $s : \mathsf{P}A$ is a function yielding a *Set*, $\mathsf{P}A$ is in the universe *Set*1. The function *singleton* creates singleton sets:

$$\begin{array}{l} \mathit{singleton} : \{A : \mathit{Set}\} \rightarrow A \rightarrow \mathsf{P}A \\ \mathit{singleton}\; a = \backslash a' \rightarrow\; a \equiv a'. \end{array}$$

Set union and inclusion, for example, are naturally encoded by disjunction and implication:

$$\begin{array}{ll} \_\cup\_ : \{A : \mathit{Set}\} \rightarrow \mathsf{P}A \rightarrow \mathsf{P}A \rightarrow \mathsf{P}A & \_\subseteq\_ : \{A : \mathit{Set}\} \rightarrow \mathsf{P}A \rightarrow \mathsf{P}A \rightarrow \mathit{Set} \\ r \cup s \;=\; \backslash\, a \rightarrow\; r\; a \uplus s\; a, & r \subseteq s \;=\; \mathit{forall}\; a \rightarrow\; r\; a \rightarrow s\; a. \end{array}$$

[3] Notations used in the beginning of this section, for example ×, ∈, and set comprehension, refer to their usual set-theoretical definitions. We will talk about how they can be represented in Agda in the next few subsections.

[4] Isomorphically, the base case can be represented by a relation $B \leftarrow \top$.

A term of type $r \subseteq s$ is a function which, given an a and a proof that a is in r, produces a proof that a is in s.

A relation $B \leftarrow A$, seen as a set of pairs, could be represented as $\mathsf{P}(A \times B) = (A \times B) \rightarrow \mathit{Set}$. However, we find the following "curried" representation more convenient:

$$\begin{aligned} \_\!\leftarrow\!\_ &: \ \mathit{Set} \rightarrow \mathit{Set} \rightarrow \mathit{Set}1 \\ B \leftarrow A &= \ A \rightarrow B \rightarrow \mathit{Set}. \end{aligned}$$

One of the advantages is that relations and set-valued functions are unified. The Λ operator, for example, is simply the identity function at the term-level:

$$\begin{aligned} &\Lambda \quad : \{A\, B : \mathit{Set}\} \rightarrow (B \leftarrow A) \rightarrow (A \rightarrow \mathsf{P}B) \\ &\Lambda\, R = R. \end{aligned}$$

A function can be converted to a relation:

$$\begin{aligned} &\mathit{fun} : \{A\, B : \mathit{Set}\} \rightarrow (A \rightarrow B) \rightarrow (B \leftarrow A) \\ &\mathit{fun}\, f\ a\ b \ = \ f\ a \equiv b. \end{aligned}$$

The identity relation, for example, is denoted $\mathit{id}_R : \{A : \mathit{Set}\} \rightarrow (A \leftarrow A)$ and defined by $\mathit{id}_R = \mathit{fun}\ \mathit{id}$.

Relational composition could be defined by $R \circ S = \exists\, B\, (\backslash b \rightarrow (S\ a\ b \times R\ b\ c))$. For reasons that will be clear in the next section, we split the definition into two parts, shown in Fig. 6. The operator $\_\bullet\_$ applies a relation $R : B \leftarrow A$ to a set $\mathsf{P}A$, yielding another set $\mathsf{P}B$. Composition $\_\circ\_$ is then defined using $\_\bullet\_$.

Complication arises when we try to represent $\in$. Recall that $\in$ maps $\mathsf{P}A$ to A. However, the second argument to $\_\!\leftarrow\!\_$ must be in Set, while $\mathsf{P}A$ is in $\mathit{Set}1$! At present, we have no choice but to declare another type of arrows that accepts $\mathit{Set}1$-sorted inputs:

$$\begin{aligned} \_\!\leftarrow_1\!\_ &: \ \mathit{Set} \rightarrow \mathit{Set}1 \rightarrow \mathit{Set}1 \\ B \leftarrow_1 \mathit{PA} &= \ \mathit{PA} \rightarrow B \rightarrow \mathit{Set}. \end{aligned}$$

It means we need several alternatives of relational composition that differ only in their types. Fig. 6 shows $\_\circ_1\_$ and $\_{}_1\!\circ\_$ for example. Such inconvenience may be resolved if Agda introduces *universe polymorphism*, a feature on the wish-list at the time of writing. Also summarised in Fig. 6 are $\_^{\smile}$ for relational converse, and $\_\times_1\_$, a higher-kinded variation of the product functor.

3.2 Inclusion and Monotonicity

A relation S can be refined to R if every possible outcome of R is a legitimate outcome of S. We represent the refinement relation by:

$$\begin{aligned} &\_\sqsubseteq\_ : \{A\, B : \mathit{Set}\} \rightarrow (B \leftarrow A) \rightarrow (B \leftarrow A) \rightarrow \mathit{Set} \\ &R \sqsubseteq S = \mathit{forall}\ a \rightarrow R\ a \subseteq S\ a, \end{aligned}$$

which expands to $\mathit{forall}\ a \rightarrow \mathit{forall}\ b \rightarrow R\ a\ b \rightarrow S\ a\ b$. Conversely, $R \sqsupseteq S = S \sqsubseteq R$. Both $\sqsubseteq$ and $\sqsupseteq$ can be shown to be reflexive and transitive. Therefore, we can use them for preorder reasoning.

$$\begin{array}{ll}
\_^{\smile} : \{A\,B : \mathit{Set}\} \to (A \leftarrow B) \to (B \leftarrow A) & \in\ : \{A : \mathit{Set}\} \to (A \leftarrow_1 \mathsf{P}A) \\
R^{\smile} = \backslash\, a\, b\ \to\ R\, b\, a & \in\ =\ \backslash pa\ a \to pa\ a
\end{array}$$

$$\begin{array}{l}
\_\centerdot\_ : \{A\,B : \mathit{Set}\} \to (B \leftarrow A) \to \mathsf{P}A \to \mathsf{P}B \\
R \centerdot s = \backslash\, b\ \to\ \exists\, A\, (\backslash a \to (s\ a \times R\ a\ b))
\end{array}$$

$$\begin{array}{l}
\_\circ\_ : \{A\,B\,C : \mathit{Set}\} \to (C \leftarrow B) \to (B \leftarrow A) \to (C \leftarrow A) \\
(R \circ S)\, a\ =\ R \centerdot (S\ a)
\end{array}$$

$$\begin{array}{l}
\_\circ_1\_ : \{A : \mathit{Set1}\}\{B\ C : \mathit{Set}\} \to (C \leftarrow B) \to (B \leftarrow_1 A) \to (C \leftarrow A) \\
(R \circ_1 S)\, a\ =\ R \centerdot (S\ a)
\end{array}$$

$$\begin{array}{l}
\_{}_1\!\circ\_ : \{A\,B\,C : \mathit{Set}\} \to (C \leftarrow_1 \mathsf{P}B) \to (B \leftarrow A) \to (C \leftarrow A) \\
(R\, {}_1\!\circ S)\, a\ =\ R\, (S\ a)
\end{array}$$

$$\begin{array}{l}
\_\times_1\_ : \{A\,B : \mathit{Set}\}\{PC : \mathit{Set1}\}\{D : \mathit{Set}\} \to \\
\qquad\qquad (B \leftarrow A) \to (D \leftarrow_1 PC) \to ((B \times D) \leftarrow_1 (A \times_1 PC)) \\
(R \times_1 S)\, (a ,_1 pc)\, (b, d)\ =\ R\ a\ b \times S\ pc\ d
\end{array}$$

Fig. 6. Some operators on $\_\leftarrow\_$ and $\_\leftarrow_1\_$ relations, including composition, membership and product. In this paper, $\_\times_1\_$ is overloaded for the type of pairs whose right component is in *Set*1 ($\_,_1\_$ being the data constructor), and its functor action on relations (defined in this figure).

In hand-written derivation, the monotonicity of $\_\circ\_$ (that is, $S \sqsubseteq T$ implies $R \circ S \sqsubseteq R \circ T$) is often used without being explicitly stated. In our Agda encoding where there are many versions of composition, it appears that we need one monotonicity lemma for each of them. Luckily, since those alternatives of composition are all defined in terms of $\_\centerdot\_$, it is enough to model monotonicity for $\_\centerdot\_$ only:

$$\begin{array}{l}
\centerdot\text{-}\mathit{monotonic} : \{A\,B : \mathit{Set}\} \to (R : B \leftarrow A) \to \{s\ t : \mathsf{P}A\} \to \\
\qquad\qquad\qquad\qquad s \subseteq t \to R \centerdot s \subseteq R \centerdot t \\
\centerdot\text{-}\mathit{monotonic}\ R\ s{\subseteq}t\ b\ (\mathit{exists}\ a_1\ (a_1{\in}s, bRa_1)) = \\
\qquad \mathit{exists}\ a_1\ (s{\subseteq}t\ a_1\ a_1{\in}s, bRa_1).
\end{array}$$

To refine $R \circ S \circ T$ to $R \circ U \circ T$ given $U \sqsubseteq S$, for example, we may use $(\backslash x \to \centerdot\text{-}\mathit{monotonic}\, R\, (U{\sqsubseteq}S\, (T \centerdot x)))$ as the reason. It is instructive to study the definition of $\centerdot$-*monotonic*. After taking R and $s{\subseteq}t$ (a proof of $s \subseteq t$), the function $\centerdot$-*monotonic* shall return a proof of $R \centerdot s \subseteq R \centerdot t$. The proof, given a value b and a proof that some a_1 in s is mapped to b through R, shall produce a proof that there exists some value in t that is also mapped to b. The obvious choice of such a value is a_1. Notice how we apply $s{\subseteq}t$ to a_1 and $a_1{\in}s$ to produce a proof that a_1 is also in t.

Another lemma often used without being said is that we can introduce id_R anywhere we need. It can be proved using $\equiv$-*subst*:

$$\begin{aligned}&\mathit{id\text{-}intro} : \{A\,B : \mathit{Set}\}\{R : B \leftarrow A\} \rightarrow R \sqsupseteq R \circ \mathit{id}_R\\ &\mathit{id\text{-}intro}\,\{\_\}\{\_\}\{R\}\;a\;b\;(\mathit{exists}\;a'\;(a{\equiv}a', bRa')) \;=\\ &\qquad \equiv\text{-}\mathit{subst}\;(\backslash a \rightarrow R\;a\;b)\;(\equiv\text{-}\mathit{sym}\;a{\equiv}a')\;bRa'.\end{aligned}$$

3.3 Relational Fold

Having defined all the necessary components, we can now define relational fold in terms of functional fold:

$$\begin{aligned}&\mathit{foldR} : \{A\,B : \mathit{Set}\} \rightarrow (B \leftarrow (A \times B)) \rightarrow \mathsf{P}B \rightarrow (B \leftarrow [A])\\ &\mathit{foldR}\;R\;s \;=\; \mathit{foldr}\;(R \circ_1 (\mathit{id}_R \times_1 \in))\;s.\end{aligned}$$

On the top of the list of properties that we wish to have proved is, of course, fold fusion for relational folds:

$$\begin{aligned}&\mathit{foldR\text{-}fusion} : \{A\,B\,C : \mathit{Set}\} \rightarrow (R : C \leftarrow B) \rightarrow \{S : B \leftarrow (A \times B)\} \rightarrow\\ &\qquad \{T : C \leftarrow (A \times C)\}\{u : \mathsf{P}B\}\{v : \mathsf{P}C\} \rightarrow\\ &\qquad\qquad R \circ S \sqsupseteq T \circ (\mathit{id}_R \times R) \;\rightarrow\; R \centerdot u \supseteq v \rightarrow\\ &\qquad\qquad\qquad R \circ \mathit{foldR}\;S\;u \sqsupseteq \mathit{foldR}\;T\;v.\end{aligned}$$

The proof proceeds by converting both sides to functional folds. It is omitted here for brevity but is available online [16]. To use fold fusion, however, there has to be a fold to start with. Luckily, this is hardly a problem, given the following lemma showing that id_R, when instantiated to lists, is a fold:

$$\mathit{id}_R{\sqsupseteq}\mathit{foldR} : \{A : \mathit{Set}\} \rightarrow \mathit{id}_R\,\{[A]\} \sqsupseteq \mathit{foldR}\;\mathit{cons}\;\mathit{nil},$$

where $\mathit{cons} = \mathit{fun}\,(\mathit{uncurry}\,\_{::}\_)$ and $\mathit{nil} = \mathit{singleton}\,[\,]$. Let us try to construct its proof term. The inclusion $\mathit{id}_R\,\{[A]\} \sqsupseteq \mathit{foldR}\;\mathit{cons}\;\mathit{nil}$ expands to:

$$\mathit{forall}\;\mathit{xs}\;\mathit{ys} \rightarrow\; \mathit{foldR}\;\mathit{cons}\;\mathit{nil}\;\mathit{xs}\;\mathit{ys} \;\rightarrow\; \mathit{xs} \equiv \mathit{ys}.$$

The proof term of $\mathit{id}_R{\sqsupseteq}\mathit{foldR}$ should be a function which takes xs, ys, and a proof that $\mathit{foldR}\;\mathit{cons}\;\mathit{nil}$ maps xs to ys, and returns a proof of $\mathit{xs} \equiv \mathit{ys}$. When xs is $[\,]$, $\mathit{foldR}\;\mathit{cons}\;\mathit{nil}\,[\,]\;\mathit{ys}$ simplifies to $[\,] \equiv \mathit{ys}$, and we can simply return the proof:

$$\mathit{id}_R{\sqsupseteq}\mathit{foldR}\;[\,]\;\mathit{ys}\;[]{\equiv}\mathit{ys} \;=\; []{\equiv}\mathit{ys}.$$

Consider the case $a :: \mathit{xs}$. The proposition $\mathit{foldR}\;\mathit{cons}\;\mathit{nil}\,(a :: \mathit{xs})\;\mathit{ys}$ expands to $\exists\,(V{\times}[V])\,P$, where $P\,(a', \mathit{as}) = ((a \equiv a'){\times}(\mathit{foldR}\;\mathit{cons}\;\mathit{nil}\;\mathit{xs}\;\mathit{as})){\times}(\mathit{cons}\,(a, \mathit{as})\;\mathit{ys})$. Given $a :: \mathit{xs}$, ys, and a proof of $\exists\,(V \times [V])\,P$, we should construct a proof that $a :: \mathit{xs} \equiv \mathit{ys}$. We can do so by equational reasoning:

$$\begin{aligned}&\mathit{id}_R{\sqsupseteq}\mathit{foldR}\;(a :: \mathit{xs})\;\mathit{ys}\;(\mathit{exists}\,(a', \mathit{as})\,((a{\equiv}a', \mathit{foldRxsas}), a'{::}\mathit{as}{\equiv}\mathit{ys})) =\\ &\qquad\quad a :: \mathit{xs}\\ &\qquad \equiv\langle\quad a{\equiv}a'\;\langle{::}\rangle\;(\mathit{id}_R{\sqsupseteq}\mathit{foldR}\;\mathit{xs}\;\mathit{as}\;\mathit{foldRxsas})\quad\rangle\\ &\qquad\quad a' :: \mathit{as}\\ &\qquad \equiv\langle\quad a'{::}\mathit{as}{\equiv}\mathit{ys}\quad\rangle\\ &\qquad\quad \mathit{ys}\\ &\qquad \equiv\square,\end{aligned}$$

where $\langle{::}\rangle$ is $\equiv$-cong applied twice, substituting a for a' and xs for as.

4 Example: Deriving Insertion Sort

We are finally in a position to present our main example: a derivation of insertion sort, adopted from Bird [7].

4.1 Specifying Sort

We first specify what a sorted list is, assuming a datatype Val and a binary ordering $\_\leq\_ : Val \to Val \to Set$ that form a decidable total order. To begin with, let $lbound$ be the set of all pairs (a, xs) such that a is a lower bound of xs:

$$\begin{array}{l} lbound : \mathsf{P}(Val \times [Val]) \\ lbound\,(a, [\,]) \qquad = \top \\ lbound\,(a, b :: xs) = (a \leq b) \times lbound\,(a, xs). \end{array}$$

A *coreflexive* is a sub-relation of id_R. The following operator $\_?$ converts a set to a coreflexive, letting the input go through iff it is in the set:

$$\begin{array}{l} \_? : \{A : Set\} \to \mathsf{P}A \to (A \leftarrow A) \\ (p\,?)\; a\; b \;=\; (a \equiv b) \times p\; a. \end{array}$$

The coreflexive $ordered?$, which lets a list go through iff it is sorted, can then be defined as a fold:

$$\begin{array}{l} ordered? \;:\; [Val] \leftarrow [Val] \\ ordered? = foldR\,(cons \circ lbound\,?)\; nil. \end{array}$$

We postulate a datatype Bag, representing bags of values. Bags are formed by two constructors: $\Lbag\Rbag : Bag$ and $\_::_{\mathsf{b}}\_ : Val \to Bag \to Bag$. For the derivation to work, we demand that the result of $\_::_{\mathsf{b}}\_$ be distinguishable from the empty bag, and that $\_::_{\mathsf{b}}\_$ be commutative:[5]

$$\begin{array}{l} ::_{\mathsf{b}}\text{-}nonempty : forall\,\{a\; w\} \to (\Lbag\Rbag \equiv a ::_{\mathsf{b}} w) \to \bot \\ ::_{\mathsf{b}}\text{-}commute \;\; : (a\; b \,:\, Val) \to (w : Bag) \to a ::_{\mathsf{b}} (b ::_{\mathsf{b}} w) \equiv b ::_{\mathsf{b}} (a ::_{\mathsf{b}} w). \end{array}$$

The function $bagify$, defined below, converts a list to a bag by a fold:

$$\begin{array}{l} bagify \;:\; [Val] \to Bag \\ bagify = foldr\; \_::_{\mathsf{b}}\_\; \Lbag\Rbag. \end{array}$$

To map a list to one of its arbitrary permutations, we simply convert it to a bag, and convert the bag back to a list! To sort a list is to find one of its permutations that is sorted:

$$\begin{array}{l} permute \;:\; [Val] \leftarrow [Val] \\ permute = (fun\; bagify)^{\smile} \circ fun\; bagify, \\ \\ sort \qquad :\; [Val] \leftarrow [Val] \\ sort \qquad = ordered? \circ permute. \end{array}$$

Thus completes the specification, from which we shall derive an algorithm that actually sorts a list.

[5] We can put more constraints on bags, such as that $\_::_{\mathsf{b}}\_$ discards no elements. But the two properties are enough to guarantee that $isort$ is included in $ordered? \circ permute$.

4.2 The Derivation

The derivation begins with observing that *permute* can be turned into a fold. We first introduce an id_R by *id-intro*, followed by the lemma $id_R \sqsupseteq foldR$, and fold fusion:

$$\begin{array}{lll}
\mathit{perm\text{-}der} : \exists_1\,([\,Val] \leftarrow [\,Val])\,(\backslash \mathit{perm} \rightarrow \mathit{permute} \sqsupseteq \mathit{perm}) & & \\
\mathit{perm\text{-}der} = \mathit{exists}_1\,\_\,(& & \mathit{permute} \\
 & \sqsupseteq\langle & \mathit{id\text{-}intro} \quad \rangle \\
 & & \mathit{permute} \circ id_R \\
 & \sqsupseteq\langle & (\backslash xs \rightarrow \bullet\text{-}\mathit{monotonic}\; \mathit{permute}\; (id_R \sqsupseteq \mathit{foldR}\; xs)) \quad \rangle \\
 & & \mathit{permute} \circ \mathit{foldR}\; \mathit{cons}\; \mathit{nil} \\
 & \sqsupseteq\langle & \mathit{foldR\text{-}fusion}\; \mathit{permute}\; \mathit{perm\text{-}step}\; \mathit{perm\text{-}base} \quad \rangle \\
 & & \mathit{foldR}\; \mathit{combine}\; \mathit{nil} \\
 & \sqsupseteq\Box), &
\end{array}$$

where $\exists_1$ is a *Set*1 variant of $\exists$, with extraction functions $\mathit{witness}_1$ and proof_1. The relation *combine* can be defined as follows:

$$\begin{array}{ll}
\mathit{combine} : [\,Val] \leftarrow (\,Val \times [\,Val]) & \\
\mathit{combine}\,(a, xs) & = \mathit{cons}\,(a, xs) \cup \mathit{combine}'\,(a, xs), \\
\mathit{combine}' : [\,Val] \leftarrow (\,Val \times [\,Val]) & \\
\mathit{combine}'\,(a, [\,]) & = \backslash ys \rightarrow \bot \\
\mathit{combine}'\,(a, b :: xs) & = (\backslash zs \rightarrow \mathit{cons}\,(b, zs)) \bullet \mathit{combine}\,(a, xs).
\end{array}$$

Given (a, xs), it inserts a into an arbitrary position of xs. For the *foldR-fusion* to work, we have to provide two proofs:

$$\begin{array}{l}
\mathit{perm\text{-}step} : \mathit{permute} \circ \mathit{cons} \sqsupseteq \mathit{combine} \circ (id_R \times \mathit{permute}) \\
\mathit{perm\text{-}base} : \mathit{permute} \bullet \mathit{nil} \supseteq \mathit{nil}.
\end{array}$$

But the real work is done in proving that shuffling the input list does not change the result of *bagify*:

$$\begin{array}{l}
\mathit{bagify\text{-}homo} : (a : \,Val) \rightarrow (xs\; ys : [\,Val]) \rightarrow \\
\qquad\qquad \mathit{combine}\,(a, xs)\; ys \rightarrow \mathit{bagify}\,(a :: xs) \equiv \mathit{bagify}\; ys.
\end{array}$$

It is when proving this lemma that we need $::_{\mathsf{b}}$ -*commute*.

After the reasoning above, we have at our hands:

$$\begin{array}{l}
\mathit{perm} \;:\; [\,Val] \leftarrow [\,Val] \\
\mathit{perm} = \mathit{witness}_1\; \mathit{perm\text{-}der}, \\
\\
\mathit{permute\text{-}is\text{-}fold} \;:\; \mathit{permute} \sqsupseteq \mathit{perm} \\
\mathit{permute\text{-}is\text{-}fold} = \mathit{proof}_1\; \mathit{perm\text{-}der}.
\end{array}$$

Therefore, $\mathit{perm} = \mathit{foldR}\; \mathit{combine}\; \mathit{nil}$, while *permute-is-fold* is a proof that *perm* refines *permute*.

Now that *permute* can be refined to a fold, a natural step to try is to fuse *ordered*? into the fold. We derive:

$$\begin{array}{lll}
\mathit{sort\text{-}der} : \exists\,([\,\mathit{Val}] \to [\,\mathit{Val}])\,(\backslash f \to \mathit{ordered?} \circ \mathit{permute} \sqsupseteq \mathit{fun}\,f) & & \\
\mathit{sort\text{-}der} = \mathit{exists}\ \_ & & \\
\qquad\qquad\qquad (& \mathit{ordered?} \circ \mathit{permute} & \\
\qquad\qquad\qquad\ \sqsupseteq\langle & (\backslash xs \to \bullet\text{-}\mathit{monotonic}\ \mathit{ordered?}\,(\mathit{permute\text{-}is\text{-}fold}\ xs)) & \rangle \\
 & \mathit{ordered?} \circ \mathit{perm} & \\
\qquad\qquad\qquad\ \sqsupseteq\langle & \sqsupseteq\text{-}\mathit{refl} & \rangle \\
 & \mathit{ordered?} \circ \mathit{foldR}\ \mathit{combine}\ \mathit{nil} & \\
\qquad\qquad\qquad\ \sqsupseteq\langle & \mathit{foldR\text{-}fusion}\ \mathit{ordered?}\ \mathit{ins\text{-}step}\ \mathit{ins\text{-}base} & \rangle \\
 & \mathit{foldR}\,(\mathit{fun}\,(\mathit{uncurry}\ \mathit{insert}))\ \mathit{nil} & \\
\qquad\qquad\qquad\ \sqsupseteq\langle & \mathit{foldR\text{-}to\text{-}foldr}\ \mathit{insert}\,[\,] & \rangle \\
 & \mathit{fun}\,(\mathit{foldr}\ \mathit{insert}\,[\,]) & \\
\qquad\qquad\qquad\ \sqsupseteq\Box). & &
\end{array}$$

The function *insert* follows the usual definition:

$$\begin{array}{llcl}
\mathit{insert} : \mathit{Val} \to [\,\mathit{Val}] \to [\,\mathit{Val}] & & & \\
\mathit{insert}\ a\,[\,] = a :: [\,] & & & \\
\mathit{insert}\ a\,(b :: xs)\ \mathbf{with}\ a \le ?\, b & & & \\
\ldots & \mid\ \mathit{yes}\ a{\le}b & = & a :: b :: xs \\
\ldots & \mid\ \mathit{no}\ \ a{\nleq}b & = & b :: \mathit{insert}\ a\ xs,
\end{array}$$

where $a\ \le ?\, b$ determines whether $a \le b$, whose result is case-matched by the **with** notation. The fusion conditions are:

$$\begin{array}{l}
\mathit{ins\text{-}step} : \mathit{ordered?} \circ \mathit{combine} \sqsupseteq \mathit{fun}\,(\mathit{uncurry}\ \mathit{insert}) \circ (\mathit{id}_R \times \mathit{ordered?}) \\
\mathit{ins\text{-}base} : \mathit{ordered?} \bullet \mathit{nil} \supseteq \mathit{nil}.
\end{array}$$

Finally, *foldR-to-foldr* is a small lemma allowing us to convert a relational fold to a functional fold, provided that its arguments have been refined to a function and a singleton set already:

$$\begin{array}{l}
\mathit{foldR\text{-}to\text{-}foldr} : \{A\ B : \mathit{Set}\} \to (f : A \to B \to B) \to (e : B) \to \\
\qquad\qquad \mathit{foldR}\,(\mathit{fun}\,(\mathit{uncurry}\,f))\,(\mathit{singleton}\ e) \sqsupseteq \mathit{fun}\,(\mathit{foldr}\,f\ e).
\end{array}$$

We have thus derived $\mathit{isort} = \mathit{witness}\ \mathit{sort\text{-}der} = \mathit{foldr}\ \mathit{insert}\,[\,]$, while at the same time proved that it meets the specification $\mathit{ordered?} \circ \mathit{permute}$. The details of the proofs are available online [16]. The library code defining sets, relations, folds, and their properties, amounts to about 800 lines. The main derivation of *isort* is not long. Proving the fusion condition *ins-step* and its related properties turned out to take some hard work and eventually adds up to about 700 lines of code. The interactive mode was of great help — the proof would have been difficult to construct by hand.

5 Conclusion and Related Work

We have shown how to encode relational program derivation in a dependently typed language. Derivation is carried out in the host language, the correctness being guaranteed by the type system. It also demonstrates that dependent types are expressive enough to demand that a program satisfies an input/output relation. An interesting way to construct the corresponding proof term, which would be difficult to build otherwise, is derivation.

McKinna and Burstall's paper on "Deliverables" [15] is an early example of machine checked program + proof construction (using Pollack's LEGO). In their terminology *sort-der* would be a deliverable — an element of a dependent Σ-type pairing up a function and a proof of correctness. In the Coq tradition Program Extraction has been used already from Paulin-Mohring's early paper [18] to the impressive four-colour theorem development (including the development of a verified compiler). Our contribution is more modest — we aim at formally checked but still readable Algebra-of-Programming style derivations.

The concept of Inductive Families [12], especially the identity type ($\_\equiv\_$), is central to the Agda system and to our derivations. A recent development of relations in dependent type theory was carried out by Gonzalía [13, Ch. 5]. The advances in Agda's notation and support for hidden arguments between that derivation and our work is striking.

There has been a trend in recent years to bridge the gap between dependent types and practical programming. Projects along this line include Cayenne [2], Coq [10], Dependent ML [23], Agda [17], Ωmega [19], Epigram [14], and the GADT extension [9] to Haskell. It is believed that dependent types have an important role in the next generation of programming languages [20].

Acknowledgements. We are grateful to Nils Anders Danielsson for pointing out typos and giving valuable suggestions regarding the presentation.

References

1. Altenkirch, T., McBride, C., McKinna, J.: Why dependent types matter. Draft (2005)
2. Augustsson, L.: Cayenne – a language with dependent types. In: ICFP 1998, pp. 239–250 (1998)
3. Augustsson, L.: Equality proofs in Cayenne. Chalmers Univ. of Tech. (1999)
4. Backhouse, R.C., et al.: Relational catamorphisms. In: IFIP TC2/WG2.1 Working Conference on Constructing Programs, pp. 287–318. Elsevier, Amsterdam (1991)
5. Backhouse, R.C., Hoogendijk, P.F.: Elements of a relational theory of datatypes. In: Möller, B., Schuman, S., Partsch, H. (eds.) Formal Program Development. LNCS, vol. 755, pp. 7–42. Springer, Heidelberg (1993)
6. Bird, R.S.: Algebraic identities for program calculation. Computer Journal 32(2), 122–126 (1989)
7. Bird, R.S.: Functional algorithm design. Science of Computer Programming 26, 15–31 (1996)
8. Bird, R.S., de Moor, O.: Algebra of Programming. International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1997)
9. Cheney, J., Hinze, R.: First-class phantom types. Technical Report TR2003-1901, Cornell University (2003)

10. The Coq Development Team, LogiCal Project. The Coq Proof Assistant Reference Manual (2006)
11. de Moor, O., Sittampalam, G.: Higher-order matching for program transformation. Theoretical Computer Science 269(1-2), 135–162 (2001)
12. Dybjer, P.: Inductive families. Formal Aspects of Computing, 440–465 (1994)
13. Gonzalía, C.: Relations in Dependent Type Theory. PhD thesis, Chalmers Univ. of Tech. (2006)
14. McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming 14(1), 69–111 (2004)
15. McKinna, J., Burstall, R.M.: Deliverables: A categorial approach to program development in type theory. In: Borzyszkowski, A.M., Sokolowski, S. (eds.) MFCS 1993. LNCS, vol. 711, pp. 32–67. Springer, Heidelberg (1993)
16. Mu, S.-C., Ko, H.-S., Jansson, P.: AoPA: Algebra of programming in Agda, `http://www.iis.sinica.edu.tw/~scm/2008/aopa/`
17. Norell, U.: Towards a Practical Programming Language Based on Dependent Type Theory. PhD thesis, Chalmers Univ. of Tech. (2007)
18. Paulin-Mohring, C.: Extracting F_ω's programs from proofs in the Calculus of Constructions. In: POPL 1989, Austin. ACM Press, New York (1989)
19. Sheard, T.: Programming in *Ω*mega. The 2nd Central European Functional Programming School (June 2007)
20. Sweeney, T.: The next mainstream programming language: a game developer's perspective. In: POPL 2006 (January 2006) (invited talk)
21. The Agda Team. The Agda Wiki (2007), `http://www.cs.chalmers.se/~ulfn/Agda/`
22. Verhoeven, R., Backhouse, R.C.: Towards tool support for program verification and construction. In: World Congress on Formal Methods, pp. 1128–1146 (1999)
23. Xi, H.: Dependent ML: an approach to practical programming with dependent types. Journal of Functional Programming 17(2), 215–286 (2007)
24. Yokoyama, T., Hu, Z., Takeichi, M.: Yicho - a system for programming program calculations. In: The 3rd Asian Workshop on Programming Languages and Systems (APLAS 2002), pp. 366–382 (2002)

Safe Modification of Pointer Programs in Refinement Calculus

Susumu Nishimura

Dept. of Mathematics, Graduate School of Science, Kyoto University
Sakyo-ku, Kyoto 606-8502, Japan
susumu@math.kyoto-u.ac.jp

Abstract. This paper discusses stepwise refinement of pointer programs in the framework of refinement calculus. We augment the underlying logic with formulas of separation logic and then introduce a pair of new predicate transformers, called *separating assertion* and *separating assumption*. The new predicate transformers are derived from separating conjunction and separating implication, which are fundamental logical connectives in separation logic. They represent primitive forms of heap allocation/deallocation operators and the basic pointer statements can be specified by means of them. We derive several refinement laws that are useful for stepwise refinement and demonstrate the use of the laws in the context of correctness preserving transformations that are intended for improved memory usage.

The formal development is carried out in the framework of higher-order logic and is based on Back and Preoteasa's axiomatization of state space and its extension to the heap storage [BP05, Pre06]. All the results have been implemented and verified in the theorem prover PVS.

1 Introduction

Pointers are a powerful tool that provides clean and efficient solutions to certain programming tasks. However, pointers are also notoriously hard to handle because of the problems caused by their effectful nature, e.g., pointer aliasing and dangling pointers. Thus correct implementation of pointers is a challenging issue in any stage of program development.

This paper studies safe modifications of pointer programs in the framework of refinement calculus: we want to safely modify a pointer program to another one in a way that the modification is guaranteed to preserve the correctness. For the moment we informally presume that a program T being a correct modification of a program S means that T executes gracefully in any program context that S does so. In other words, T can safely replace any occurrence of S in a program.

Modifying a pointer program correctly is often a delicate task that requires great care. (Throughout, we are particularly interested in correct transformations of pointer programs that are intended for improved memory usage and therefore we deal with mostly such examples of program transformations. The results of this paper are not limited to this particular variety of applications but they can be adopted to wider purposes, e.g., deriving programs from specifications.) Consider the following program S_0 that successively executes two pointer statements.

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 284–304, 2008.

$$S_0 \quad = \quad x := \mathbf{alloc}(12); \mathbf{free}(p)$$

The first statement $x := \mathbf{alloc}(12)$ allocates an arbitrary fresh heap address, updates the value stored at that address to 12, and assigns variable x the fresh address; the second statement $\mathbf{free}(p)$ performs deallocation that reclaims the address p.

One might be tempted to rewrite this program into the following program U, in order to improve memory efficiency.

$$U \quad = \quad [p] := 12; x := p$$

The mutation statement $[p] := 12$ updates the value stored at the address p to 12; the subsequent assignment $x := p$ assigns p to the variable x. This modified program is intended to reuse the address p for the subsequent execution, instead of discarding it away and allocating a fresh address.

This rewrite is not correct, however. The former program S_0 assigns x a fresh address different from p (because the address p is already allocated at the moment of fresh allocation), while the latter program U sets p to x. This means that, the two programs will show different behaviors, if they are put in a larger context of a program that tests on the equality between p and x.

In contrast, it is safe to rewrite the following program S_1 into program U.

$$S_1 \quad = \quad \mathbf{free}(p); x := \mathbf{alloc}(12)$$

Note that this modification is indeed safe but the two programs are not exactly equal. Program U always assigns p to x, while program S_1 assigns either p or an arbitrary fresh address. This means that U can safely replace S_1 (since a graceful execution of S_1 in a context implies a graceful execution of U, which has a lesser variety of variable assignments, in the same context), but not vice versa.

The example above indicates that correct modifications of pointer programs can be a very delicate matter: even slight changes may unexpectedly disrupt the safety of programs. This implies that the correctness of a pointer program of reasonable size would be desperately intricate to be established by an informal argument.

Seeking for a rigorous way to derive safe modifications of pointer programs, we augment the framework of *refinement calculus* [BvW98, Mor94] so that it can handle pointer statements. Refinement calculus deals with concrete programs as well as abstract specifications universally as predicate transformers. Predicate transformers are ordered by *refinement relation* $S \sqsubseteq T$, which is defined by "$S(\varphi)$ implies $T(\varphi)$ for every postcondition φ". In other words, $S \sqsubseteq T$ iff Hoare triple assertion $\{\varphi\}T\{\psi\}$ holds whenever $\{\varphi\}S\{\psi\}$ holds for any pair of precondition φ and postcondition ψ.[1] Thus $S \sqsubseteq T$ implies that T can safely replace S. As for the above example, $S_1 \sqsubseteq U$ holds, but $S_0 \sqsubseteq U$ does not.

To deal with pointer programs in refinement calculus, we make use of formulas of separation logic [ORY01, Rey02] to qualify the properties of (a formalized) heap storage. Then we can verify safety of program modifications by proving propositions expressed in separation logic formulas.

[1] The triple $\{\varphi\}S\{\psi\}$ asserts the total correctness of program, i.e., when executed in any state satisfying precondition φ, program S terminates and establishes postcondition ψ.

Developing a formal proof of the refinement of an entire program, however, can be a task of unmanageable size and complexity. Therefore a modular style development, called *stepwise refinement*, is effective in practice: we successively apply refinement laws to subcomponents of the program until we obtain the desired result.

In order to manage modifications of pointer programs in the paradigm of stepwise refinement, we identify two predicate transformers, called *separating assertion* and *separating assumption*, as fundamental units. They represent primitive forms of heap operations that work in complementary ways: separating assertion, written $\{\varphi\}^*$, works as a generic deallocation statement that reclaims a portion of the heap storage as specified by φ; separating assumption, written $[\varphi]^*$, works as a generic allocation statement that allocates a subheap as specified by φ.

The merits of formulating pointer programs using these two predicate transformers are twofold.

- Separating assertion and separating assumption enjoy several simple but useful refinement laws. They were inspired from predicate transformers called *assertion* and *assumption*, which are useful for handling context information in the stepwise refinement process [Bac88, Mor94, LvW97, BvW98, Gro00]. Assertions and assumptions are defined in terms of conjunction $\cap$ and implication $\Rightarrow$ in classical logic, while separating assertions and separating assumptions are defined in terms of logical operators of separating counterpart, namely, separating conjunction $*$ and separating implication $-\!\!*$ in separation logic. Both counterparts share certain logical properties. In particular, the adjunctive relationship of the classical counterpart:

$$\varphi \cap \psi \text{ entails } \gamma \quad \text{iff} \quad \varphi \text{ entails } \psi \Rightarrow \gamma$$

  is paralleled by that of the separating counterpart [Rey02]:

$$\varphi * \psi \text{ entails } \gamma \quad \text{iff} \quad \varphi \text{ entails } \psi -\!\!* \gamma.$$

  Due to the similarities in logical properties, both counterparts of predicate transformers enjoy similar sets of refinement laws. With these laws, separating assertions and assumptions can be used to handle context information about pointer programs together with assertions and assumptions.
- Basic pointer statements that are in common use in programming can be defined as composition of more primitive predicate transformers such as separating assertions and separating assumptions. This compound representation is useful for proving refinement laws about pointer statements, because finer logical granularity implies greater opportunities of stepwise refinement and lesser size and complexity of proof.

We have implemented and verified the results in this paper in the theorem prover PVS [SRSC01]. In the present paper, however, we will show the proofs for interesting cases only. Most of the remaining cases are an easy exercise, except for a few non-trivial cases whose proofs are too lengthy to write down on papers. Interested readers can take a look at the proof script that is available from the author's homepage `http://www.math.kyoto-u.ac.jp/~susumu/mpc08pvs/`.

Related work. Extensions of refinement calculus by pointer manipulations and separation logic have been studied by a few researchers. Back, Fan, and Preoteasa [BFP03] extended refinement calculus with pointer statements by providing a model that allows explicit reference to the heap storage. Preoteasa [Pre06] presented, based on Back and Preoteasa's axiomatization of states [BP05], another extension of refinement calculus that employs separation logic formulas as a means to expressing heap properties.

The formalization in the present paper basically follows Preoteasa's but we add two substantial extensions to it. First, we introduce separating implication and give a formal semantics for it. Separating implication is a principal logical operation for describing the axiomatic semantics of heap allocation in separation logic, while Preoteasa gives the semantics in a different way without using it. Second, we introduce the new predicate transformers, separating assertions and separating assumptions, and define pointer statements by means of them.

The present paper provides a set of refinement laws for handling separating assertions and separating assumptions and applies them to derivations of pointer programs, in much the same way as is done for their classical counterpart [Bac88, Mor94, LvW97, BvW98, Gro00]. In contrast, Preoteasa's work aims at establishing the correctness of Hoare rules and the frame rule of separation logic; he did not give refinement laws explicitly. The refinement laws and examples in the present paper shall be successfully proved in his formalization. However, we believe that the present formalization enables a finer stepwise refinement process, as we have argued earlier. The most crucial consequence of the present work is that we identify separating assertions and separating assumptions as fundamental units of pointer programs and provide a set of refinement laws that are useful for stepwise refinement of pointer programs.

Outline. The rest of the paper is organized as follows. Section 2 gives a formal definition of refinement calculus and its extension with separation logic formulas. Section 3 introduces predicate transformers that represent primitive operations on the heap storage and defines basic pointer statements in terms of them. Section 4 lists refinement laws for these transformers. In Section 5, we examine and discuss how pointer statements interact with each other, and in Section 6 we show a transformation process for a larger pointer program. Finally Section 7 concludes the paper.

2 Refinement Calculus and Its Formalization

This section summarizes refinement calculus and its formalization in the higher-order logic, following the formalization techniques in [BvW98, BP05]. We then extend refinement calculus with separation logic formulas, in order to deal with pointer statements. This extension is formalized by a variant of the technique proposed in [Pre06].

We will use the following notations. Let A, B be types (sets) in the higher-order logic. We write either $a \in A$ or $a : A$ to mean that a is a member of A. We write $\mathcal{P}_{\mathit{fin}}(A)$ for the finite powerset of A, and $A \setminus B$ for the set difference. A function of type $A \to B$ is often written in λ-notation $\lambda x : A.M$, where M is a term that denotes a value of type B. The type annotations in λ-notation are often left implicit. A predicate over A is a function $A \to \mathsf{bool}$, where bool denotes the type of boolean values $\{\mathsf{true}, \mathsf{false}\}$.

The type of predicates forms a complete boolean lattice. Given predicates φ, ψ over A, we write $\varphi \cup \psi$, $\varphi \cap \psi$, $\varphi \subseteq \psi$, and $\neg\varphi$ to denote predicate union, intersection, inclusion, and complement, respectively. The logical interpretations of $\cup$, $\cap$, $\subseteq$, $\neg$ are disjunction, conjunction, entailment, and negation, resp., and we also write implication $\varphi \Rightarrow \psi$ to abbreviate $\neg\varphi \cup \psi$. A relation R between types A and B is a subset of the product $A \times B$. For a relation $R \subseteq A \times B$, we define its inverse relation by $R^{-1} = \{(y,x) \mid (x,y) \in R\}$. For a function $f : A \to B$, we write *graph*(f) for the graph of f, i.e., $\{(x, f(x)) | x \in A\}$.

2.1 Axiomatization of States

Let Value be the type representing the set of all values. Value includes the set of locations $\mathsf{Location}$ and the set of constants $\mathsf{Constant}$. We assume $\mathsf{Constant}$ at least contains boolean values and nil. For every location x, we write $\mathsf{T}(x)$ to denote the type of values assignable to x.

In order to formalize a state model that has a heap storage, Preoteasa [Pre06] defined $\mathsf{Location}$ to be the disjoint union of sets:

$$\mathsf{Location} \triangleq \mathsf{Variable} \uplus \mathsf{Address} \uplus \{\mathsf{alloc}\},$$

where $\mathsf{Variable}$ is the infinite set of variables, $\mathsf{Address}$ is the infinite set of addresses (of the heap storage), and alloc is a distinguished location denoting the finite set of allocation addresses. We assume that variables and addresses can be assigned arbitrary values and thus the types of locations are given by $\mathsf{T}(x) = \mathsf{Value}$ for any $x \in \mathsf{Variable} \cup \mathsf{Address}$ and $\mathsf{T}(\mathsf{alloc}) = \mathcal{P}_{fin}(\mathsf{Address})$.

Let State be the type of *states*. It is intended that a state is a pair of a mapping from locations to values and a LIFO queue (often called *stack*) that is used for saving/restoring values. Following Back and Preoteasa [BP05], we do not stick to a concrete definition of a state space but rather specify it by a set of state manipulation functions and its axiomatization.

We have three functions for state manipulation.

Lookup	$\mathsf{val}(x) : \mathsf{State} \to \mathsf{T}(x)$	$(x \in \mathsf{Location})$
Update	$\mathsf{set}(x) : \mathsf{T}(x) \to \mathsf{State} \to \mathsf{State}$	$(x \in \mathsf{Location})$
Restore	$\mathsf{del}(x) : \mathsf{State} \to \mathsf{State}$	$(x \in \mathsf{Location})$

Figure 1 gives the axioms[2] that characterize the intended meaning of these functions: $\mathsf{val}(x)(\sigma)$ returns the value that is assigned to the location x in the state σ. $\mathsf{set}(x)(v)(\sigma)$ returns a new state obtained by updating the value of the location x to v in the state σ. $\mathsf{del}(x)(\sigma)$ returns a new state obtained by "popping" the top most stack value and restoring the value to the location x in the state σ. The operation for saving values to the stack will be defined as the relational inverse of del.

In what follows, two states $\sigma, \sigma' \in \mathsf{State}$ are called val-*equivalent* if $\mathsf{val}(x)(\sigma) = \mathsf{val}(x)(\sigma')$ holds for any location $x \in \mathsf{Location}$ [BP05]. We notice that $\sigma = \sigma'$ implies

[2] Back and Preoteasa used these axioms to formalize local variable declarations and recursive procedure calls [BP05]. The present paper does not deal with recursive procedure calls but they should be incorporated without much difficulty.

$$\mathsf{val}(x)(\mathsf{set}(x)(v)(\sigma)) = v \tag{2.1}$$
$$x \neq y \;\Rightarrow\; \mathsf{val}(y)(\mathsf{set}(x)(v)(\sigma)) = \mathsf{val}(y)(\sigma) \tag{2.2}$$
$$\mathsf{set}(x)(v)(\mathsf{set}(x)(u)(\sigma)) = \mathsf{set}(x)(v)(\sigma) \tag{2.3}$$
$$x \neq y \;\Rightarrow\; \mathsf{set}(x)(v)(\mathsf{set}(y)(u)(\sigma)) = \mathsf{set}(y)(u)(\mathsf{set}(x)(v)(\sigma)) \tag{2.4}$$
$$\mathsf{set}(x)(\mathsf{val}(x)(\sigma))(\sigma) = \sigma \tag{2.5}$$
$$\forall\sigma.\exists\sigma'.\sigma = \mathsf{del}(x)(\sigma') \tag{2.6}$$
$$x \neq y \;\Rightarrow\; \mathsf{val}(y)(\mathsf{del}(x)(\sigma)) = \mathsf{val}(y)(\sigma) \tag{2.7}$$
$$\mathsf{del}(x)(\mathsf{set}(x)(v)(\sigma)) = \mathsf{del}(x)(\sigma) \tag{2.8}$$
$$x \neq y \;\Rightarrow\; \mathsf{del}(x)(\mathsf{set}(y)(u)(\sigma)) = \mathsf{set}(y)(u)(\mathsf{del}(x)(\sigma)) \tag{2.9}$$

Fig. 1. Axioms for states [BP05]

σ and σ' are val-equivalent but the converse does not hold in general, because two val-equivalent states may hold different values in the stack.

2.2 Expressions and Predicates over States

An *expression* over State is a function State→A, written Exp(A), where A is the type of the value of the expression.

We remark that the bare variable x just denotes the *location* of the variable and that an expression that refers to the *value* of x is denoted by val(x). For example, an assignment statement `x:=y` is expressed as $x := \mathsf{val}(y)$ in our notation.

Let e be an expression of type Exp(A), x be a location, and e' be an expression of type Exp(T(x)). We define *substitution* of location x in e by e', written $e[e'/x]$, as follows.

$$e[e'/x] \;\triangleq\; \lambda\sigma.e(\mathsf{set}(x)(e'(\sigma))(\sigma))$$

An expression e is called *x-independent* if $e(\mathsf{set}(x)(v)(\sigma)) = e(\sigma)$ holds for any state σ and any value v of type $\mathsf{T}(x)$. For notational convention, we write $FV(e)$ to denote the set of "free" locations in e, i.e., $FV(e) \triangleq \{x \in \mathsf{Location} \mid e \text{ is not } x\text{-independent}\}$.

An expression e is called *finitely-dependent* if $FV(e)$ is finite. An expression e is called val-*determined*, if $e(\sigma) = e(\sigma')$ for any val-equivalent states σ, σ'. An expression e is called *pure* if its value does not depend on the state of the heap, i.e., it holds that

$$e(\mathsf{set}(\mathsf{alloc})(h)(\sigma)) = e(\sigma) \;\text{ and }\; e(\mathsf{set}(a)(v)(\sigma)) = e(\sigma)$$

for any $\sigma \in \mathsf{State}$, $h \in \mathcal{P}_{fin}(\mathsf{Address})$, $a \in \mathsf{Address}$, and $v \in \mathsf{Value}$.

We call an expression e is *nonalloc-independent* [Pre06] if it holds that

$$e(\mathsf{set}(a)(v)(\sigma)) = e(\sigma)$$

for any $\sigma \in \mathsf{State}$, $a \in \mathsf{Address} \setminus \mathsf{val}(\mathsf{alloc})(\sigma)$, and $v \in \mathsf{Value}$. Apparently pure expressions are a subclass of nonalloc-independent expressions.

A predicate (over state) is an expression of boolean type, namely an expression of type Exp(bool).

2.3 Separation Logic Formulas

We denote by Exp the subtype of $\mathsf{Exp}(\mathsf{Value})$ that consists of expressions which are pure, finitely-dependent, and val-determined. In what follows we assume that every program expression e is a member of Exp. The usual (syntactically constructed) expressions that contain no access to the heap storage have type Exp.

Let us use the following shorthand notations.

$$\begin{aligned}\mathsf{add\_alloc}(a)(\sigma) &\triangleq \mathsf{set}(\mathsf{alloc})(\{a\} \cup \mathsf{val}(\mathsf{alloc})(\sigma))(\sigma)\\ \mathsf{diff\_alloc}(h)(\sigma) &\triangleq \mathsf{set}(\mathsf{alloc})(\mathsf{val}(\mathsf{alloc})(\sigma) \setminus h)(\sigma)\end{aligned}$$

We introduce a partial order relation $\preceq$ over states that denotes one state is an extension of the other. The formal definition is by induction on the size of allocation set: $\sigma \preceq \sigma'$ iff $\mathsf{val}(\mathsf{alloc})(\sigma) \subseteq \mathsf{val}(\mathsf{alloc})(\sigma')$ and either

- $\sigma = \sigma'$ (hence $\mathsf{val}(\mathsf{alloc})(\sigma) = \mathsf{val}(\mathsf{alloc})(\sigma')$) or
- $\mathsf{set}(a)(\mathsf{val}(a)(\sigma'))(\mathsf{add\_alloc}(a)(\sigma)) \preceq \sigma'$ holds for any $a \in \mathsf{val}(\mathsf{alloc})(\sigma') \setminus \mathsf{val}(\mathsf{alloc})(\sigma)$.

This is intended that σ' allocates an equal or larger set of addresses than σ does and that σ' may assign different values for the extended set of addresses, i.e., $\mathsf{val}(\mathsf{alloc})(\sigma') \setminus \mathsf{val}(\mathsf{alloc})(\sigma)$. There are no other differences between σ and σ': they have the same assignment of values to variables and the same values saved in the stack. A similar but more abstract notion of this partial ordering can be found in [COY07].

For expressions $e, e' \in \mathsf{Exp}$ and predicates $\varphi, \psi \in \mathsf{Exp}(\mathsf{bool})$, we define the semantics of separation logic formulas as follows.

$$\begin{aligned}\mathbf{emp} &\triangleq \lambda\sigma.(\mathsf{val}(\mathsf{alloc})(\sigma) = \emptyset)\\ e \mapsto e' &\triangleq \lambda\sigma.\big(\mathsf{val}(\mathsf{alloc})(\sigma) = \{e(\sigma)\} \text{ and } \mathsf{val}(e(\sigma))(\sigma) = e'(\sigma)\big)\\ \varphi * \psi &\triangleq \lambda\sigma.\Big(\exists h.\big(h \subseteq \mathsf{val}(\mathsf{alloc})(\sigma) \text{ and}\\ &\qquad\qquad \varphi(\mathsf{set}(\mathsf{alloc})(h)(\sigma)) \text{ and } \psi(\mathsf{diff\_alloc}(h)(\sigma))\big)\Big)\\ \varphi \mathrel{-\!\!*} \psi &\triangleq \lambda\sigma.\big(\forall\sigma'.\ [\sigma \preceq \sigma' \text{ and } \varphi(\mathsf{diff\_alloc}(\mathsf{val}(\mathsf{alloc})(\sigma))(\sigma'))] \text{ implies } \psi(\sigma')\big)\end{aligned}$$

The intended semantics is explained as below.

emp (empty heap). The heap allocates nothing.

$e \mapsto e'$ **(single allocation).** The heap allocates the value e' at the address e (and nothing else).

$\varphi * \psi$ **(separating conjunction).** The heap can be split into two separate heap domains (that is, the allocation set $\mathsf{val}(\mathsf{alloc})(\sigma)$ is divided into two disjoint address sets, h and $\mathsf{val}(\mathsf{alloc})(\sigma) \setminus h$ for some h) so that φ holds for one subheap and ψ holds for the other.

$\varphi \mathrel{-\!\!*} \psi$ **(separating implication).** ψ holds for any extension σ' of the current heap σ such that φ holds for the extended heap domain. That is, no matter how the current allocation set $\mathsf{val}(\mathsf{alloc})(\sigma)$ is extended with an additional address set, say h, that satisfies φ and is disjoint from $\mathsf{val}(\mathsf{alloc})(\sigma)$, ψ holds for the extended allocation set

$h \cup \mathsf{val}(\mathsf{alloc})(\sigma)$. The partial order $\sigma \preceq \sigma'$ indicates that the additional address set is the difference $\mathsf{val}(\mathsf{alloc})(\sigma') \setminus \mathsf{val}(\mathsf{alloc})(\sigma)$ and that arbitrary values are stored at the extended heap addresses.

For notational convenience, we also make use of the following abbreviations.

- We write $e \mapsto -$ to mean that $e \mapsto v$ holds for some value v. The notation $e \mapsto -$ represents a heap that allocates a single address e but the stored value does not matter.
- $e \hookrightarrow e'$ abbreviates $e \mapsto e' * \mathsf{true}$; similarly, $e \hookrightarrow -$ abbreviates $e \mapsto - * \mathsf{true}$. They represent a heap that allocates the address e and possibly other addresses.

We can also formally specify some specific classes of separation logic formulas that advocate stronger logical properties. For example, we specify the class of *precise* predicates [OYR04] as those predicates φ satisfying:

$$\sigma \preceq \sigma_0 \text{ and } \varphi(\sigma) \text{ and } \sigma' \preceq \sigma_0 \text{ and } \varphi(\sigma') \ \text{ implies } \ \mathsf{val}(\mathsf{alloc})(\sigma) = \mathsf{val}(\mathsf{alloc})(\sigma')\ ,$$

for any states σ, σ', and σ_0.[3] Informally, a predicate φ is precise if, for any state σ_0, there exists at most one restriction to a subheap that satisfies φ: that is, there exists at most one subset h of the allocation set $\mathsf{val}(\mathsf{alloc})(\sigma_0)$ such that, for any σ satisfying φ, $\sigma \preceq \sigma_0$ implies $\mathsf{val}(\mathsf{alloc})(\sigma) = h$. We can formally show that, when φ and ψ are precise, so are **emp**, $e \mapsto e'$, $e \mapsto -$, $\varphi * \psi$, and $\varphi \cap \psi$.

2.4 Predicate Transformers

We define pointer manipulating programs as predicate transformers that operate on the type of predicates:

$$\mathsf{Pred} \ \triangleq \ \{\varphi \in \mathsf{Exp}(\mathsf{bool}) \mid \varphi \text{ is nonalloc-independent}\}.$$

We restrict the type of predicates to those nonalloc-independent ones, since execution of programs should not be affected by the values of heap addresses that are not allocated. (Any attempt to access the heap storage at a non-allocated address immediately causes an error, e.g., segmentation fault.) The type Pred is closed under logical operators including those of separation logic.

Proposition 2.1. *Let $x \in \mathsf{Variable}$, $e, e' \in \mathsf{Exp}$, $\varphi, \psi \in \mathsf{Pred}$. Then the following predicates are all members of* Pred.

$$\varphi \cup \psi \quad \varphi \cap \psi \quad \neg\psi \quad \varphi \Rightarrow \psi \quad (\forall x)\varphi \quad (\exists x)\varphi \quad \mathbf{emp} \quad e \mapsto e' \quad \varphi * \psi \quad \varphi \mathrel{-\!\!*} \psi$$

where $(\forall x)$ and $(\exists x)$ are quantifications over Pred:

$$(\forall x)\varphi \ \triangleq \ \lambda\sigma.(\forall v \in \mathsf{Value}.\varphi(\mathsf{set}(x)(v)(\sigma)))$$
$$(\exists x)\varphi \ \triangleq \ \lambda\sigma.(\exists v \in \mathsf{Value}.\varphi(\mathsf{set}(x)(v)(\sigma)))\ .$$

[3] This definition is slightly more general than the usual definition in separation logic: "for any state σ, there is at most one address set h such that h is a subset of the current allocation set and the restriction of σ to h satisfies φ." The two definitions are indeed equivalent if we restrict the set of predicates to the class of nonalloc-independent ones, which we will consider in Section 2.4.

We define the type MTran of predicate transformers that are monotonic w.r.t. predicate inclusion, i.e.,

$$\mathsf{MTran} \;\triangleq\; \{S \in \mathsf{Pred}{\rightarrow}\mathsf{Pred} \mid \forall\varphi,\psi.\big(\varphi \subseteq \psi \text{ implies } S(\varphi) \subseteq S(\psi)\big)\}.$$

We write $S;T$ for the sequential composition of monotonic predicate transformers $S,T \in \mathsf{MTran}$ and define it simply by function composition, i.e., $S;T \triangleq S \circ T$. The type MTran is closed under sequential composition.

The refinement relation $\sqsubseteq$ over MTran is defined as pointwise extension of $\subseteq$, i.e.,

$$S \sqsubseteq T \quad \text{iff} \quad \forall\varphi \in \mathsf{Pred}.S(\varphi) \subseteq T(\varphi)\ .$$

We say S is refined to T, if $S \sqsubseteq T$ holds. The equality $S = T$ holds iff both $S \sqsubseteq T$ and $T \sqsubseteq S$ hold.

We also define operations $\sqcup$ and $\sqcap$ over MTran as pointwise extensions of $\cup$ and $\cap$:

$$(S \sqcup T)(\varphi) \;\triangleq\; S(\varphi) \cup T(\varphi) \qquad\qquad (S \sqcap T)(\varphi) \;\triangleq\; S(\varphi) \cap T(\varphi)\ .$$

Proposition 2.2 ([BvW98]). MTran *forms a complete lattice with* $\sqsubseteq$, $\sqcap$, *and* $\sqcup$ *being the partial order, meet, and join, respectively. The least element of* MTran *is* **abort** $\triangleq \lambda\varphi.\mathsf{false}$ *and the greatest element is* **magic** $\triangleq \lambda\varphi.\mathsf{true}$.

The least element **abort** represents a program whose execution may not terminate normally (because of non-termination or errors). The greatest element **magic** is a miraculous (unimplementable) program that can establish arbitrary postcondition for any precondition.

A predicate transformer $S \in \mathsf{MTran}$ is called *conjunctive* if it holds that $S(\bigsqcap\{\varphi \mid \varphi \in \Gamma\}) = \bigsqcap\{S(\varphi) \mid \varphi \in \Gamma\}$ for any nonempty subset Γ of Pred; dually, S is called *disjunctive* if it holds that $S(\bigsqcup\{\varphi \mid \varphi \in \Gamma\}) = \bigsqcup\{S(\varphi) \mid \varphi \in \Gamma\}$. We also call $S \in \mathsf{MTran}$ *terminating* if $S;\textbf{magic} = \textbf{magic}$; Dually we call S *feasible* if $S;\textbf{abort} = \textbf{abort}$.

3 Statements as Predicate Transformers

We introduce several predicate transformers of type MTran. In what follows, let us write $x,y,\cdots$ for variables, $e,e',\cdots$ for expressions of type Exp, and $\varphi,\psi,\cdots$ for predicates of type Pred. We also write B for *pure* predicates.

3.1 Basic Program Statements

The idle statement **skip** and the assignment statement $x := e$ are defined as follows.

$$\textbf{skip} \;\triangleq\; \lambda\varphi.\varphi \qquad\qquad x := e \;\triangleq\; \lambda\varphi.(\varphi[e/x])$$

Back and Preoteasa [BP05] defined statements for saving/restoring the value of variable. Let $[R] \triangleq \lambda\varphi.\lambda\sigma.(\forall\sigma'.\sigma\, R\, \sigma' \text{ implies } \varphi(\sigma'))$ be the relational demonic update for arbitrary relation $R \subseteq \mathsf{State} \times \mathsf{State}$. The definition of the save statement $\mathsf{Add}(x)$ and the restore statement $\mathsf{Del}(x)$ is given as below.

$$\mathsf{Add}(x) \triangleq \left[graph(\mathsf{del}(x))^{-1}\right] \qquad \mathsf{Del}(x) \triangleq \left[graph(\mathsf{del}(x))\right]$$

Using the save/restore statements in pair, we can put a predicate transformer S into a scope of a local variable x by

$$\mathsf{Add}(x); S; \mathsf{Del}(x)\ .$$

Note that, since $\mathsf{Add}(x)$ is the inverse of $\mathsf{Del}(x)$, it saves the current value of x onto the stack and assigns an *arbitrary* value to x; the saved value is restored on exit by $\mathsf{Del}(x)$. This indicates that the local scope might be alternatively interpreted by means of universal quantification as Morgan [Mor94] did, i.e.,

$$\big(\mathsf{Add}(x); S; \mathsf{Del}(x)\big)(\varphi) = \forall x.(S(\varphi))\ ,$$

where the local variable x is assumed not to occur free in φ. In the present formalization, this is not provable for arbitrary S but it actually holds when S represents a "normal" program where any Add and Del always appear in pair and the value saved in the stack is never accessed until the program execution leaves the corresponding local scope. Throughout the paper, we consider such normal programs only and thus the above alternative interpretation of local scoping is valid, although we have to justify it for each concrete instance of S in a formal development.

3.2 Abstract Statements

The logical quantifications $(\forall x)$ and $(\exists x)$ can be as well regarded as predicate transformers and we call them *demonic assignment* and *angelic assignment* [BvW98], respectively. Both non-deterministically assign a value to the variable, but the angelic assignment chooses a value that establishes the postcondition, if possible.

We can also define predicate transformers that are related to logical conjunction and implication as follows.

$$\{\psi\} \triangleq \lambda\varphi.(\psi \cap \varphi) \qquad [\psi] \triangleq \lambda\varphi.(\psi \Rightarrow \varphi)$$

The predicate transformers $\{\psi\}$ is called *assertion* and $[\psi]$ is called *assumption*. They work as primitive forms of conditional statements: if ψ is satisfied, both of them act like **skip**; otherwise, $\{\psi\}$ acts like **abort**, while $[\psi]$ acts like **magic**.

Some programming constructs can be defined in terms of the above abstract predicate transformers. The conditional statement can be expressed using $\sqcap$:

$$\textbf{if } B \textbf{ then } S \textbf{ else } T \textbf{ fi} \triangleq \big([B]; S\big) \sqcap \big([\neg B]; T\big)\ .$$

Let $\mathcal{F}$ be a monotonic function from MTran to MTran. By proposition 2.2 and Knaster-Tarski theorem, $\mathcal{F}$ has least fixpoint, written $\mu.\mathcal{F}$. This allows us to define recursive program constructs as least fixpoints in MTran. We give below the least fixpoint definition of the while loop.

$$\textbf{while } B \textbf{ do } S \textbf{ od} \triangleq \mu.(\lambda U.\textbf{if } B \textbf{ then } S; U \textbf{else skip fi})$$

3.3 Pointer Statements

Let us consider four basic pointer statements: *lookup* $x := [e]$, *mutation* $[e] := e'$, *allocation* $x := \mathbf{alloc}(e)$, and *deallocation* $\mathbf{free}(e)$. (The *lookup* statement $x := [e]$ updates the variable x to the value stored at the address e of the heap. The other statements have been explained in Introduction.)

The weakest preconditions for these basic pointer statements can be expressed in terms of separation logic formulas [Rey02]. Accordingly they can be recognized as predicate transformers of the following forms:

$$\begin{aligned} x := [e] &= \lambda\varphi.\exists y.(e \hookrightarrow y \cap \varphi[y/x]) && (3.1)\\ [e] := e' &= \lambda\varphi.(e \mapsto -) * (e \mapsto e' \mathrel{-\!\!*} \varphi) && (3.2)\\ x := \mathbf{alloc}(e) &= \lambda\varphi.\forall y.(y \mapsto e \mathrel{-\!\!*} \varphi[y/x]) && (3.3)\\ \mathbf{free}(e) &= \lambda\varphi.(e \mapsto - * \varphi) && (3.4) \end{aligned}$$

where x and y are distinct and $y \notin FV(e) \cup FV(\varphi)$.

Here we can observe that these predicate transformers calculate a compound formula that combines logical connectives such as $*$ and $\mathrel{-\!\!*}$. This suggests that the definitions above could be expressed by combining simpler predicate transformers, each of which corresponds to a single logical connective.

Let us define a pair of new predicate transformers as follows.

$$\{\psi\}^* \triangleq \lambda\varphi.(\psi * \varphi) \qquad\qquad [\psi]^* \triangleq \lambda\varphi.(\psi \mathrel{-\!\!*} \varphi)$$

These are a separating counterpart of assertion and assumption in Section 3.2 and therefore called *separating assertion* and *separating assumption*, respectively. They represent the complementary pair of operations over the heap storage, namely, heap deallocation and allocation. Separating assertion $\{\psi\}^*$ reclaims a part of the current heap as mentioned by ψ; if no subheap establishes ψ, it acts like **abort**. Separating assumption $\{\psi\}^*$ extends the heap with a set of fresh allocations as mentioned by ψ; if no fresh allocation that satisfies ψ is available, it acts like **magic**.

Combining separating assertion, separating assumption, local variable scoping, etc., we give alternative definitions of pointer statements as below:

$$\begin{aligned} x := [e] &\triangleq \mathsf{Add}(y); (\exists y); \{e \hookrightarrow \mathsf{val}(y)\}; x := \mathsf{val}(y); \mathsf{Del}(y) && (3.5)\\ [e] := e' &\triangleq \{e \mapsto -\}^*; [e \mapsto e']^* && (3.6)\\ x := \mathbf{alloc}(e) &\triangleq \mathsf{Add}(y); [\mathsf{val}(y) \mapsto e]^*; x := \mathsf{val}(y); \mathsf{Del}(y) && (3.7)\\ \mathbf{free}(e) &\triangleq \{e \mapsto -\}^* && (3.8) \end{aligned}$$

where x and y are distinct and $y \notin FV(e)$.

We justify these compound definitions as follows. By a simple calculation we have that the compound definition $\{e \mapsto -\}^*; [e \mapsto e']^*$ of the mutation statement is equivalent to the predicate transformer that maps a postcondition φ to $(e \mapsto -) * (e \mapsto e' \mathrel{-\!\!*} \varphi)$; as for the allocation, by the discussion in Section 3.1, we have that $[\mathsf{val}(y) \mapsto e]^*$;

$x := \mathsf{val}(y)$ maps φ to $\forall y.(\mathsf{val}(y) \mapsto e \mathrel{-\!\!*} \varphi[y/x])$, where the universal quantification is introduced by the local variable scoping. Here the local variable scoping is needed for another reason: it guarantees that y is a fresh variable (as required by the side condition for the definition (3.3)). The local variable y can be recognized as fresh because a possible occurrence of y in φ, though being syntactically identical, denotes a different entity: the former denotes the value local to the scope, while the latter denotes the value outside the scope. Similar arguments apply to the lookup and deallocation statements.

The definition of lookup statement (3.5) implicitly contains separating assertion and separating assumption: from the fact that $e \hookrightarrow e' \cap \varphi$ is equivalent to $e \mapsto e' * (e \mapsto e' \mathrel{-\!\!*} \varphi)$ for any φ [Rey02], we have

$$\{e \hookrightarrow e'\} \;=\; \{e \mapsto e'\}^*; [e \mapsto e']^* \,. \tag{3.9}$$

Thus the definition is alternatively expressed as follows.

$$x := [e] \;\triangleq\; \mathsf{Add}(y); (\exists y); \{e \mapsto \mathsf{val}(y)\}^*; [e \mapsto \mathsf{val}(y)]^*; x := \mathsf{val}(y); \mathsf{Del}(y) \tag{3.10}$$

4 Refinement Laws

We will show refinement laws for predicate transformers defined in the previous section. We denote predicate transformers in MTran by $S, T, U, \cdots$, program expressions in Exp by $e, e', \cdots$, and predicates in Pred by $P, Q, \cdots$. We also assume that program expressions and predicates are val-determined and finitely-dependent, unless otherwise stated. Any usual (syntactically constructed) expressions and predicates satisfy these properties.

The compound statements, which combine the basic statements given in the previous section by sequential composition ; , meet $\sqcap$, join $\sqcup$, and the least fixpoint operator, are conventionally called *programs*. The next proposition indicates that a refinement of any substatement of a program gives a refinement of the entire program. (In the subsequent development, we will exploit this fact without explicitly mentioning it.)

Proposition 4.1 (monotonicity[BvW98]). *The sequential composition, meet, and join of predicate transformers preserve monotonicity. That is, for any S, T, S', T' such that $S \sqsubseteq S'$ and $T \sqsubseteq T'$ it holds that $S;T \sqsubseteq S';T'$, $S \sqcap T \sqsubseteq S' \sqcap T'$, and $S \sqcup T \sqsubseteq S' \sqcup T'$. Furthermore, the least fixpoint operator preserves monotonicity, in the sense that $\mu.\mathcal{F}$ is monotonic if $\mathcal{F}$ is a monotonic function that preserves monotonicity.*

We list below several simple refinement laws.

$$S;\mathbf{skip} = \mathbf{skip};S = S \tag{4.1}$$

$$(S \sqcap T);U = (S;U) \sqcap (T;U) \tag{4.2}$$

$$S;(T \sqcap U) \sqsubseteq (S;T) \sqcap (S;U) \tag{4.3}$$

$$(S \sqcup T);U = (S;U) \sqcup (T;U) \tag{4.4}$$

$$(S;T) \sqcup (S;U) \sqsubseteq S;(T \sqcup U) \tag{4.5}$$

4.1 Laws for Local Variable Scoping

The next law indicates that one can freely introduce or eliminate an empty local variable scope.

Proposition 4.2. *[BP05]*

$$\mathsf{Add}(x);\mathsf{Del}(x) \;=\; \mathbf{skip} \tag{4.6}$$

In the course of program refinement, we often need to enlarge or shrink the scope of local variables. The next proposition shows that this is possible, provided that there is no collision in variable names.

Proposition 4.3. *It holds that $S;\mathsf{Add}(x) = \mathsf{Add}(x);S$ and $S;\mathsf{Del}(x) = \mathsf{Del}(x);S$, in either of the following cases.*

- *S is either* **abort**, **magic**, $\{P\}$, $[P]$, $[P]^*$, $y := e$, $y := [e]$, $[e] := e'$, $y := \mathbf{alloc}(e)$, *or* $\mathbf{free}(e)$, *where x and y are distinct and $x \notin FV(e) \cup FV(e') \cup FV(P)$.*
- *S is a separating assertion $\{P\}^*$, where P is precise and $x \notin FV(P)$.*

On exit from a local scope $\mathsf{Add}(x);S;\mathsf{Del}(x)$, the original value of the variable x is restored and hence any assignment to x in the local scope is overridden (the state axiom (2.8)). Thus we can prove the following refinement laws.

$$x := e;\mathsf{Del}(x) \;=\; \mathsf{Del}(x) \quad (4.7) \qquad x := e;\mathsf{Del}(y);\mathsf{Del}(x) \;=\; \mathsf{Del}(y);\mathsf{Del}(x) \quad (4.8)$$

If x and y are distinct and $x \notin FV(e')$, it follows from the state axioms (2.3) and (2.9) that

$$x := \mathsf{val}(y);\mathsf{Del}(y);x := e' \;=\; \mathsf{Del}(y);x := e' \;. \tag{4.9}$$

4.2 Laws for Assertions and Assumptions and Their Separating Counterpart

We list below some of the refinement laws of assertions and assumptions and contrast them with the laws of the separating counterpart.

Proposition 4.4

$$\begin{array}{lll|lll}
\{P\} \sqsubseteq \{Q\} & \textit{if } P \subseteq Q & (4.10) & [P] \sqsubseteq [Q] & \textit{if } Q \subseteq P & (4.14)\\
\{P\};\{Q\} = \{Q\};\{P\} = \{P \cap Q\} & & (4.11) & [P];[Q] = [Q];[P] = [P \cap Q] & & (4.15)\\
\{P\}^* \sqsubseteq \{Q\}^* & \textit{if } P \subseteq Q & (4.12) & [P]^* \sqsubseteq [Q]^* & \textit{if } Q \subseteq P & (4.16)\\
\{P\}^*;\{Q\}^* = \{Q\}^*;\{P\}^* = \{P * Q\}^* & & (4.13) & [P]^*;[Q]^* = [Q]^*;[P]^* = [P * Q]^* & & (4.17)
\end{array}$$

These laws show that assertion (assumption, resp.) is monotonic (antimonotonic, resp.) w.r.t. predicate inclusion and also assertions and assumptions are commutative for sequential composition; similarly for the separating counterpart.

Some of the laws for the interaction between assertions and assumptions and the corresponding laws for the separating counterpart are given below.

Proposition 4.5

$$\begin{array}{lll|lll}
\mathbf{skip} \sqsubseteq [P];\{Q\} & \textit{if } P \subseteq Q & (4.18) & \{P\};[Q] \sqsubseteq \mathbf{skip} & \textit{if } P \subseteq Q & (4.22)\\
\{P\};S \sqsubseteq T \;\textit{ iff }\; S \sqsubseteq [P];T & & (4.19) & \{P\};[Q] \sqsubseteq [Q];\{P\} & & (4.23)\\
\{P\}^*;[Q]^* \sqsubseteq \mathbf{skip} & \textit{if } P \subseteq Q & (4.20) & \mathbf{skip} \sqsubseteq [P]^*;\{Q\}^* & \textit{if } P \subseteq Q & (4.24)\\
\{P\}^*;S \sqsubseteq T \;\textit{ iff }\; S \sqsubseteq [P]^*;T & & (4.21) & \{P\}^*;[Q]^* \sqsubseteq [Q]^*;\{P\}^* & & (4.25)
\end{array}$$

We can observe that the refinement laws for both counterparts run completely in parallel. This is because both counterparts satisfy some significant logical properties such as monotonicity, commutativity, and the adjunctive relationship in parallel. For example, given the proof for the law (4.25):

$$\begin{array}{lll}
& \{P\}^*;[Q]^* \sqsubseteq [Q]^*;\{P\}^* & \\
\text{iff} & P*(Q \mathrel{-\!\!*} \varphi) \subseteq Q \mathrel{-\!\!*} (P*\varphi) & \text{for any } \varphi \\
\text{iff} & Q*P*(Q \mathrel{-\!\!*} \varphi) \subseteq P*\varphi & \text{for any } \varphi \quad \text{(adjunction)} \\
\text{iff} & P*Q*(Q \mathrel{-\!\!*} \varphi) \subseteq P*\varphi & \text{for any } \varphi \quad \text{(commutativity)} \\
\text{if} & Q*(Q \mathrel{-\!\!*} \varphi) \subseteq \varphi & \text{for any } \varphi \quad \text{(monotonicity)} \\
\text{iff} & Q \mathrel{-\!\!*} \varphi \subseteq Q \mathrel{-\!\!*} \varphi & \text{for any } \varphi \quad \text{(adjunction)} \\
\text{iff} & \mathit{True}\,, &
\end{array}$$

we instantly obtain the proof for the other counterpart (4.23) simply by replacing the symbols $\{-\}^*$, $[-]^*$, and $*$ by $\{-\}$, $[-]$, and $\cap$, respectively.[4]

We can also show that some stronger refinement laws hold for the separating counterpart, provided that the predicate P belongs to a particular class of predicates.

Proposition 4.6. *The followings hold for any pure predicate P.*

$$\{P\} \sqsubseteq \{P\}^* \quad (4.26) \qquad \{Q\}^*;\{P\} = \{P\};\{Q\}^* = \{Q \cap P\}^* \quad (4.28)$$

$$[P]^* \sqsubseteq [P] \quad (4.27) \qquad \{Q\}^*;[P] \sqsubseteq [P];\{Q\}^* \quad (4.29)$$

Proposition 4.7. *The followings hold for any precise predicate P.*

$$\{P\}^*;\{Q\} = \{P*Q\};\{P\}^* \quad (4.30) \qquad \{Q\};[P]^* \sqsubseteq [P]^*;\{Q*P\} \quad (4.31)$$

4.3 Commutativity Laws for Statements

The next proposition gives several commutativity laws for pairs of statements that have no collision in variable names.

Proposition 4.8. *The following refinement laws hold, if x and y are distinct, $x \notin FV(e') \cup FV(P)$, and $y \notin FV(e)$.*

$$\begin{array}{llll}
x := e;\{P\} = \{P\};x := e & (4.32) & x := e;y := e' = y := e';x := e & (4.36) \\
x := e;[P] = [P];x := e & (4.33) & (\exists x);y := e' = y := e';(\exists x) & (4.37) \\
x := e;\{P\}^* = \{P\}^*;x := e & (4.34) & (\exists x);\{P\} = \{P\};(\exists x) & (4.38) \\
x := e;[P]^* = [P]^*;x := e & (4.35) & (\exists x);[P]^* \sqsubseteq [P]^*;(\exists x) & (4.39)
\end{array}$$

When two sequentially composed statements have a collision in variable names, we may use refinement laws that exchange the execution order subject to modifications of the original statements. We list below a few such laws for assignment statements.

[4] Note that not every refinement law in one counterpart has a corresponding law in the other. For instance, $\{P\};S \sqsubseteq S$ holds for arbitrary P but $\{P\}^*;S \sqsubseteq S$ does not.

Proposition 4.9.

$$x := e; \{P\} = \{P[e/x]\}; x := e \quad (4.40) \qquad x := e; \{P\}^{*} = \{P[e/x]\}^{*}; x := e \quad (4.42)$$
$$x := e; [P] = [P[e/x]]; x := e \quad (4.41) \qquad x := e; [P]^{*} = [P[e/x]]^{*}; x := e \quad (4.43)$$

5 Refinement of Pointer Statements

Now we are ready to discuss refinement laws for pointer statements. First we show that pointer statements commute with assignment, provided that there is no collision in variable names.

Lemma 5.1. *The following refinement laws hold, if x and y are distinct, $x \notin FV(e')$, and $y \notin FV(e)$.*

$$x := e; y := [e'] = y := [e']; x := e \tag{5.1}$$
$$x := e; [y] := e' = [y] := e'; x := e \tag{5.2}$$
$$x := e; y := \mathbf{alloc}(e') = y := \mathbf{alloc}(e'); x := e \tag{5.3}$$
$$x := e; \mathbf{free}(e') = \mathbf{free}(e'); x := e \tag{5.4}$$

Proof. These refinement laws follow from the definition of the pointer statements and the commutativity laws in proposition 4.3 and 4.8. □

The next lemma shows that any pointer statement that attempts to access the heap storage via a dangling pointer is equal to **abort**.

Lemma 5.2.

$$\{\neg e \hookrightarrow -\}; x := [e] = \{\neg e \hookrightarrow -\}; [e] := e' = \{\neg e \hookrightarrow -\}; \mathbf{free}(e) = \mathbf{abort}$$

Proof. We first show the proof for the deallocation statement. By the definition, we have $\{\neg e \hookrightarrow -\}; \mathbf{free}(e) = \{\neg e \hookrightarrow -\}; \{e \mapsto -\}^{*}$. This is equivalent to **abort**, since for any postcondition φ, $(\neg e \hookrightarrow -) \cap (e \mapsto - * \varphi)$ implies $(\neg e \hookrightarrow -) \cap (e \hookrightarrow -)$, which is unsatisfiable. The proof for the mutation statement is similar.

For the lookup statement, we have $\{\neg e \hookrightarrow -\}; x := [e] = \mathsf{Add}(y); (\exists y); \{\neg e \hookrightarrow -\}; \{e \hookrightarrow \mathsf{val}(y)\}; x := \mathsf{val}(y); \mathsf{Del}(y)$ by proposition 4.3 and law (4.38). This is equivalent to **abort**, because so is the subcomponent $\{\neg e \hookrightarrow -\}; \{e \hookrightarrow \mathsf{val}(y)\}$. □

The things get more complicated and delicate when we deal with interaction between pointer statements, as we have seen in Introduction. In what follows, we demonstrate that some refinement laws for such delicate interactions can be verified by means of stepwise refinement.

We first show the following refinement law that we have argued in Introduction.

Proposition 5.1. *If $y \notin FV(e)$, then it holds that*

$$\mathbf{free}(e); y := \mathbf{alloc}(e') \sqsubseteq [e] := e'; y := e\ . \tag{5.5}$$

Proof. By the definition of allocation and deallocation statements and proposition 4.3, the lhs is equal to:

$$\mathsf{Add}(z); \{e \mapsto -\}^*; [\mathsf{val}(z) \mapsto e']^*; y := \mathsf{val}(z); \mathsf{Del}(z)$$

for some fresh variable z. As we have observed in Section 3, the scope of local variable z constitutes a universal quantification over z and hence (by instantiating z to e) the above compound statement can be refined to[5]

$$\{e \mapsto -\}^*; [e \mapsto e']^*; y := e\ ,$$

which is equal to the rhs of (5.5). □

Let us examine another refinement law, whose proof is more involved.

Proposition 5.2. *If x and y are distinct, $x \notin FV(e')$, and $y \notin FV(e)$, then it holds that*

$$x := [e]; y := \mathbf{alloc}(e') \quad = \quad y := \mathbf{alloc}(e'); x := [e]\ . \tag{5.6}$$

Proof. We prove this by showing that both sides of the equation refine each other.

The following derivation steps prove that the rhs is a refinement of the lhs. (Throughout, we put references to the laws, propositions, etc. that justify the derivation, on the left of each derivation step within a pair of angle brackets. A symbol like ♠ refers to a subsidiary law that will be justified later on.)

$$\begin{aligned}
& x := [e]; y := \mathbf{alloc}(e') \\
\langle \text{Def.\&Prop. 4.3} \rangle \ = \ & \mathsf{Add}(z); x := [e]; [\mathsf{val}(z) \mapsto e']^*; y := \mathsf{val}(z); \mathsf{Del}(z) \\
\langle \spadesuit \rangle \ \sqsubseteq \ & \mathsf{Add}(z); [\mathsf{val}(z) \mapsto e']^*; x := [e]; y := \mathsf{val}(z); \mathsf{Del}(z) \\
\langle \text{(5.1), Prop. 4.3} \rangle \ = \ & y := \mathbf{alloc}(e'); x := [e]
\end{aligned}$$

In the derivation step ♠, we applied the following law.

$$x := [e]; [\mathsf{val}(y) \mapsto e']^* \sqsubseteq [\mathsf{val}(y) \mapsto e']^*; x := [e]$$

This subsidiary law is derived as follows. By the definition of lookup statement, proposition 4.3, and the refinement laws (4.35), (4.39), this is equivalent to showing:

$$\begin{aligned}
& \mathsf{Add}(z); (\exists z); \{e \hookrightarrow \mathsf{val}(z)\}; [\mathsf{val}(y) \mapsto e']^*; x := \mathsf{val}(z); \mathsf{Del}(z) \\
\sqsubseteq \ & \mathsf{Add}(z); (\exists z); [\mathsf{val}(y) \mapsto e']^*; \{e \hookrightarrow \mathsf{val}(z)\}; x := \mathsf{val}(z); \mathsf{Del}(z)\ .
\end{aligned}$$

Hence it is enough to show that $\{e \hookrightarrow \mathsf{val}(z)\}; [\mathsf{val}(y) \mapsto e']^* \sqsubseteq [\mathsf{val}(y) \mapsto e']^*; \{e \hookrightarrow \mathsf{val}(z)\}$. This is derived by the following stepwise refinement.

$$\begin{aligned}
& \{e \hookrightarrow \mathsf{val}(z)\}; [\mathsf{val}(y) \mapsto e']^* \\
\langle (3.9) \rangle \quad = \ & \{e \mapsto \mathsf{val}(z)\}^*; [e \mapsto \mathsf{val}(z)]^*; [\mathsf{val}(y) \mapsto e']^* \\
\langle (4.17) \rangle \quad \sqsubseteq \ & \{e \mapsto \mathsf{val}(z)\}^*; [\mathsf{val}(y) \mapsto e']^*; [e \mapsto \mathsf{val}(z)]^* \\
\langle (4.25) \rangle \quad \sqsubseteq \ & [\mathsf{val}(y) \mapsto e']^*; \{e \mapsto \mathsf{val}(z)\}^*; [e \mapsto \mathsf{val}(z)]^* \\
\langle (3.9) \rangle \quad = \ & [\mathsf{val}(y) \mapsto e']^*; \{e \hookrightarrow \mathsf{val}(z)\}
\end{aligned}$$

[5] In the formal proof, we need to boil down the compound statements into single predicate transformers and prove that they are equal up to extensionality, appealing to the state axioms given in Figure 1. The proof procedure is cumbersome but a routine.

To show the converse refinement, since it holds that $S = (\{P\};S) \sqcup (\{\neg P\};S)$, it is enough to show the following two cases:

$$\{\neg e \hookrightarrow -\}; y := \mathbf{alloc}(e'); x := [e] \ \sqsubseteq\ x := [e]; y := \mathbf{alloc}(e') \ \text{ and}$$
$$\{e \hookrightarrow -\}; y := \mathbf{alloc}(e'); x := [e] \ \sqsubseteq\ x := [e]; y := \mathbf{alloc}(e')\ .$$

For the first case, by the definition of allocation statement, Proposition 4.3, laws (4.31), (4.32), and lemma 5.2, we have $\{\neg e \hookrightarrow -\}; y := \mathbf{alloc}(e'); x := [e] \sqsubseteq \mathsf{Add}(z); \left[\mathsf{val}(z) \mapsto e' * \neg e \hookrightarrow -\right]^*; \mathbf{abort}$. This is equivalent to the least element **abort**, since $\mathsf{Add}(z); \left[\mathsf{val}(z) \mapsto e' * \neg e \hookrightarrow -\right]^*$ is feasible. The feasibility follows from the fact that $\mathsf{val}(z) \mapsto e' * \neg e \hookrightarrow -$ can be satisfied by assigning z a non-allocated heap address other than the one denoted by e.

To establish the other case, we derive as follows.

$$\begin{array}{rl}
 & \{e \hookrightarrow -\}; y := \mathbf{alloc}(e'); x := [e] \\
\langle \text{Prop. 4.3, (5.1)} \rangle & = \ \mathsf{Add}(z); \{e \hookrightarrow -\}; \left[\mathsf{val}(z) \mapsto e'\right]^*; x := [e]; y := z; \mathsf{Del}(z) \\
\langle \clubsuit \rangle & \sqsubseteq \ \mathsf{Add}(z); x := [e]; \left[\mathsf{val}(z) \mapsto e'\right]^*; y := z; \mathsf{Del}(z) \\
\langle \text{Prop. 4.3} \rangle & \sqsubseteq \ x := [e]; y := \mathbf{alloc}(e').
\end{array}$$

To justify the derivation step ♣, we need a subsidiary law:

$$\{e \hookrightarrow -\}; \left[\mathsf{val}(z) \mapsto e'\right]^*; x := [e] \sqsubseteq x := [e]; \left[\mathsf{val}(z) \mapsto e'\right]^*.$$

By a similar calculation as above, we can see that this is established by showing

$$\{e \hookrightarrow -\}; \left[\mathsf{val}(z) \mapsto e'\right]^*; (\exists w); \{e \hookrightarrow w\} \sqsubseteq (\exists w); \{e \hookrightarrow w\}; \left[\mathsf{val}(z) \mapsto e'\right]^*.$$

This refinement law holds, since

$$e \hookrightarrow - \cap (\mathsf{val}(z) \mapsto e' \twoheadrightarrow (\exists w)(e \hookrightarrow w \cap \varphi)) \Rightarrow (\exists w)(e \hookrightarrow w \cap (\mathsf{val}(z) \mapsto e' \twoheadrightarrow \varphi))$$

is a valid separation logic formula for any (postcondition) φ. □

6 Example: Recycling Heap beyond Loop Boundaries

We apply our refinement laws to a larger program. Let us consider the following program.

$$S_0 \ \triangleq\ \begin{array}{l}
\mathsf{Add}(j); \mathbf{while}\ i \neq \mathsf{nil}\ \mathbf{do} \\
\qquad j := \mathsf{val}(i); i := [\mathsf{val}(i)]; o := \mathbf{alloc}(\mathsf{val}(o)); \mathbf{free}(\mathsf{val}(j)) \\
\quad \mathbf{od};\ \mathsf{Del}(j)
\end{array}$$

In the program, variables i and o are initially assigned a pointer to a chain of pointers terminated by the constant nil (Fig. 2-(0)). Every single iteration of the loop body is intended to update the pointer structure in the following steps. First the value of i is saved in j and then the pointer i is dereferenced to follow the link one step forward

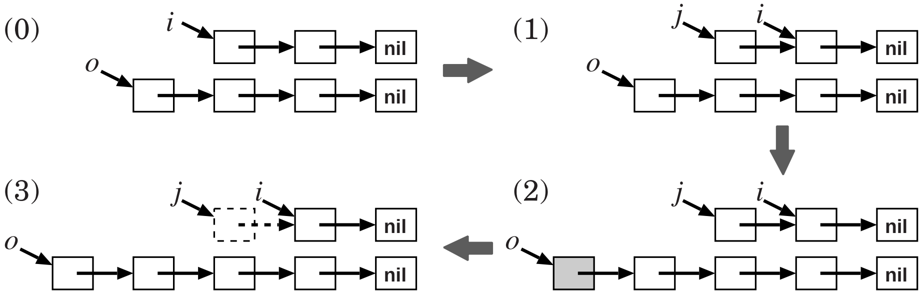


Fig. 2. Updating pointer structure by a single iteration of program S_0. (Dotted lines represent reclaimed data and shaded box represent a freshly allocated heap cell.)

(Fig. 2-(1)). Next, a fresh address is allocated to extend the chain of o by one (Fig. 2-(2)). Finally the address that has been saved in j is reclaimed (Fig. 2-(3)). Repeating this iteration until i reaches to nil, the program "appends" the two chains of pointers; the variable o is updated to refer to the resulting chain of pointers, whose length is the sum of the lengths of the initial chains.

One might be tempted to improve memory usage by rewriting the substatement $o := \mathbf{alloc}(\mathsf{val}(o)); \mathbf{free}(\mathsf{val}(j))$ into $[\mathsf{val}(j)] := \mathsf{val}(o); o := \mathsf{val}(j)$, which is intended to reuse the address $\mathsf{val}(j)$ in place of a freshly allocated one, instead of deallocating it. However, this is not a safe rewrite, as we have observed in Introduction.

We will show another safe way to improving memory usage. Refining the deallocation statement $\mathbf{free}(\mathsf{val}(j))$ together with its subsequent allocation statement $o := \mathbf{alloc}(\mathsf{val}(o))$ beyond the loop boundary, we can obtain the following program T_0 as a refinement of S_0.

$$T_0 \triangleq \begin{array}{l} \mathsf{Add}(j); j := \mathbf{alloc}(\mathsf{nil}); \\ \quad \mathbf{while}\ i \neq \mathsf{nil}\ \mathbf{do} \\ \quad\quad [\mathsf{val}(j)] := \mathsf{val}(o); o := \mathsf{val}(j); j := \mathsf{val}(i); i := [\mathsf{val}(i)] \\ \quad \mathbf{od}; \\ \quad \mathbf{free}(\mathsf{val}(j)); \mathsf{Del}(j) \end{array}$$

To show this formally, we need some lemmas regarding the loop construct.

Lemma 6.1.

(a) If x and y are distinct, $x, y \notin FV(B)$, and $y \notin FV(e)$, we have

$$\begin{array}{l} \mathsf{Del}(x); \mathbf{while}\ B\ \mathbf{do}\ y := e; S\ \mathbf{od}; \mathsf{Del}(y) \\ \sqsubseteq\ y := \mathsf{val}(x); \mathsf{Del}(x); \mathbf{while}\ B\ \mathbf{do}\ y := e; S\ \mathbf{od}; \mathsf{Del}(y)\ . \end{array} \tag{6.1}$$

(b) If S is disjunctive, and the refinement relations $S; [B] \sqsubseteq [B]; S$, $S; [\neg B] \sqsubseteq [\neg B]; S$, and $S; T \sqsubseteq T'$ hold, we have

$$S; \mathbf{while}\ B\ \mathbf{do}\ T; S\ \mathbf{od} \sqsubseteq \mathbf{while}\ B\ \mathbf{do}\ T'\ \mathbf{od}; S\ . \tag{6.2}$$

The first law (6.1) holds, since the effect of assignment $y := \mathsf{val}(x)$ in the rhs is either (i) canceled by the subsequent $\mathsf{Del}(y)$ if the loop condition B is never established or (ii) overridden by the assignment $y := e$ in the first iteration of the loop body. The second law (6.2) holds because $S;(T;S)^n$ $(n \geq 0)$ is refined to $(S;T)^n;S$, which is further refined to $T'^n;S$. We can formally prove these properties by exploiting the fixpoint property of the loop statement and also applying transfinite induction over ordinals [BvW98]. See Appendix for the detail of the proof.

Let us write S_0' for the loop statement in S_0. We derive the refinement relation $S_0 \sqsubseteq T_0$ as follows.

$$
\begin{array}{rcl}
S_0 & = & \mathsf{Add}(j);S_0';\mathsf{Del}(j) \\
\langle(4.6)\rangle & \sqsubseteq & \mathsf{Add}(j);\mathsf{Add}(k);\mathsf{Del}(k);S_0';\mathsf{Del}(j) \\
\langle\text{Lemma 6.1(a)}\rangle & \sqsubseteq & \mathsf{Add}(j);\mathsf{Add}(k);j := \mathsf{val}(k);\mathsf{Del}(k);S_0';\mathsf{Del}(j) \\
\langle(4.24)\rangle & \sqsubseteq & \mathsf{Add}(j);\mathsf{Add}(k);\left[\mathsf{val}(k) \mapsto \mathsf{nil}\right]^*;\left\{\mathsf{val}(k) \mapsto \mathsf{nil}\right\}^*;j := \mathsf{val}(k); \\
 & & \qquad\qquad \mathsf{Del}(k);S_0';\mathsf{Del}(j) \\
\langle(4.12)\rangle & \sqsubseteq & \mathsf{Add}(j);\mathsf{Add}(k);\left[\mathsf{val}(k) \mapsto \mathsf{nil}\right]^*;\left\{\mathsf{val}(k) \mapsto -\right\}^*;j := \mathsf{val}(k); \\
 & & \qquad\qquad \mathsf{Del}(k);S_0';\mathsf{Del}(j) \\
\langle\text{(4.42), Prop. 4.3}\rangle & = & \mathsf{Add}(j);\mathsf{Add}(k);\left[\mathsf{val}(k) \mapsto \mathsf{nil}\right]^*;j := \mathsf{val}(k); \\
 & & \qquad\qquad \mathsf{Del}(k);\left\{\mathsf{val}(j) \mapsto -\right\}^*;S_0';\mathsf{Del}(j) \\
\langle\text{Defs.}\rangle & \sqsubseteq & \mathsf{Add}(j);j := \mathbf{alloc}(\mathsf{nil});\mathbf{free}(\mathsf{val}(j));S_0';\mathsf{Del}(j) \\
\langle\text{Lemma 6.1(b)}\rangle & \sqsubseteq & T_0
\end{array}
$$

To justify the last step of derivation, we need to check if the premises of lemma 6.1(b) hold. The disjunctivity of $\mathbf{free}(\mathsf{val}(j))$ follows from the fact that separating conjunction distributes over disjunction [Rey02]. The first and second prerequisite refinement relations hold by law (4.29); The remaining prerequisite refinement relation is verified as follows.

$$
\begin{array}{rcl}
 & & \mathbf{free}(\mathsf{val}(j));j := \mathsf{val}(i);i := [\mathsf{val}(i)];o := \mathbf{alloc}(\mathsf{val}(o)) \\
\langle(5.6)\rangle & = & \mathbf{free}(\mathsf{val}(j));j := \mathsf{val}(i);o := \mathbf{alloc}(\mathsf{val}(o));i := [\mathsf{val}(i)] \\
\langle(5.3)\rangle & = & \mathbf{free}(\mathsf{val}(j));o := \mathbf{alloc}(\mathsf{val}(o));j := \mathsf{val}(i);i := [\mathsf{val}(i)] \\
\langle(5.5)\rangle & \sqsubseteq & [\mathsf{val}(j)] := \mathsf{val}(o);o := \mathsf{val}(j) j := \mathsf{val}(i);i := [\mathsf{val}(i)]
\end{array}
$$

7 Conclusion and Future Work

We have introduced two new predicate transformers, called separating assertion and separating assumption, into refinement calculus as primitive forms of deallocating and allocating the heap storage. These primitives are defined by means of separating conjunction and separating implication that are fundamental adjunctive logical operators in separation logic. We have shown that they satisfy several refinement laws that are useful for developing safe modification of pointer programs.

There are a few topics that will merit further investigations. The refinement laws and program modifications discussed in this paper are mostly due to simple logical facts and stepwise refinement, but a cumbersome proof task is required in the proof of Proposition 5.1 to show that a particular instance of local variable scoping constitutes a universal quantification. We would be able to simplify this proof step by identifying a class of predicate transformers for which a general refinement law on local variable scoping holds. Another future work would be to take into account of the local reasoning principle (so called frame rule) of separation logic [ORY01, YO02] and to investigate refinement laws that hold under a richer separation context on the heap storage.

Acknowledgment. I thank anonymous reviewers for their helpful comments.

References

[Bac88] Back, R.-J.: A calculus of refinements for program derivations. Acta Informatica 25(6), 593–624 (1988)

[BFP03] Back, R.-J., Fan, X., Preoteasa, V.: Reasoning about pointers in refinement calculus. In: 10th Asia-Pacific Software Engineering Conference (APSEC 2003), pp. 425–434 (2003)

[BP05] Back, R.-J., Preoteasa, V.: An algebraic treatment of procedure refinement to support mechanical verification. Formal Aspects of Computing 17(1), 69–90 (2005)

[BvW98] Back, R.-J., von Wright, J.: Graduate Texts in Computer Science. In: Refinement Calculus: A Systematic Introduction, Springer, Heidelberg (1998)

[COY07] Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: 22nd IEEE Symposium on Logic in Computer Science (LICS 2002), pp. 366–378. IEEE Computer Society, Los Alamitos (2007)

[Gro00] Groves, L.J.: Evolutionary Software Development in the Refinement Calculus. PhD thesis, Victoria University of Wellington (2000)

[LvW97] Laibinis, L., von Wright, J.:Context handling in the refinement calculus framework. Technical Report 118, TUCS Technical Report (1997)

[Mor94] Morgan, C.: Programming from specifications, 2nd edn. Prentice-Hall International Series in Computer Science. Prentice-Hall International, Englewood Cliffs (1994)

[ORY01] O'Hearn, P., Reynolds, J.C., Yang, H.: Local Reasoning about Programs that Alter Data Structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)

[OYR04] O'Hearn, P., Yang, H., Reynolds, J.C.: Separation and information hiding. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, pp. 268–280. ACM Press, New York (2004)

[Pre06] Preoteasa, V.: Mechanical Verification of Recursive Procedures Manipulating Pointers Using Separation Logic. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 508–523. Springer, Heidelberg (2006)

[Rey02] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), pp. 55–74. IEEE Computer Society, Los Alamitos (2002)

[SRSC01] Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference. Computer Science Laboratory, SRI International (November 2001), `http://pvs.csl.sri.com/`

[YO02] Yang, H., O'Hearn, P.: A Semantic Basis for Local Reasoning. In: Nielsen, M., Engberg, U. (eds.) ETAPS 2002 and FOSSACS 2002. LNCS, vol. 2303, pp. 402–416. Springer, Heidelberg (2002)

Appendix

Proof of lemma 6.1(a). Let us define $\mathcal{F}(U) \triangleq \lceil\neg B\rceil \sqcap \lceil B\rceil; y := e; S; U$. The lemma is shown by the following derivation.

$$\begin{array}{rll}
\mathsf{Del}(x);\mu.\mathcal{F};\mathsf{Del}(y) & = & \mathsf{Del}(x);\big(\lceil\neg B\rceil \sqcap \lceil B\rceil;y:=e;S;\mu.\mathcal{F}\big);\mathsf{Del}(y)\\
\langle(4.3),(4.2)\rangle & \sqsubseteq & \mathsf{Del}(x);\lceil\neg B\rceil;\mathsf{Del}(y) \sqcap \mathsf{Del}(x);\lceil B\rceil;y:=e;S;\mu.\mathcal{F};\mathsf{Del}(y)\\
\langle\text{Prop. } 4.3\rangle & \sqsubseteq & \lceil\neg B\rceil;\mathsf{Del}(x);\mathsf{Del}(y) \sqcap \lceil B\rceil;\mathsf{Del}(x);y:=e;S;\mu.\mathcal{F};\mathsf{Del}(y)\\
\langle(4.8),\ (4.9)\rangle & = & \lceil\neg B\rceil;y:=x;\mathsf{Del}(x);\mathsf{Del}(y) \ \sqcap\\
& & \quad\lceil B\rceil;y:=x;\mathsf{Del}(x);y:=e;S;\mu.\mathcal{F};\mathsf{Del}(y)\\
\langle(4.33),\ \text{Prop. } 4.3\rangle & = & y:=x;\mathsf{Del}(x);\lceil\neg B\rceil;\mathsf{Del}(y) \ \sqcap\\
& & y:=x;\mathsf{Del}(x);\lceil B\rceil;y:=e;S;\mu.\mathcal{F};\mathsf{Del}(y)\\
\langle\text{conjunctivity}\&(4.2)\rangle & = & y:=x;\mathsf{Del}(x);\big(\lceil\neg B\rceil \sqcap \lceil B\rceil;y:=e;S;\mu.\mathcal{F}\big);\mathsf{Del}(y)\\
& = & y:=x;\mathsf{Del}(x);\mu.\mathcal{F};\mathsf{Del}(y)
\end{array}$$

□

Proof of lemma 6.1(b). Defining $\mathcal{F}(U) \triangleq \lceil\neg B\rceil \sqcap \lceil B\rceil;T;S;U$ and $\mathcal{G}(U) \triangleq \lceil\neg B\rceil \sqcap \lceil B\rceil;T';U$, we will show that $S;\mathcal{F}^{\alpha}(\mathbf{abort}) \sqsubseteq \mu.\mathcal{G};S$ holds for any ordinal α by transfinite induction. For the case that α is a non-limit ordinal, we derive as follows.

$$\begin{array}{rll}
S;\mathcal{F}^{\alpha+1}(\mathbf{abort}) & = & S;\big(\lceil\neg B\rceil \sqcap \lceil B\rceil;T;S;\mathcal{F}^{\alpha}(\mathbf{abort})\big)\\
\langle(4.3)\rangle & \sqsubseteq & S;\lceil\neg B\rceil \sqcap S;\lceil B\rceil;T;S;\mathcal{F}^{\alpha}(\mathbf{abort})\\
\langle\text{premises}\rangle & \sqsubseteq & \lceil\neg B\rceil;S \sqcap \lceil B\rceil;T';S;\mathcal{F}^{\alpha}(\mathbf{abort})\\
\langle\text{induction}\rangle & \sqsubseteq & \lceil\neg B\rceil;S \sqcap \lceil B\rceil;T';\mu.\mathcal{G};S\\
\langle(4.2)\rangle & = & \big(\lceil\neg B\rceil \sqcap \lceil B\rceil;T';\mu.\mathcal{G}\big);S\\
& = & \mu.\mathcal{G};S
\end{array}$$

The case for α being a limit ordinal follows as below.

$$\begin{array}{rll}
S;\mathcal{F}^{\alpha}(\mathbf{abort}) & = & S;\Big(\bigsqcup_{\beta<\alpha}\mathcal{F}^{\beta}(\mathbf{abort})\Big)\\
\langle S\text{: disjunctive}\rangle & = & \bigsqcup_{\beta<\alpha}\big(S;\mathcal{F}^{\beta}(\mathbf{abort})\big)\\
\langle\text{induction}\rangle & \sqsubseteq & \bigsqcup_{\beta<\alpha}\big(\mu.\mathcal{G};S\big) \ = \ \mu.\mathcal{G};S
\end{array}$$

□

A Hoare Logic for Call-by-Value Functional Programs

Yann Régis-Gianas[1] and François Pottier[2]

[1] INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
LRI, Université Paris-Sud, CNRS, Orsay, F-91405
[2] INRIA Paris - Rocquencourt, Gallium - Domaine de Voluceau - F-78153

Abstract. We present a Hoare logic for a call-by-value programming language equipped with recursive, higher-order functions, algebraic data types, and a polymorphic type system in the style of Hindley and Milner. It is the theoretical basis for a tool that extracts proof obligations out of programs annotated with logical assertions. These proof obligations, expressed in a typed, higher-order logic, are discharged using off-the-shelf automated or interactive theorem provers. Although the technical apparatus that we exploit is by now standard, its application to call-by-value functional programming languages appears to be new, and (we claim) deserves attention. As a sample application, we check the partial correctness of a balanced binary search tree implementation.

1 Introduction

Hoare logic [1, 2, 3] is a discipline for annotating programs with logical formulae, known as assertions, and for extracting logical formulae, known as proof obligations, out of such annotated programs. The validity of the proof obligations, which can be verified either manually or mechanically, entails the correctness of the annotated program. That is, it guarantees that the assertions are correct static predictions of the program's dynamic behavior.

Hoare logic was originally designed for a "while language", that is, a simple imperative programming language, equipped with an iteration construct and a fixed number of global, mutable variables. Recursive, higher-order procedures were the subject of much attention in the late 1970's and early 1980's [4, 5, 6, 7, 8]. More recently, heap-allocated, mutable data structures, as well as object-oriented features, have been deeply investigated. This has led to the development of practical specification languages and tools targeting, for instance, Java [9, 10, 11], C [12] and C# [13].

We would like to put forth the thesis that this traditional focus on imperative programming languages has been, to some extent, detrimental: it has consumed a great amount of energy, while comparatively little effort was being devoted to the key features that will be required in order for the methodology to scale up, such as modularity and abstraction. We would also like to raise a question: since functional programs are significantly easier to check for correctness, why hasn't this activity become routine in the functional programming community, forty years after Floyd and Hoare's seminal papers?

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 305–335, 2008.

On the cost of imperative programming. There are several reasons why functional programming can be considered superior to imperative programming [14]. One of them is that functional programs are easier to reason about. In other words, there is a cost to reasoning about state.

In a typical modern imperative programming language, all heap-allocated data is mutable. As a result, instead of reasoning in terms of high-level entities such as, say, pairs, lists, trees, etc., programmers are forced to reason in terms of a view of the heap as a graph. More concretely, they must write down and prove formulae that involve mappings of memory addresses to memory blocks [12, 15].

The possibility of aliasing means that, whenever some memory block is written, the memory that is accessible through every type-compatible pointer is potentially affected. This makes it difficult to reason about the effects of a single write operation, and creates the problem of representation exposure [16, 17]. In order to address this issue, researchers have developed linear types and regions [18], ownership types [19], and separation logic [20], among other approaches.

Our research agenda. We do not claim that the above issues are not worth investigating: on the contrary, they are quite fascinating. However, it is a pity that we do not, today, have mature tools for checking the correctness of functional programs. This explains why, in this paper, we study a Hoare logic for (call-by-value) functional programs without state.

The programs that we are interested in checking rely heavily on (possibly higher-order) functions, algebraic data structures, and type polymorphism. We claim that it is quite easy to extract succinct and natural proof obligations out of such programs, provided, of course, that they are annotated with specifications.

There are two benefits to be reaped by not reasoning about state. As far as the user is concerned, this leads to simpler specifications and proof obligations. As far as the implementor is concerned, this saves a large part of the "implementation budget", which can then be spent on features such as type polymorphism, type abstraction, and modularity. The importance of these features cannot be overstated: in the end, the key to success is the ability to develop and check program components independently.

Contribution. In this paper, we present the design of a typed, polymorphic, higher-order programming language, where programs can be annotated with assertions expressed in a typed, polymorphic, higher-order logic. We define a procedure for extracting proof obligations out of programs, and show that it is sound. A publicly available prototype tool [21] has been developed, which works in conjunction with the interactive theorem prover Coq [22], with the automated first-order theorem prover Alt-Ergo [23], or with both at once. This tool has been used to check the partial correctness of several non-trivial data structure implementations, including balanced binary search trees and purely functional double-ended queues [24]. We hope to publish detailed accounts of these implementations in the future.

Highlights of our approach. Here are some of the key technical features of our approach.

We focus on partial correctness. We do not require programs to terminate, and do not generate proof obligations to ensure termination. It is up to the user to determine which properties of the code are of sufficient interest to deserve proof, and to insert assertions where desired. At one extreme, a program that contains no assertions leads to no proof obligations. There is no cost to be paid up front for using our methodology.

Our preconditions are prescriptive: it is impossible to call a function unless its precondition F_1 holds. A descriptive interpretation of preconditions can be simulated by using the precondition **true** and the postcondition $F_1 \Rightarrow F_2$. This allows unconditional invocation, and states that the function's result must satisfy F_2 if its argument satisfies F_1.

Values, programs, types, and logical formulae are distinct syntactic categories. Proofs do not necessarily appear within programs: proof obligations are delegated to an external theorem prover, which may or may not require or produce explicit proof terms.

We do not embed values, programs, or formulae within types. Thus, our types are first-order terms: they include type variables, parameterized algebraic data types, and function types, just as in ML. As a result, type inference in the style of Milner [25] is possible, and implemented in our tool [21]. Type inference does not generate any proof obligations. We do not have dependent types, such as lists indexed with an integer length [26], but simulate them as follows. Instead of declaring that x has type *list n*, we declare that x has type *list*, and assert the logical formula $\mathit{length}(x) = n$, where the function *length* is inductively defined at the logical level.

Formulae can refer to values, but not to expressions. This is important, because values are pure, whereas expressions are potentially impure. Although our logic cannot explicitly reason about state, it is nevertheless soundly applicable to programs that involve non-termination, non-determinism, input/output, or mutable state. (Reading an input stream, or dereferencing a pointer to mutable storage, can be viewed as non-deterministic operations.) In that case, it allows establishing properties that do not depend on the behavior of any impure operation. This means, for instance, that we can prove the partial correctness of a functional program even if it has been instrumented with possibly impure debugging, profiling, or logging instructions.

In our programming language, functions, which are potentially impure, are values, so they can appear within formulae. But what does it mean for a formula to refer to a computational function f of type, say, $\tau_1 \overset{\cdot}{\longrightarrow} \tau_2$? Our answer is to view f, at the logical level, as a pair of predicates, which represent f's precondition and postcondition. In other words, when used within a formula, f has type (roughly) $(\tau_1 \rightarrow \mathbf{prop}) \times (\tau_1 \rightarrow \tau_2 \rightarrow \mathbf{prop})$. The two pair projections, written **pre** and **post**, can be used to refer to the pair components. That is, $\mathbf{pre}(f)$ and $\mathbf{post}(f)$ offer lightweight notations for referring to f's precondition and postcondition. When f is a known (**let**-bound) function, this mechanism can be viewed merely as offering abbreviations for known formulae. However, when f is unknown (λ- or $\forall$-bound), it becomes key to writing natural specifications for higher-order functions (§7.5).

In summary, although the technical apparatus that we exploit is by now standard, we believe that it is worth drawing attention to the combination of power and simplicity offered by our technical choices. If extended with a suitable module system, and equipped with a compilation path down to, say, Objective Caml [27], our tool could be used to construct correct purely functional program components, possibly for use within larger, partly imperative programs.

Outline of the paper. The paper is laid out as follows. First, we briefly introduce a higher-order logic, in which assertions and proof obligations are expressed (§2). Then, we present the syntax and call-by-value semantics of a core functional programming language whose expressions carry explicit assertions (§3). We describe the type system, as well as the procedure for extracting proof obligations out of programs (§4). We present a few extensions of the language (§5) and discuss how proof obligations are transformed for submission to external theorem provers (§6). Last, we present a few excerpts of our balanced binary search tree implementation (§7) and review related work (§8).

2 The Underlying Logic

2.1 Syntax

We rely on a mostly standard higher-order logic [28] whose types and terms appear in Figure 1. Types θ include type variables α, parameterized inductive types, function types, product types, and the type **prop** of logical propositions. In the following, the syntax of terms is extended with standard syntactic sugar for falsity, disjunction, implication, equivalence, existential quantification, etc.

The typing rules appear in Figure 2. In general, we write t for terms of arbitrary type. We write F for *formulae*, that is, terms of type **prop**, and P for *predicates*, that is, terms of type $\theta \rightarrow \mathbf{prop}$. The binary operator $\#$, used in several definitions, expresses the fact that two objects have no common free names.

Our logic is not simply-typed. Because our computational language (§3) is polymorphic, and because we wish to lift every computational value up to the logical level, we need polymorphism at the logical level as well. For this reason, we have logical type schemes $\varsigma ::= \forall \bar{\alpha}.\, \theta$, where $\bar{\alpha}$ is a vector of distinct type variables. Every occurrence of a variable x is explicitly applied to a type vector $\bar{\theta}$, which states how the type scheme associated with x is instantiated. For this reason also, we introduce universal quantification over type variables, and use *facts* of the form $\forall \bar{\alpha}.F$. Facts are not formulae: they do not appear in Figure 1. Facts appear only within computational-level type environments Γ (§3.1, Figure 3). The extension of higher-order logic with this very simple form of explicit quantification over types is embedded within the Calculus of Inductive Constructions (§2.2).

$$\begin{array}{lll}
 & & \textbf{\textit{Logical Types}} \\
\theta & ::= \alpha & \textit{Variable} \\
 & \mid d\,\bar{\theta} & \textit{Data} \\
 & \mid \theta \rightarrow \theta & \textit{Function} \\
 & \mid \theta \times \theta & \textit{Product} \\
 & \mid \mathbf{prop} & \textit{Proposition} \\
\varsigma & ::= \forall \bar{\alpha}.\, \theta & \textit{Scheme}
\end{array}$$

$$\begin{array}{lll}
 & & \textbf{\textit{Logical Type Environments}} \\
\Delta & ::= \emptyset & \textit{Nil} \\
 & \mid \Delta, (x : \varsigma) & \textit{Variable} \\
 & \mid \Delta, \bar{\alpha} & \textit{Type Variables}
\end{array}$$

$$\begin{array}{lll}
 & & \textbf{\textit{Logical Terms}} \\
t, F, P & ::= x\,\bar{\theta} & \textit{Variable} \\
 & \mid D\,\bar{\theta}\,(t, \ldots, t) & \textit{Data} \\
 & \mid \lambda(x : \theta).t & \textit{Abstraction} \\
 & \mid t(t) & \textit{Application} \\
 & \mid (t, t) & \textit{Product} \\
 & \mid \pi_1 & \textit{Projection (also written } \mathbf{pre}\textit{)} \\
 & \mid \pi_2 & \textit{Projection (also written } \mathbf{post}\textit{)} \\
 & \mid \mathbf{true} & \textit{Truth} \\
 & \mid t = t & \textit{Equality} \\
 & \mid t \wedge t & \textit{Conjunction} \\
 & \mid \neg\, t & \textit{Negation} \\
 & \mid \forall(x : \theta).t & \textit{Universal Quantification}
\end{array}$$

Fig. 1. The logic (syntax)

The logic offers parameterized inductive types. We assume that each inductive type constructor d carries a fixed integer arity, and that every application $d\,\bar{\theta}$ is arity-consistent. We further assume that d comes with a finite number of data constructors D, each of which is assigned a type scheme of the form:

$$\forall \bar{\alpha}.\, \theta_1 \times \ldots \times \theta_n \rightarrow d\,\bar{\alpha}$$

We impose a positivity condition [29], which is informally summed up as follows: in the above type scheme, the type constructor d (or any type constructor whose definition is mutually recursive with the definition of d) must not appear under the left-hand side of an arrow within $\theta_1, \ldots, \theta_n$.

Although there is an introduction form for inductive types, namely the application of a data constructor D, no elimination form is provided here. We can get away with this omission because the process of *extracting* proof obligations, which is the focus of the present paper, requires no such forms. Of course, when it comes to *discharging* proof obligations, that is, proving theorems, then inductive definitions and proofs become necessary.

$$\begin{aligned}(\Delta,(x:\varsigma))(x) &= \varsigma\\ (\Delta,(x_1:\varsigma))(x_2) &= \Delta(x_2) \text{ if } x_1 \# x_2\\ (\Delta,\bar{\alpha})(x) &= \Delta(x) \quad \text{if } \bar{\alpha} \# \Delta(x)\end{aligned}$$

$$\frac{\Delta(x) = \forall\bar{\alpha}.\,\theta}{\Delta \vdash x\,\bar{\theta} : [\bar{\alpha}\mapsto\bar{\theta}]\theta} \qquad \frac{\begin{array}{c}D : \forall\bar{\alpha}.\,\theta_1 \times \ldots \times \theta_n \to d\,\bar{\alpha}\\ \forall i \qquad \Delta \vdash t_i : [\bar{\alpha}\mapsto\bar{\theta}]\theta_i\end{array}}{\Delta \vdash D\,\bar{\theta}\,(t_1,\ldots,t_n) : d\,\bar{\theta}} \qquad \frac{\Delta,(x:\theta_1) \vdash t : \theta_2}{\Delta \vdash \lambda(x:\theta_1).t : \theta_1 \to \theta_2}$$

$$\frac{\begin{array}{c}\Delta \vdash t_1 : \theta_1 \to \theta_2\\ \Delta \vdash t_2 : \theta_1\end{array}}{\Delta \vdash t_1(t_2) : \theta_2} \qquad \frac{\forall i \qquad \Delta \vdash t_i : \theta_i}{\Delta \vdash (t_1,t_2) : \theta_1 \times \theta_2} \qquad \frac{\Delta \vdash t : \theta_1 \times \theta_2}{\Delta \vdash \pi_i(t) : \theta_i} \qquad \frac{}{\Delta \vdash \mathbf{true} : \mathbf{prop}}$$

$$\frac{\forall i \qquad \Delta \vdash t_i : \theta}{\Delta \vdash t_1 = t_2 : \mathbf{prop}} \qquad \frac{\forall i \qquad \Delta \vdash t_i : \mathbf{prop}}{\Delta \vdash t_1 \wedge t_2 : \mathbf{prop}} \qquad \frac{\Delta \vdash t : \mathbf{prop}}{\Delta \vdash \neg\, t : \mathbf{prop}}$$

$$\frac{\Delta,(x:\theta) \vdash t : \mathbf{prop}}{\Delta \vdash \forall(x:\theta).t : \mathbf{prop}} \qquad \frac{\Delta,\bar{\alpha} \vdash t : \mathbf{prop}}{\Delta \vdash \forall\bar{\alpha}.t : \mathbf{prop}}$$

Fig. 2. The logic (type system)

2.2 Interpretation

Our higher-order logic is embedded within the Calculus of Inductive Constructions [29, 30], abbreviated to CiC in the sequel. Indeed, each type of our logic can be translated into a term of CiC whose type is Type_0. This guarantees that the translation of polymorphic quantification only introduces type variables of type Type_0 in CiC. Each construct of our logic is directly mapped to its counterpart in CiC. This interpretation guarantees that our logic is consistent and validates a number of laws that are used in establishing the soundness of our system (§4.8).

3 The Computational Language

3.1 Syntax

The syntax of our programming language appears in Figure 3. It is equipped with an ML-style type system [25], so types τ and type schemes σ are distinguished. Types include type variables, parameterized algebraic data types, and function types. We write $\dot{\longrightarrow}$ for the computational function type constructor, so as to distinguish it from the logical function type constructor, written $\to$ (Figure 1).

We impose a syntactic separation between values and expressions, and require both operands of the function application operator, as well as **case** scrutinees, to be values. This imposes a style, reminiscent of A-normal form [31], where the result of every intermediate computation is named via a **let** construct. Of course, such a style is quite user-unfriendly, so, in practice, we offer an unrestricted

$$
\begin{array}{lll}
 & & \textbf{\textit{Computational Types}} \\
\tau & ::= \alpha & \textit{Variable} \\
 & \mid d\,\bar{\tau} & \textit{Data} \\
 & \mid \tau \overset{\cdot}{\longrightarrow} \tau & \textit{Function} \\
\sigma & ::= \forall\bar{\alpha}.\,\tau & \textit{Scheme} \\
 & & \\
 & & \textbf{\textit{Computational Type Environments}} \\
\Gamma & ::= \emptyset & \textit{Nil} \\
 & \mid \Gamma, (x : \sigma) & \textit{Variable} \\
 & \mid \Gamma, \bar{\alpha} & \textit{Type Variables} \\
 & \mid \Gamma, \forall\bar{\alpha}.F & \textit{Assumption} \\
 & & \\
 & & \textbf{\textit{Values}} \\
v & ::= x\,\bar{\tau} & \textit{Variable} \\
 & \mid D\,\bar{\tau}\,(v, \ldots, v) & \textit{Data} \\
 & \mid \mathbf{fun}\, f(x : \tau/F) : (x : \tau/F) = \; e & \textit{Recursive Function} \\
 & & \\
 & & \textbf{\textit{Patterns}} \\
p & ::= x\,\bar{\tau} & \textit{Variable} \\
 & \mid D\,\bar{\tau}\,(p, \ldots, p) & \textit{Data} \\
 & & \\
 & & \textbf{\textit{Expressions}} \\
e & ::= v & \textit{Value} \\
 & \mid v(v) & \textit{Function Application} \\
 & \mid \mathbf{let}\,(x\,\bar{\alpha} : \tau/F) = e \;\mathbf{in}\; e & \textit{Local Binding} \\
 & \mid \mathbf{case}\, v \,\mathbf{of}\, c & \textit{Pattern Matching} \\
 & & \\
 & & \textbf{\textit{Cases}} \\
c & ::= \emptyset & \textit{Nil} \\
 & \mid (p \mapsto e) \;[]\; c & \textit{Cons}
\end{array}
$$

Fig. 3. The computation language (syntax)

surface language, and automatically translate it down to the kernel language described here.

The language supports type inference in the style of Hindley and Milner. However, in this paper, we are not concerned with type inference, so we work with explicitly-typed programs. This is visible (i) in the syntax of values and patterns, where variables and data constructors are annotated with vectors of types that indicate how polymorphic type schemes are instantiated, (ii) at **fun** and **let** constructs, where bound variables are annotated with types, and (iii) at **let** constructs, where a vector of type variables $\bar{\alpha}$ can be explicitly bound.

A function definition takes the general form:

$$\mathbf{fun}\, f(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = \; e$$

The symbol / should be read "where". Every function is recursive, so that f is bound within e. The formal parameter x_1 is bound within the precondition

F_1, within the postcondition F_2, and within e. The variable x_2, which stands for the result of the function, is bound within the postcondition F_2. We require every function to be annotated with an explicit precondition and postcondition (if missing, **true** is assumed).

A local variable definition takes the general form:

$$\mathbf{let}\,(x\,\bar{\alpha} : \tau / F) = e_1 \;\mathbf{in}\; e_2$$

The local variable x is bound within F and within e_2. The type variables $\bar{\alpha}$ are bound within τ, F, and e_1. The proposition F serves as a postcondition for e_1. If it is missing, a default postcondition is assumed, whose definition is deferred to §3.3.

A case analysis takes the general form:

$$\mathbf{case}\, v\, \mathbf{of}\, c$$

Here, c is a possibly empty sequence of cases (i.e., branches). Each branch is of the form $(p \mapsto e)$, where the variables that appear in the pattern p are bound within e. Patterns must be linear, that is, a pattern cannot bind a variable twice.

3.2 Lifting Computational Entities to the Logical Level

In a Hoare logic, formulae refer to values. That is, if x is bound, at the computational level, by a **fun**, **let**, or **case** construct, then it is possible for a formula F, embedded in the code within the scope of x, to refer to x. This raises two questions: first, if x has computational type τ, what is its logical type, to be used when typechecking F? Second, if, for the purposes of evaluation, x is substituted with a computational value v, what is the corresponding logical value, to be used when interpreting F?

The problem of lifting types and values to the logical level is trivial in a first-order language. Indeed, the type algebra only contains basic types which are translated to type constants (**int** is mapped to **int**). Besides, computational values are essentially first-order terms, interpreted as data in the logic. Yet, in an higher-order language, functions are first-class values. What should be the logical reflection of their code ?

We answer these questions by lifting both computational types and computational values up to the logical level (Figure 4). That is, to each computational type τ, we associate a logical type $\lceil \tau \rceil$, and to each computational value v, we associate a logical term $\lceil v \rceil$, with the intended property that if v has computational type τ, then $\lceil v \rceil$ has logical type $\lceil \tau \rceil$. Patterns are lifted too. Because patterns form a subset of values, no extra definitions are needed.

As announced (§1), computational functions are reflected, at the logical level, as pairs of a precondition and postcondition. This is made explicit in the lifting of computational function types:

$$\lceil \tau_1 \overset{\cdot}{\longrightarrow} \tau_2 \rceil = (\lceil \tau_1 \rceil \to \mathbf{prop}) \times (\lceil \tau_1 \rceil \to \lceil \tau_2 \rceil \to \mathbf{prop})$$

Types

$$\begin{aligned}\lceil \alpha \rceil &= \alpha \\ \lceil d\,\bar{\tau} \rceil &= d\,\lceil \bar{\tau} \rceil \\ \lceil \tau_1 \overset{\cdot}{\longrightarrow} \tau_2 \rceil &= (\lceil \tau_1 \rceil \to \mathbf{prop}) \times (\lceil \tau_1 \rceil \to \lceil \tau_2 \rceil \to \mathbf{prop})\end{aligned}$$

Type schemes

$$\lceil \forall \bar{\alpha}.\,\tau \rceil = \forall \bar{\alpha}.\,\lceil \tau \rceil$$

Type environments

$$\begin{aligned}\lceil \emptyset \rceil &= \emptyset \\ \lceil \Gamma, (x : \sigma) \rceil &= \lceil \Gamma \rceil, (x : \lceil \sigma \rceil) \\ \lceil \Gamma, \bar{\alpha} \rceil &= \lceil \Gamma \rceil, \bar{\alpha} \\ \lceil \Gamma, \forall \bar{\alpha}.F \rceil &= \lceil \Gamma \rceil\end{aligned}$$

Values

$$\begin{aligned}\lceil x\,\bar{\tau} \rceil &= x\,\lceil \bar{\tau} \rceil \\ \lceil D\,\bar{\tau}\,(v_1, \ldots, v_n) \rceil &= D\,\lceil \bar{\tau} \rceil\,(\lceil v_1 \rceil, \ldots, \lceil v_n \rceil) \\ \lceil \mathbf{fun}\, f(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = \ e \rceil &= (\lambda(x_1 : \lceil \tau_1 \rceil).F_1, \lambda(x_1 : \lceil \tau_1 \rceil).\lambda(x_2 : \lceil \tau_2 \rceil).F_2)\end{aligned}$$

Fig. 4. Lifting computational types and values to the logical level

The first component of the pair, which represents the function's precondition, is abstracted over the function's argument, while the second component, which represents the postcondition, is abstracted over both argument and result.

As a result of this definition, if f is bound, at the computational level, to a function of type $\tau_1 \overset{\cdot}{\longrightarrow} \tau_2$, then a formula embedded within the code, in the scope of f, views f as a pair of predicates, and can refer to $\mathbf{pre}(f)$ and $\mathbf{post}(f)$. (Recall that, as per Figure 1, **pre** and **post** are sugar for the projections π_1 and π_2.) Note that f does not denote a logical function. Within a formula, an application $f(t)$ does not make sense: it is ill-typed.

Values of computational function type (that is, λ-abstractions) are lifted up to the logical level in a way that is consistent with this definition. A function's precondition and postcondition alone determine how it is lifted: its code is ignored. (The conformance of a function's body to its declared pre- and postcondition is checked, of course, via a proof obligation: see rule FUN in Figure 6.) This reflects a philosophy in which the only way of reasoning about the behavior of a function value is via its specification: code never appears within formulae.

In order to lift algebraic data types, we lift every algebraic data type definition into an isomorphic inductive type definition. So, for every computational-level algebraic data type constructor d, there must be a logical-level inductive type constructor, also written d, of identical arity. For every computational-level data constructor

$$D : \forall \bar{\alpha}.\,\tau_1 \times \ldots \times \tau_n \to d\,\bar{\alpha},$$

there must be a logical-level data constructor

$$D : \forall \bar{\alpha}.\,\lceil \tau_1 \rceil \times \ldots \times \lceil \tau_n \rceil \to d\,\bar{\alpha}.$$

Due to the manner in which computational function types are lifted, the positivity condition (§2) requires the type constructor d to not appear *under any side* of a computational arrow within $\tau_1, \ldots, \tau_n$. This can be a limitation (§9).

3.3 Inferring Strongest Postconditions

In order to simplify the definition of the procedure that extracts proof obligations, we have required every **let** construct to carry an explicit postcondition for its left-hand sub-expression (§3.1). In practice, however, annotating every **let** construct would be quite unpleasant, so it is desirable to construct a reasonable postcondition when the user does not provide one.

Ideally, the formula that we should construct in such a situation is the *strongest postcondition* of the left-hand sub-expression. Our logic is, in fact, sufficiently powerful to express strongest postconditions for every construct in our programming language. For instance, the strongest postcondition for a value v is $\lambda x.(x = \lceil v \rceil)$. The strongest postcondition for a function application $v_1(v_2)$ is $\mathbf{post}(\lceil v_1 \rceil)(\lceil v_2 \rceil)$. We could go on and explain how to construct strongest postconditions for **let** and **case** constructs. However, in these two cases, they would be complex formulae, involving existential quantification and disjunction.

Eventually, the postconditions carried by **let** constructs become part of proof obligations, where they appear as hypotheses. For this reason, we do not want them to be too complex: we wish to produce simple, comprehensible proof obligations.

Our answer to this issue is to construct strongest postconditions for values and function applications, as suggested above, but not for **let** and **case** constructs: instead, we rely on the user-provided postcondition, if there is one, or use the trivial postcondition **true**, otherwise.

In practice, when is it necessary for the user to provide an explicit annotation? The left-hand side of a **let** construct can be one of four expression forms: a value, a function application, a **let** form, or a **case** form. In the first two cases, we do use a strongest postcondition. The third case can be made to never happen, up to a conversion to A-normal form [31]. Only the last case remains. In summary, the only case where our simple-minded approach may call for an explicit, user-provided annotation is that of a **let** construct whose left-hand sub-expression is a **case** construct.

3.4 Notions of Substitution

Neither types nor formulae influence execution, but do appear in the syntax of values and expressions, in order to allow stating subject reduction and proving the soundness of our Hoare logic. So, the operational semantics reduces expressions that contain explicit types and formulae. To ensure that these annotations remain consistent as expressions are transformed, we must define a few slightly non-standard notions of substitution.

A single type variable α can appear within logical types as well as within computational types. Similarly, a single variable x can appear within formulae as well as within expressions. For this reason, we write $[\alpha \mapsto \tau]$ for the substitution that replaces every free occurrence of α at the computational level with τ and every free occurrence of α at the logical level with $\lceil \tau \rceil$. Similarly, we write $[x \mapsto v]$ for the substitution that replaces every free occurrence of x at the computational level with v and every free occurrence of x at the logical level with $\lceil v \rceil$.

We have annotated **let** constructs with explicit type abstractions and occurrences of variables with explicit type applications. As a result, contracting a **let**-redex requires contracting type-level β-redexes as well. In order to do so, we write $[x \mapsto \Lambda\bar{\alpha}.v]$ for a substitution that replaces every variable occurrence of the form $x\,\bar{\tau}$ with $[\bar{\alpha} \mapsto \bar{\tau}]v$. Again, this replacement is performed at both computational and logical levels, up to a lifting operation in the latter case.

Last, the notation $[x \mapsto v]$, which denotes a substitution of a value for a variable, is extended to the notation $[p \mapsto v]$, which, when p does not match v, is undefined, and, when p does match v, denotes a simultaneous substitution of values for variables, as follows. The formal definition is:

$$[D\,\bar{\tau}\,(p_1, \ldots, p_n) \mapsto D\,\bar{\tau}\,(v_1, \ldots, v_n)]$$

stands for

$$[p_1 \mapsto v_1] \cup \ldots \cup [p_n \mapsto v_n]$$

Because patterns are linear, this is a union of substitutions whose domains are pairwise disjoint.

3.5 Operational Semantics

A standard small-step, call-by-value operational semantics appears in Figure 5. There are three kinds of redexes (β, **let**, and **case**) and one evaluation context (the left-hand side of a **let** construct). An expression is *stuck* if it is irreducible and not a value. It is easy to check that an expression is stuck if and only if it contains, within an evaluation context, a sub-expression of the form $v_1(v_2)$, where v_1 is not a syntactic function, or of the form $\mathbf{case}\,v\,\mathbf{of}\,\emptyset$.

$$\begin{aligned}
v_1(v_2) &\rightarrow [x \mapsto v_2][f \mapsto v_1]e \\
&\quad\text{if } v_1 \text{ is } \mathbf{fun}\,f(x : \tau/F) : (\ldots) = e \\
\mathbf{let}\,(x\,\bar{\alpha} : \tau/F) = v\ \mathbf{in}\ e &\rightarrow [x \mapsto \Lambda\bar{\alpha}.v]e \\
\mathbf{case}\,v\,\mathbf{of}\,(p \mapsto e)\ []\ c &\rightarrow [p \mapsto v]e \\
&\quad\text{if } [p \mapsto v] \text{ is defined} \\
\mathbf{case}\,v\,\mathbf{of}\,(p \mapsto e)\ []\ c &\rightarrow \mathbf{case}\,v\,\mathbf{of}\,c \\
&\quad\text{if } [p \mapsto v] \text{ is undefined} \\
\mathbf{let}\,(x\,\bar{\alpha} : \tau/F) = e_1\ \mathbf{in}\ e_2 &\rightarrow \mathbf{let}\,(x\,\bar{\alpha} : \tau/F) = e_1'\ \mathbf{in}\ e_2 \\
&\quad\text{if } e_1 \rightarrow e_1'
\end{aligned}$$

Fig. 5. Operational semantics

4 The Type System and Proof System

We now equip the computational language with an ML-style type system and with a proof system (a Hoare logic), which can be viewed as an algorithm for extracting proof obligations out of well-typed programs. For the sake of succinctness, both are described using a single set of judgements, which assert at once that a program is well-typed and is annotated with consistent formulae.

In practice, our tool [21] first checks that the program is well-typed, and, at the same time, infers any omitted type annotations. Then, a set of proof obligations, expressed in our typed higher-order logic, is extracted. The fact that the program (including embedded formulae) is well-typed guarantees that the proof obligations are in turn well-typed.

4.1 Environments

The syntax of type environments Γ appears in Figure 3. As is standard, type environments bind variables and type variables. Environments also contain assumptions, that is, formulae that become hypotheses when proof obligations are emitted. An environment of the form $\Gamma, \forall\bar{\alpha}.F$ is well-formed when $\forall\bar{\alpha}.F$ has type **prop** under $\lceil\Gamma\rceil$.

4.2 Proof Obligations

A proof obligation is a judgement of the form $\Gamma \models F$, where F has type **prop** under $\lceil\Gamma\rceil$. The semantics of the judgment is the validity of the interpretation of F in CiC under the interpretation of the environment Γ, which is decided via an external theorem prover.

4.3 Judgements

The proof system is defined via three judgements, which state properties about values, patterns, and expressions, respectively:

Values	$\Gamma \vdash v : \tau$	(Figure 6)
Patterns	$\Gamma \vdash p : \tau$	(Figure 7)
Expressions	$\Gamma \vdash e : \tau \; \{P\}$	(Figure 8)

4.4 Values

The judgement $\Gamma \vdash v : \tau$ (Figure 6) states that, under the type environment Γ, the value v has type τ. No precondition or postcondition appear in the judgement. Indeed, because values require no computation, they never have a precondition. Furthermore, because all values can be lifted up to the logical level, they don't need an explicit postcondition: the strongest possible postcondition of a value v is simply equality with $\lceil v\rceil$.

$$\begin{aligned}
(\Gamma, (x : \sigma))(x) &= \sigma \\
(\Gamma, (x_1 : \sigma))(x_2) &= \Gamma(x_2) \text{ if } x_1 \# x_2 \\
(\Gamma, \bar{\alpha})(x) &= \Gamma(x) \;\text{ if } \bar{\alpha} \# \Gamma(x) \\
(\Gamma, \forall\bar{\alpha}.F)(x) &= \Gamma(x)
\end{aligned}$$

Var
$$\frac{\Gamma(x) = \forall\bar{\alpha}.\,\tau}{\Gamma \vdash x\,\bar{\tau} : [\bar{\alpha} \mapsto \bar{\tau}]\tau}$$

Data
$$\frac{D : \forall\bar{\alpha}.\,\tau_1 \times \ldots \times \tau_n \rightarrow d\,\bar{\alpha} \qquad \forall i \quad \Gamma \vdash v_i : [\bar{\alpha} \mapsto \bar{\tau}]\tau_i}{\Gamma \vdash D\,\bar{\tau}\,(v_1, \ldots, v_n) : d\,\bar{\tau}}$$

Fun
$$\frac{\begin{array}{c} f \# F_1, F_2 \\ \lceil \Gamma, (x_1 : \tau_1) \rceil \vdash F_1 : \mathbf{prop} \qquad \lceil \Gamma, (x_1 : \tau_1), (x_2 : \tau_2) \rceil \vdash F_2 : \mathbf{prop} \\ \Gamma, (f : \tau_1 \stackrel{\cdot}{\longrightarrow} \tau_2), f = \lceil \mathbf{fun}\, f \ldots \rceil, (x_1 : \tau_1), F_1 \vdash e : \tau_2 \; \{\lambda(x_2 : \lceil \tau_2 \rceil).F_2\} \end{array}}{\Gamma \vdash \mathbf{fun}\, f(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = \; e : \tau_1 \stackrel{\cdot}{\longrightarrow} \tau_2}$$

Fig. 6. The computation language (proof system: values)

Rules Var and Data are straightforward. Rule Fun is more complex. Two premises require the precondition F_1 and postcondition F_2 to be well-formed formulae, under appropriate environments. The last premise checks that the function's body conforms to the function's specification. In order to do so, the type environment is extended with bindings for f and x_1. It is also extended with the hypothesis

$$f = \lceil \mathbf{fun}\, f \ldots \rceil,$$

which by definition of lifting (Figure 4) is synonymous for

$$f = (\lambda(x_1 : \lceil \tau_1 \rceil).F_1, \lambda(x_1 : \lceil \tau_1 \rceil).\lambda(x_2 : \lceil \tau_2 \rceil).F_2).$$

This hypothesis gives meaning to occurrences of $\mathbf{pre}(f)$ and $\mathbf{post}(f)$ within the body of the function, allowing recursive calls to f to be checked. Last, the environment is also extended with the precondition F_1, which means that, within the body of the function, the precondition is assumed to hold. Under this extended environment, the body of the function is required to produce a value that meets the postcondition $\lambda(x_2 : \lceil \tau_2 \rceil).F_2$.

It is not difficult to see that $\Gamma \vdash v : \tau$ implies $\lceil \Gamma \rceil \vdash \lceil v \rceil : \lceil \tau \rceil$. This property is required for the typing rules to construct only well-formed formulae.

4.5 Patterns

The judgement $\Gamma \vdash p : \tau$ (Figure 7) states that a value of type τ can safely be matched against the pattern p, giving rise to (exactly) the bindings described by Γ. As in ML, these bindings are monomorphic (see Pat-Var). Because patterns are linear, the type environments $\Gamma_1, \ldots, \Gamma_n$ in Pat-Data have disjoint domains.

$$\textsc{Pat-Var}\quad (x : \tau) \vdash x : \tau \qquad\qquad \textsc{Pat-Data}\quad \frac{\begin{array}{c} D : \forall \bar{\alpha}. \tau_1 \times \ldots \times \tau_n \rightarrow d\,\bar{\alpha} \\ \forall i \qquad \Gamma_i \vdash p_i : [\bar{\alpha} \mapsto \bar{\tau}]\tau_i \end{array}}{\Gamma_1, \ldots, \Gamma_n \vdash D\,\bar{\tau}\,(p_1, \ldots, p_n) : d\,\bar{\tau}}$$

Fig. 7. The computation language (proof system: patterns)

4.6 Expressions

The judgement $\Gamma \vdash e : \tau\ \{P\}$ (Figure 8) states that, under the type environment Γ, the expression e has type τ and (if it terminates) produces a value whose logical reflection satisfies the predicate P. In such a judgement, P has type $\lceil \tau \rceil \rightarrow \mathbf{prop}$ under $\lceil \Gamma \rceil$.

Rule Value directly reflects this intended meaning: the judgement $\Gamma \vdash v : \tau\ \{P\}$ holds if and only if v has type τ under Γ and its logical reflection $\lceil v \rceil$ provably satisfies P under the hypotheses found in Γ. The premise $\Gamma \models P(\lceil v \rceil)$ is a proof obligation.

Rule App requires the function v_1 and its actual argument v_2 to have matching computational types. Furthermore, it emits two proof obligations, stating that (i) the actual argument must satisfy the function's precondition, and (ii) the function's postcondition must imply the desired postcondition P. In the last premise, we write $P' \Rightarrow P$, where P' and P have type $\lceil \tau_2 \rceil \rightarrow \mathbf{prop}$, for $\forall (x : \lceil \tau_2 \rceil).(P'(x) \Rightarrow P(x))$, where x is fresh for P' and P.

Rule Let checks that e_1 has type τ_1 and that e_1 complies with the postcondition F. Then, the rule performs type generalization, in the style of Milner [25], so

$$\textsc{Value}\quad \frac{\begin{array}{c} \Gamma \vdash v : \tau \\ \Gamma \models P(\lceil v \rceil) \end{array}}{\Gamma \vdash v : \tau\ \{P\}} \qquad\qquad \textsc{App}\quad \frac{\begin{array}{c} \Gamma \vdash v_1 : \tau_1 \overset{\cdot}{\longrightarrow} \tau_2 \qquad \Gamma \vdash v_2 : \tau_1 \\ \Gamma \models \mathbf{pre}(\lceil v_1 \rceil)(\lceil v_2 \rceil) \\ \Gamma \models \mathbf{post}(\lceil v_1 \rceil)(\lceil v_2 \rceil) \Rightarrow P \end{array}}{\Gamma \vdash v_1(v_2) : \tau_2\ \{P\}}$$

$$\textsc{Let}\quad \frac{\begin{array}{c} x \# P \\ \lceil \Gamma, \bar{\alpha}, (x : \tau_1) \rceil \vdash F : \mathbf{prop} \\ \Gamma, \bar{\alpha} \vdash e_1 : \tau_1\ \{\lambda(x : \lceil \tau_1 \rceil).F\} \\ \Gamma, (x : \forall \bar{\alpha}.\, \tau_1), \forall \bar{\alpha}.[x \mapsto x\,\bar{\alpha}]F \vdash e_2 : \tau_2\ \{P\} \end{array}}{\Gamma \vdash \mathbf{let}\,(x\,\bar{\alpha} : \tau_1 / F) = e_1\ \mathbf{in}\ e_2 : \tau_2\ \{P\}} \qquad \textsc{Case-Nil}\quad \frac{\Gamma \vdash v : \tau \qquad \Gamma \models \mathbf{false}}{\Gamma \vdash \mathbf{case}\,v\,\mathbf{of}\,\emptyset : \tau'\ \{P\}}$$

$$\textsc{Case-Cons}\quad \frac{\begin{array}{c} \Gamma \vdash v : \tau \qquad \Gamma' \vdash p : \tau \qquad p \# v, P \\ \Gamma, \Gamma', \lceil v \rceil = \lceil p \rceil \vdash e : \tau'\ \{P\} \\ \Gamma, (\forall \Gamma'. \lceil v \rceil \neq \lceil p \rceil) \vdash \mathbf{case}\,v\,\mathbf{of}\,c : \tau'\ \{P\} \end{array}}{\Gamma \vdash \mathbf{case}\,v\,\mathbf{of}\,(p \mapsto e) \mathbin{[\!]} c : \tau'\ \{P\}}$$

Fig. 8. The computation language (proof system: expressions)

that e_2 is checked under the assignment $(x : \forall\bar{\alpha}.\, \tau_1)$. The hypothesis F is changed into $\forall\bar{\alpha}.[x \mapsto x\,\bar{\alpha}]F$, so as to reflect the fact that x now has polymorphic type.

In the operational semantics, a **let** construct behaves just like a β-redex. This suggests that it could perhaps be treated as syntactic sugar, obviating the need for the Let rule. However, this is not possible, for two reasons. One is that **let** allows type generalization, as explained above, whereas a β-redex does not. The other is that an appropriate postcondition for the function $\lambda x.e_2$ cannot be determined prior to extracting proof obligations: indeed, it has to be $\lambda x.P$, where P is computed only at extraction time.

Rule Case-Nil emits the proof obligation $\Gamma \models$ **false**, which requires the conjunction of hypotheses found within Γ to be inconsistent. This ensures that a **case** construct with zero branches is never executed.

Rule Case-Cons requires the value v and the pattern p to have a common type τ. The environment Γ' collects the variables bound by p, together with their types. Under the hypothesis that a certain instance of p matches v, which is expressed by extending Γ with Γ' and with the hypothesis $\lceil v \rceil = \lceil p \rceil$, the branch e must have the desired type τ' and meet the desired postcondition P. Furthermore, under the hypothesis that no instance of p matches v, which is written $\forall\Gamma'.\lceil v \rceil \neq \lceil p \rceil$, the remaining branches must have type τ' and meet the postcondition P. (Our use of $\lceil p \rceil$ exploits the fact that patterns form a subset of values, a welcome but unessential property.)

When checking a **case** construct with n branches, the $(k+1)$-th branch is checked under the assumption that none of the patterns $p_1, \ldots, p_k$ match the value v. In particular, for $k = n$, the conjunction of all hypotheses of the form $(\forall\Gamma'_i.\lceil v \rceil \neq \lceil p_i \rceil)$ is required to be inconsistent. This ensures that control cannot fall off the end of a **case** construct, or, in other words, that the case analyses are exhaustive. Today's ML and Haskell compilers implement a sound approximation to this check, using a purely syntactic criterion. We also implement this syntactic criterion: when it succeeds, emitting a proof obligation is unnecessary.

4.7 Algorithmic Reading

The judgement $\Gamma \vdash e : \tau\ \{P\}$ defines an algorithm for generating proof obligations. All four parameters of the judgement, namely Γ, e, τ, and P, are inputs of the algorithm, which attempts to build a derivation of the judgement by starting at the root of the expression e and working its way down into the sub-expressions of e. As the algorithm descends, entering **fun**, **let**, and **case** constructs, the environment Γ grows, accumulating new bindings and assumptions. At the same time, the postcondition P is propagated down, in a very straightforward process. At **let** constructs, this propagation process relies on the (default or user-provided, see §3.3) annotation in order to determine which postcondition must be propagated into the left-hand sub-expression. The output of the algorithm consists of the proof obligations, of the form $\Gamma \models F$, carried by the leaves of the derivation (see Value, App, and Case-Nil).

4.8 Soundness

The soundness of our type system and proof system is established in a standard, syntactic manner. The proofs appear in the first author's dissertation [32]. It states that the types and logical assertions carried by a program are a sound approximation of its dynamic semantics.

Lemma 1 (Environment Weakening). *$\Gamma_1, F, \Gamma_2 \vdash e : \tau \; \{P\}$ and $\Gamma_1 \models F$ imply $\Gamma_1, \Gamma_2 \vdash e : \tau \; \{P\}$.*

Lemma 2 (Postcondition Weakening). *$\Gamma \vdash e : \tau \; \{P_1\}$ and $\Gamma \models P_1 \Rightarrow P_2$ imply $\Gamma \vdash e : \tau \; \{P_2\}$.*

Lemma 3 (Type Substitution). *Let ϕ stand for $[\bar{\alpha} \mapsto \bar{\tau}]$. Then, $\Gamma_1, \bar{\alpha}, \Gamma_2 \vdash e : \tau \; \{P\}$ and $\bar{\alpha} \# dom(\Gamma_2)$ imply*

$$\Gamma_1, \phi(\Gamma_2) \vdash \phi(e) : \phi(\tau_2) \; \{\phi(P)\}$$

Lemma 4 (Value Substitution). *Let ρ stand for $[x \mapsto \Lambda\bar{\alpha}.v]$. Then, $\Gamma_1, (x : \forall\bar{\alpha}.\, \tau_1), \Gamma_2 \vdash e : \tau_2 \; \{P\}$ and $\Gamma_1, \bar{\alpha} \vdash v : \tau_1$ and $x \notin dom(\Gamma_2)$ imply*

$$\Gamma_1, \rho(\Gamma_2) \vdash \rho(e) : \tau_2 \; \{\rho(P)\}$$

Lemma 5 (Pattern Matching). *Let $\emptyset \vdash v : \tau$ and $\Gamma' \vdash p : \tau$ and $p \# v$. Then, $[p \mapsto v]$ is defined if and only if the formula $\exists\Gamma'.\lceil v \rceil = \lceil p \rceil$ is valid.*

Theorem 6 (Subject Reduction). *$\Gamma \vdash e : \tau \; \{P\}$ and $e \rightarrow e'$ imply $\Gamma \vdash e' : \tau \; \{P\}$.*

Theorem 7 (Progress). *$\emptyset \vdash e : \tau \; \{P\}$ implies that e is either reducible or a value v such that $P(\lceil v \rceil)$ is valid.*

5 A Few Extensions

Extra assertions. The following construct allows inserting an assertion at an arbitrary point in the code:

$$\textbf{assert}\, F \;\textbf{in}\; e$$

This construct requires F to hold: a proof obligation is emitted. It has no computational content: dynamically, it behaves like e. It is syntactic sugar for **let** $(x : \text{unit}/F) = ()$ **in** e, where x is fresh. It is particularly useful when our tool is used in conjunction with an automated theorem prover: if the theorem prover fails to discharge a proof obligation, the user can use **assert** to cut the proof into smaller, easier steps (if the proof obligation is in fact valid) or to find out what is wrong with the specification (if the proof obligation is in fact invalid).

The construct **absurd**, which statically requires **false** to hold, marks a piece of code as inaccessible. It is syntactic sugar for a **case** construct with zero branches.

Ghost variables and ghost parameters. It is sometimes desirable to explicitly introduce a *ghost variable*, that is, a name for a witness to an existentially quantified hypothesis. For this purpose, we suggest writing

$$\textbf{let logic}\ x : \theta / F\ \textbf{in}\ e$$

This construct binds x within F and e. It requires the assertion $\exists(x : \theta).F$ to hold, and introduces F as a new hypothesis into the context. Assertions embedded within e can refer to x, and their proofs can exploit the hypothesis F. However, occurrences of x at the computational level within e are forbidden, since "**let logic**" has no computational content.

Similarly, it is sometimes desirable to abstract a function with respect to a ghost parameter x, like this:

$$\textbf{fun}\, f[x : \theta](x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = e$$

The brackets bind a ghost parameter x within F_1, F_2, and e. (Again, occurrences of x at the computational level within e are forbidden.) Note that θ can be an arbitrary logical type, so this extension allows explicitly abstracting a function with respect to a proposition or predicate, if desired (see §7.5). Ghost variables and ghost parameters can in principle be viewed as syntactic sugar and translated away [33]. In a realistic implementation, however, they should be primitive notions.

6 Interfacing with External Theorem Provers

The overall verification process, implemented in our prototype tool, is composed of three main steps. First, type inference translates an implicitly typed source code into an explicitly typed internal language, very similar to the language formalized in §3. Second, the rules of the proof system defined in §4 are applied, producing a set of proof obligations. Third, these proof obligations are turned into goals of the two external provers Coq [22] and Alt-Ergo [23]. We describe this last step in the following.

6.1 Coq

Our typed, higher-order logic is easily embedded within the Calculus of Inductive Constructions, which underlies Coq. As a result, exporting proof obligations to Coq is a simple matter of pretty-printing. Implicit type instantiations are handled by Coq's system of implicit arguments. We could have made type instantiations explicit but this would have worsened readability.

Coq is an interactive theorem prover. In order to discharge a proof obligation, the user writes a proof script. An open problem is how to maintain these scripts as the source code of the program evolves. The location in the code where a proof obligation arises might change. The statement of a proof obligation might change as well. Perhaps a solution would be to allow only explicitly-stated, explicitly-named, lemmas to be proved interactively, and to rely solely on an automated theorem prover for discharging anonymous proof obligations, possibly by appeal to an explicit lemma.

6.2 Alt-Ergo

Alt-Ergo [23] is a fully automated theorem prover for a typed, polymorphic, first-order logic. Its design is partly inspired by Simplify [34]. However, Alt-Ergo's logic is typed and polymorphic, whereas Simplify's is untyped. This makes Alt-Ergo superior, from our point of view, to Simplify. Indeed, provided our proof obligations lie in the first-order fragment of our logic, they can be directly exported towards Alt-Ergo. If, on the other hand, we wished to use Simplify, we would have to encode our typed, polymorphic logic into Simplify's untyped logic. Such encodings have been studied [35], but are complex and costly. Of course, the trivial encoding that erases all types is unsound.

In addition to first-order logic, Alt-Ergo has native support for linear arithmetic and for the theory of constructors (that is, function symbols f such that $f(x) = f(y)$ implies $x = y$). The latter is useful for reasoning efficiently about algebraic data structures.

In the general case, our proof obligations are most naturally expressed in a higher-order logic, as shown in this paper. However, higher-order logic can be encoded into first-order logic. A standard encoding introduces "apply" predicates that help simulate β-conversion [36].

Perhaps surprisingly, in our case, this encoding can be made to look fairly natural. The symbols **pre** and **post**, which so far have stood for the pair projections, can be turned into predicates and simulate not only projection, but also application. Furthermore, we can make **pre** a binary predicate and **post** a ternary predicate, avoiding curried function applications. That is, instead of the higher-order formula:

$$f = (\lambda(x_1 : \lceil \tau_1 \rceil).F_1, \lambda(x_1 : \lceil \tau_1 \rceil).\lambda(x_2 : \lceil \tau_2 \rceil).F_2),$$

we can write:

$$\begin{aligned} &\forall(x_1 : \lceil \tau_1 \rceil).(\mathbf{pre}(f, x_1) \Leftrightarrow F_1) \\ \wedge\ &\forall(x_1 : \lceil \tau_1 \rceil).\forall(x_2 : \lceil \tau_2 \rceil).(\mathbf{post}(f, x_1, x_2) \Leftrightarrow F_2) \end{aligned}$$

The pair and the three λ-abstractions have been η-expanded, and the projection and application symbols have been fused into applications of **pre** and **post**. Provided F_1 and F_2 are first-order formulae, this is a first-order formula.

Under this encoding, the definition of the lifting operation on computational types is modified so that the computational function type constructor is no longer interpreted:

$$\lceil \tau_1 \overset{\cdot}{\longrightarrow} \tau_2 \rceil = \lceil \tau_1 \rceil \overset{\cdot}{\longrightarrow} \lceil \tau_2 \rceil$$

That is, we make $\overset{\cdot}{\longrightarrow}$ an uninterpreted binary type constructor at the logical level, so that the lifting of types becomes the identity. Thus, in the above formula, f has logical type $\tau_1 \overset{\cdot}{\longrightarrow} \tau_2$. The type schemes assigned to **pre** and **post** are as follows:

$$\begin{aligned} \mathbf{pre} &: \forall \alpha_1 \alpha_2.\, (\alpha_1 \overset{\cdot}{\longrightarrow} \alpha_2) \times \alpha_1 \rightarrow \mathbf{prop} \\ \mathbf{post} &: \forall \alpha_1 \alpha_2.\, (\alpha_1 \overset{\cdot}{\longrightarrow} \alpha_2) \times \alpha_1 \times \alpha_2 \rightarrow \mathbf{prop} \end{aligned}$$

These declarations are admissible by Alt-Ergo. We believe that it should be possible to go a long way with first-order logic alone, even when the program exploits higher-order functions. However, at present, more practical experience is needed in order to support this conjecture.

7 Application: Finite Sets as Binary Search Trees

As an initial benchmark for our tool [21], we have transcribed Objective Caml's library implementation of finite sets, represented as balanced binary search trees, into our programming language. The code is presented in the concrete syntax of our prototype implementation.

7.1 Parameters

In the following, we fix a type "elt" of elements. We assume that an algebraic data type "bool", whose data constructors are "true" and "false", is available. We assume that an equality check over elements, written "=", is given. It is a function of computational type elt × elt $\overset{\cdot}{\longrightarrow}$ bool, whose specification could be written as follows:

$$\mathbf{post}(=, x_1, x_2, b) \Leftrightarrow (b = \text{true} \Leftrightarrow x_1 = x_2)$$

Similarly, we assume that an ordering relation, written "<", of logical type elt → elt → **prop**, is given, together with an ordering check, also written "<", of computational type elt × elt $\overset{\cdot}{\longrightarrow}$ bool, such that the latter decides the former.

We assume that a type of sets of elements, written "set", is available at the logical level, together with the standard operations (empty set, singleton set, union, membership, etc.) and a number of axioms or theorems that describe the properties of these operations.

In a full-scale programming language, our balanced binary search tree implementation would be a functor, parameterized over the types "elt" and "set", as well as as their operations and axioms.

7.2 Definitions

Figure 9 contains the definition of the algebraic data type "tree", of the logical-level inductive function "elements", and of the inductive predicate "bst". (The concrete syntax is provisional.) A binary tree is either empty or a binary node, carrying a root element, left and right sub-trees, and a cached measure of the tree's height. Our binary search trees are intended to implement a finite set abstraction. The logical function "elements" maps a binary tree to the finite set that it represents. It is defined by induction over the algebraic data type "tree". The property of being a binary search tree is defined by the inductive predicate "bst".

In the definition of "bst", the types of the universally quantified variables "x", "l", "r", "h", "y" are inferred. The types of the function "elements" and of the

```
type tree =
| Empty :  tree
| Node :  (int × tree × elt × tree) → tree

fixpoint elements :  tree → set =
| Empty → empty
| Node (h, l, x, r) → elements (l) ∪ singleton (x) ∪ elements (r)

inductive bst :  tree → prop =
| bst (Empty)
| ∀ (h, l, x, r).
    bst (l) and bst (r) and sup (x, elements (l)) and inf (x, elements (r))
    ⇒ bst (Node (h, l, x, r))
```

Fig. 9. Definitions for binary search trees

predicate "bst" could also be inferred, if desired. In practice, type annotations can always be omitted, except where polymorphic recursion is required.

The definition of "bst" constrains neither the shape of the tree nor the cached height information. This is done by another inductive predicate, named "avl" (not shown). In contrast with the "dependent types" [26, 37, 38] and "generalized algebraic data types" [39] schools, we favor a programming style in which invariants are not necessarily hardwired into data structures at definition time.

7.3 Membership in a Binary Search Tree

Figure 10 shows a function, "member", that checks whether an element "x" is a member of a tree "t". The precondition "bst(t)" requires "t" to be a binary search tree, but does not require it to be balanced, since this is not necessary for correctness. If one wished to (informally) ensure a logarithmic complexity bound, one could strengthen the precondition by adding the requirement "avl(t)". This illustrates how a single data structure can be equipped with multiple invariants, not all of which are necessarily enforced at all times. The postcondition states that the Boolean result tells whether "x" is a member of the set implemented

```
let rec mem_bst (t, x) where bst (t)
returns b where ((b = true) ⇔ (x ∈ elements (t)))
= match t with
  | Empty → false
  | Node (h, l, y, r) →
    if (x = y) then true
    else if (x < y) then mem_bst (l, x)
    else mem_bst (r, x)
  end
```

Fig. 10. Membership in a binary search tree

by the tree “t”. No type annotations are needed in this definition. All types are inferred.

7.4 First-Order Iteration

We now define and specify first-order, persistent iterators [40] over binary search trees. Their expressive power surpasses that of “fold” (§7.5), yet their specification is simpler.

The implementation appears in Figure 11. An iterator is represented as a list of trees, which can be thought of as a stack in a depth-first traversal of some larger tree. (The definition of the type “list”, whose constructors are “Nil” and “Cons”, is omitted.)

To an iterator “i”, there corresponds a set of elements, which we write “remaining(i)”. Its inductive definition is simply the union of the sets of elements of the trees in the list.

An iterator is well-formed only if the trees that it contains have disjoint sets of elements. This is expressed by the inductive predicate “ok”.

```
type iterator = list (tree)

fixpoint remaining : iterator → set =
| Nil → empty
| Cons (t, ts) → elements (t) ∪ remaining (ts)

inductive ok : iterator → prop =
| ok (Nil)
| ∀ (t, ts).
   (elements (t) ∩ remaining (ts)) ≡ empty and bst (t) and ok (ts)
   ⇒ ok (Cons (t, ts))

let iterator (t) where bst (t)
returns i where (ok (i) and remaining (i) ≡ elements (t)) =
   Cons (t, Nil)

let rec next (i) where ok (i)
returns oix
where ((oix = None ⇒ remaining (i) ≡ empty)
   and (∀ (i', x). oix = Some ((i', x))
          ⇒ (remaining (i) ≡ (singleton (x) ∪ remaining (i'))
                and not (x ∈ remaining (i')) and ok (i'))))
= match i with
   | Nil → None
   | Cons (Empty, ts) → next (ts)
   | Cons (Node (h, l, x, r), ts) → Some ((Cons (l, Cons (r, ts)), x))
   end
```

Fig. 11. Iterators over binary search trees

```
let eval_cardinal (t) where bst (t)
returns n where (n = cardinal (elements (t))) =
  let rec count (i, n)
  where ok (i) and n + cardinal (remaining (i)) = cardinal (elements (t))
  returns n’ where (n’ = cardinal (elements (t)))
  = match next (i) with
    | None → n
    | Some ((i’, x)) → count (i’, n + 1)
    end
  in
    count (iterator (t), 0)
```

Fig. 12. A sample client of the iterator abstraction

```
predicate hereditary (inv, s, f) =
  ∀ (x, s’, accu’).
    ((s’ ∪ singleton (x)) ⊆ s and not (x ∈ s’) and inv (accu’, s’ ∪ singleton (x))) ⇒
    (pre (f) (accu’, x) and (∀ accu”. (post (f) (accu’, x) (accu”) ⇒ inv (accu”, s’))))

lemma hereditary_subset: ∀ (s, s’, inv, f).
  (s’ ⊆ s and hereditary (inv, s, f)) ⇒ hereditary (inv, s’, f)

let rec fold [s, inv] (accu, t, f)
where bst (t) and elements (t) ⊆ s and inv (accu, s) and hereditary (inv, s, f)
returns accu’ where inv (accu’, s \ elements (t))
= match t with
  | Empty → accu
  | Node (_, l, x, r) →
    let accu_l = fold [s, inv] (accu, l, f) in
    let accu_x = f (accu_l, x) in
      fold [s \ (elements (l) ∪ singleton (x)), inv] (accu_x, r, f)
end
```

Fig. 13. Higher-order iteration over binary search trees

The function “iterator” creates an iterator “i” out of a tree “t”, and satisfies the postcondition “ok(i) $\wedge$ elements(t) $\equiv$ remaining(i)”, where $\equiv$ stands for extensional equality of sets (which may, or may not, coincide with definitional equality). This initial iterator is simply the singleton list [t].

The function “,”, when applied to an iterator “i”, returns either nothing or a pair of a new iterator “i’” and an element “x”. The postcondition describes how these values are related. (The definition of the type “option”, whose constructors are “None” and “Some”, is omitted.)

Figure 12 shows how iterators are used. Here, the client is a function that counts the number of elements in a tree. It does not depend on the internals of the tree data structure: it only depends on the specification of iterators, which

```
let incr (x, z) returns y where (y = x + 1) = x + 1

predicate cardinal_inv (t) =
  fun (accu, s) → (accu + cardinal (s) = cardinal (elements(t)))

lemma is_hereditary_cardinal_inv:
  ∀ t. hereditary (cardinal_inv (t), elements (t), incr)

let eval_cardinal (t) where bst (t)
returns x where (x = cardinal (elements (t)))
= fold [elements (t), cardinal_inv (t)] (0, t, incr)
```

Fig. 14. A sample client of the fold operator

is expressed in terms of abstract (logical-level) sets. So, this client code could be placed in another module, without access to the definition of trees.

The "eval_cardinal" function performs a loop, expressed as an internal recursive function, with an integer accumulator n. It corresponds directly to a **foreach** construct in Java or C#. The precondition of this internal function represents the loop invariant: the number of elements counted so far, plus the number of elements remaining to be seen, equals the total number of elements of the set. The postcondition is simply the precondition, specialized to the case where no elements remain.

The precondition of "count" must also state that "i" is an "ok" iterator, even though it does not have to know about the definition of "ok". This is somewhat undesirable. In the future, we will want to allow defining a dependent sum type of the form "i : iterator **where** ok(i)", and exporting it as an abstract type.

The definition of "eval_cardinal" is syntactically somewhat heavy, as it is expressed in our core language. In a full-scale programming language, a more palatable syntax for loops could be introduced, and desugared into recursive functions and iterators. A single formula, the loop invariant, would have to be written down, instead of two formulae in this low-level version of the code.

7.5 Higher-Order Iteration

We now present a specification of the classic "fold" higher-order function over sets implemented as binary search trees. The specification is rather more complex than that of first-order iterators, for at least two reasons. First, the specification must mention the client's state (the accumulator) and invariant. Second, because the code is not tail-recursive, some information is implicitly encoded within the stack, and a ghost parameter is used to make it explicit in the specification.

The function "fold" is parameterized over two ghost variables, namely the client invariant "inv" and a set "s" of remaining elements. In the case of first-order iterators, the former was unnecessary because the client retains control over the desired invariant, and the latter was unnecessary because the set of remaining elements was directly expressed as "remaining(i)". Here, the set of remaining elements is implicit in the stack, so a ghost variable must be used in order to refer to it.

The precondition of “fold” expresses the following requirements. First, “t” must a binary search tree. Second, the elements of “t” must form a subset of “s”. This reflects that, in general, “t” is a sub-tree of a larger tree over which iteration is taking place. Third, the invariant must initially hold. Last, the invariant must be hereditary: that is, at any time, if an element “x” is picked among the remaining elements, the invariant guarantees that it is legal to apply “f” to the current accumulator and to “x”, and guarantees that the new accumulator thus obtained will still satisfy the invariant.

This definition is certainly somewhat overwhelming. It shows, at the same time, that it is possible to specify and exploit higher-order functions in our framework, and that there is a cost in complexity to be paid for doing so. More experience is needed before we can tell how easily higher-order functions can be defined and used in practice.

7.6 Quantitative Results

The binary search tree library contains 22 functions. The development is composed of 108 lemmas, 603 lines of specification and 247 lines of code. The factor of 3 in size between specification and code does not necessarily mean that specifications must be heavy: in a realistic system, a large part of the specification would be imported from a standard library. 749 proof obligations are generated and are proven automatically by Alt-Ergo [23]. Only one lemma, stating that the height of a tree is nonnegative, requires an induction in Coq [22]; the other lemmas are proven automatically by Alt-Ergo. About 80% of the proofs require less than 5 seconds to be proven by Alt-Ergo. Yet, about 10% of the proofs require from 10 to 30 minutes. A forthcoming extension of Alt-Ergo with support for reasoning modulo associativity and commutativity of some set operations (such as set union) would perhaps improve these results.

8 Related Work

The roots of our work lie in Hoare logic [1, 2]. Extensions of Hoare logic with support for recursive, higher-order procedures were heavily studied in the late 1970’s and early 1980’s [4, 5, 6, 7, 8]. In particular, the issue of completeness received a lot of attention after Clarke [4] proved that there can be no sound and complete Hoare logic for a programming language equipped with recursive, higher-order procedures and global variables. Clarke’s result, however, is based upon the assumption that formulae and proof obligations are expressed in a first-order logic. Damm and Josko [6] point out that, by moving to higher-order logic, it is possible to work around Clarke’s negative result. In this paper, we follow Damm and Josko and allow specifications to be expressed in higher-order logic. The intuitive justification for this approach is that, if functions can abstract over functions, then specifications must abstract over specifications.

Our work has been strongly inspired by several existing, practical tools for checking imperative programs [10, 11, 12, 13, 41, 42]. This paper is an attempt to exploit the strengths of these works while steering away from imperative programming and placing renewed emphasis on polymorphism and modularity.

Our method for generating proof obligations is particularly straightforward: it appears in its entirety in Figure 8. In comparison with the method used in ESC/Java [43], we avoid a translation to "passive form" because we have no assignments to begin with. We avoid the exponential explosion that could follow from the interplay between sequences and alternatives by requiring sequences (that is, **let** constructs) to carry user-provided postconditions (§3.3).

Our system is not sound with respect to a call-by-name dynamic semantics. There are at least two reasons for this fact. First, some divergent expressions admit **false** as a valid postcondition. If such an expression e_1 is made the first component of a sequence, as in "$\mathbf{let}\, x/\mathbf{false} = e_1 \;\mathbf{in}\; e_2$", then second component e_2 is checked under the assumption **false**. As a result, all of the the proof obligations found within e_2 are vacuously satisfied. This is sound under call-by-value evaluation, because e_2 is never executed. It is unsound under call-by-name evaluation, because e_2 is executed immediately (after binding x to a suspension). The second reason is that, in a call-by-name semantics, every type is inhabited by a bottom value, and some types are inhabited by infinite values. This is not reflected in the way we lift computational values and types up to the logical level.

Scott's logic of computable functions [44] interprets λ-terms in a denotational model, where equality implies, or coincides with, observational equivalence. It comes with a set of sound deduction rules, and allows explicit reasoning about divergence and equality of computations. It admits call-by-value and call-by-name variants. It was implemented as early as 1972 by Milner [45]. More recent implementations [46, 47, 48] embed Scott's LCF within some form of higher-order logic. In a somewhat similar vein, Longley and Pollack [49] embed the functional core of Standard ML, via a fully abstract denotational semantics, into higher-order logic.

Our approach is less elaborate: by focusing on partial correctness, by adopting a call-by-value semantics, and by lifting only values, as opposed to expressions, up to the logical level, we are able to ignore non-termination issues entirely, and to work with value spaces that do not have bottom elements or definedness orderings. By contrast, tools or approaches that focus on lazy functional programs, such as Programatica [50, 51] or the Cover translator [52], require reasoning about non-termination, resulting in proof obligations that can become cluttered with definedness side conditions. The simplicity of our approach comes at a cost: our system can neither establish termination of an expression nor reason about observational equality of expressions.

Honda and Yoshida [53] define a Hoare logic for call-by-value higher-order functions, to which our system seems rather analogous. A technical difference is that Honda and Yoshida allow expressions (including, in particular, function applications) to appear within formulae, and interpret equality as observational equality; whereas we only lift values to the logical level, and interpret equality as equality of values. Honda and Yoshida's system does not seem to have been implemented.

Smith [54, §4.4.1] defines a type theory with partial objects, where the type $\bar{A}$ contains the possibly non-terminating computations that yield a result of type A. Smith notes that the fixed point axiom, which has type $(A \to A) \to A$, is sound only at *admissible* types. As an example of a non-admissible type, he offers a type D whose definition can be read: "D is the type of the partial functions g of naturals to naturals such that g diverges for at least one input". It is easy to construct a function of type $D \to D$ whose least fixed point is in fact a total function: this shows that D is not admissible. A reviewer of an earlier version of the present paper noted that "g diverges for at least one input" seems expressible, in our system, as $\exists x.\forall y.\neg\mathbf{post}(g)(x)(y)$, and wondered if Smith's example could be adapted to show that our system is unsound. One should note, first, that although this formula indeed represents a *sufficient* condition for g to diverge for at least one input, it is not a *necessary* condition. Indeed, the predicate $\mathbf{post}(g)$ denotes the programmer-provided postcondition of g; it does not denote the actual semantics of g. Second, when the programmer supplies an explicit definition of the predicate $\mathbf{post}(g)$ (which he must do), this definition cannot refer to g itself. As a result, there is no way that the postcondition associated with g can be the self-referent "g diverges for at least one input".

ESC/Haskell [55] allows annotating Haskell programs with preconditions and postconditions that are also expressed in Haskell. A special-purpose theorem prover, based on symbolic evaluation of Haskell terms, is developed.

The theorem prover Coq [22] can be used as a programming language, in which programs are both developed and proved correct. The Compcert certified compiler [56] offers an example of a large program developed in this style. However, there is some agreement that Coq is not (yet) a convenient programming language: for instance, it only allows writing pure, terminating functions.

The programming language Russell [38] extends Coq with facilities for defining programs annotated with assertions, in the style of Hoare logic. There are many similarities between Russell and our work. One important technical difference is that we separate the typechecking process, which is performed first and remains traditional, and the process of extracting proof obligations, which runs as a second phase, whereas, in Russell, as in Coq, typechecking and proving are one and the same activity. In particular, Russell encourages the use of indexed types, like *list* n, so that typechecking can give rise to proof obligations: for instance, supplying an actual argument of type *list* m to a function that expects a formal parameter of type *list* n generates the proof obligation $m = n$. Another difference is that Russell terms are elaborated into Coq terms, whereas we adopt a less foundational approach and are happy to trust an external theorem prover.

Hoare Type Theory [33, 57] is somewhat similar to our system, insofar as it offers decidable basic typechecking and decidable generation of proof obligations. It also shares our use of higher-order logic and our emphasis on polymorphism and abstraction. It is much more ambitious than our proposal, in that it attempts to deal not only with algebraic data types and higher-order functions, but also with heap-allocated, mutable state. As a result, its design and metatheory are considerably more involved.

Some authors [33, 55, 58, 59] allow code to appear in specifications. This is motivated partly by a desire to make formulae executable, so as to allow assertions to be checked at runtime, and partly by fear that, otherwise, a single functionality might have to be implemented twice: once at the computational level, once at the logical level. Our technical and philosophical choice is different: we consider all code as potentially impure, and do not allow code to appear within specifications. We do not check assertions at runtime: if the programmer wishes to insert a runtime check, she must do so explicitly. Furthermore, we believe that, in practice, opportunities for code sharing between computational and logical levels are rare: the oft-cited case of lists is one of only a few situations where implementation and specification coincide.

Indexed types [26, 60] and refinement types [61] rely on so-called indices. Indices are elements of some mathematical domain, such as an arbitrary finite set, or the set of all natural numbers. Types are enriched with constraints over indices, allowing invariants, preconditions, and postconditions to be expressed. The syntax of constraints is carefully restricted so as to ensure that constraint entailment is decidable. This allows proof obligations to be automatically checked. Generalized algebraic data types [39] are also an instance of this idea, where indices are types, that is, first-order terms. The appeal of this approach resides in the high degree of automation that it allows. On the other hand, this comes at the price of a restriction to a decidable logic. In fact, our decision of using a highly expressive, hence undecidable, logic was motivated by our earlier study of generalized algebraic data types [62, 63].

Going beyond indexed types, several programming languages offer full dependent types [37, 64, 65, 66]. By exploiting the Curry-Howard isomorphism, they allow code and proofs to be expressed and combined within a single language. This allows programs to appear more self-contained, but means that a fragment of the programming language must be a consistent logic, and requires mechanisms to assist the user in building proofs. Our design, which relies on an off-the-shelf theorem prover, is more modular.

9 Conclusion

We have presented a simple methodology for extracting proof obligations out of call-by-value functional programs. Our proposed future work includes:

- extending our prototype implementation [21] and equipping it with a compilation path down to Objective Caml;
- relaxing our positivity condition (§3.2), which restricts the use of functions within data structures, preventing, for instance, the standard definition of infinite streams;
- internalizing type equality, that is, introducing equations between types into the syntax of formulae, together with suitable conversion rules for exploiting

such equations; indeed, we, and other authors [33], have noticed that such an extension would subsume generalized algebraic data types [39];
- studying the issues raised by modularity and mutable state.

Acknowledgement. The authors wish to thank the anonymous reviewers of a previous version of this paper for contradicting a false claim and offering useful comments and suggestions. Thanks are also due to Sylvain Conchon and Evelyne Contejean for their great work on Alt-Ergo [23] that helped us demonstrate our approach practically.

References

1. Floyd, R.W.: Assigning meanings to programs. In: Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society (1967)
2. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
3. Cousot, P.: Methods and logics for proving programs. In: Formal Models and Semantics. Handbook of Theoretical Computer Science, vol. B, pp. 841–993. Elsevier Science, Amsterdam (1990)
4. Clarke, E.: Programming language constructs for which it is impossible to obtain good Hoare axiom systems. Journal of the ACM 26(1), 129–147 (1979)
5. Apt, K.R.: Ten years of Hoare's logic: A survey—part I. ACM Transactions on Programming Languages and Systems 3(4), 431–483 (1981)
6. Damm, W., Josko, B.: A sound and relatively* complete axiomatization of Clarke's language L4. In: Clarke, E., Kozen, D. (eds.) Logic of Programs 1983. LNCS, vol. 164, pp. 161–175. Springer, Heidelberg (1984)
7. German, S., Clarke, E., Halpern, J.: Reasoning about procedures as parameters. In: Clarke, E., Kozen, D. (eds.) Logic of Programs 1983. LNCS, vol. 164, pp. 206–220. Springer, Heidelberg (1984)
8. Goerdt, A.: A Hoare calculus for functions defined by recursion on higher types. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 106–117. Springer, Heidelberg (1985)
9. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005)
10. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 234–245 (2002)
11. Marché, C., Paulin-Mohring, C., Urbain, X.: The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. Journal of Logic and Algebraic Programming 58(1–2), 89–106 (2004)
12. Filliâtre, J.C., Marché, C.: Multi-prover Verification of C Programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
13. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)

14. Hughes, J.: Why functional programming matters. Computer Journal 32(2), 98–107 (1989)
15. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Information and Computation 199(1–2), 200–227 (2005)
16. Detlefs, D.L., Leino, K.R.M., Nelson, G.: Wrestling with rep exposure. Research Report 156, SRC (July 1998)
17. Leino, K.R.M., Nelson, G.: Data abstraction and information hiding. ACM Transactions on Programming Languages and Systems 24(5), 491–553 (2002)
18. Fähndrich, M., DeLine, R.: Adoption and focus: practical linear types for imperative programming. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 13–24 (June 2002)
19. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 48–64 (October 1998)
20. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: IEEE Symposium on Logic in Computer Science (LICS), pp. 55–74 (2002)
21. Régis-Gianas, Y.: A Hoare logic for call-by-value functional programs: Prototype tool (January 2008), `http://pangolin-programming-language.googlecode.com`
22. The Coq development team: The Coq Proof Assistant (2006)
23. Conchon, S., Contejean, E.: The Alt-Ergo automatic theorem prover (2006), `http://alt-ergo.lri.fr/`
24. Kaplan, H., Tarjan, R.E.: Purely functional, real-time deques with catenation. Journal of the ACM 46(5), 577–603 (1999)
25. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences 17(3), 348–375 (1978)
26. Xi, H.: Dependent Types in Practical Programming. PhD thesis, Carnegie Mellon University (December 1998)
27. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system (October 2005)
28. Andrews, P.B.: An introduction to mathematical logic and type theory: to truth through proof. Academic Press, London (1986)
29. Paulin-Mohring, C.: Inductive definitions in the system Coq: rules and properties. Research Report RR1992-49, ENS Lyon (1992)
30. Werner, B.: Une Théorie des Constructions Inductives. PhD thesis, Université Paris 7 (1994)
31. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 237–247 (1993)
32. Régis-Gianas, Y.: Des types aux assertions logiques: preuve automatique ou assistée de propriétés sur les programmes fonctionnels. PhD thesis, Université Paris 7 (November 2007)
33. Nanevski, A., Ahmed, A., Morrisett, G., Birkedal, L.: Abstract Predicates and Mutable ADTs in Hoare Type Theory. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 189–204. Springer, Heidelberg (2007)
34. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM 52(3), 365–473 (2005)
35. Lescuyer, S.: Codage de la logique du premier ordre polymorphe multi-sortée dans la logique sans sortes. Master's thesis, Master Parisien de Recherche en Informatique (2006)
36. Kerber, M.: How to prove higher order theorems in first order logic. In: International Joint Conferences on Artificial Intelligence, pp. 137–142 (1991)

37. Altenkirch, T., McBride, C., McKinna, J.: Why dependent types matter (unpublished) (April 2005)
38. Sozeau, M.: Subset coercions in Coq. In: TYPES (2006)
39. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 224–235 (January 2003)
40. Filliâtre, J.C.: Backtracking iterators. In: ACM Workshop on ML (September 2006)
41. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Research Report 159, Compaq SRC (December 1998)
42. Filliâtre, J.C.: Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud (March 2003)
43. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 193–205 (2001)
44. Scott, D.S.: A type-theoretical alternative to ISWIM, CUCH, OWHY. Theoretical Computer Science 121(1–2), 411–440 (1993)
45. Milner, R.: Implementation and applications of Scott's logic for computable functions. In: Proceedings of the ACM conference on proving assertions about programs, pp. 1–6 (January 1972)
46. Agerholm, S.: A HOL basis for reasoning about functional programs. Technical Report RS-94-44, BRICS (December 1994)
47. Bartels, F., von Henke, F., Pfeifer, H., Rueß, H.: Mechanizing domain theory. Ulmer Informatik-Berichte 96-10, Universität Ulm, Fakultät für Informatik (1996)
48. Müller, O., Nipkow, T., von Oheimb, D., Slotosch, O.: HOLCF = HOL + LCF. Journal of Functional Programming 9, 191–223 (1999)
49. Longley, J., Pollack, R.: Reasoning About CBV Functional Programs in Isabelle/HOL. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 201–216. Springer, Heidelberg (2004)
50. Kieburtz, R.B.: P -logic: Property verification for Haskell programs. Draft (August 2002)
51. Hallgren, T., Hook, J., Jones, M.P., Kieburtz, R.: An overview of the Programatica toolset. In: High Confidence Software and Systems Conference (HCSS) (2004)
52. Abel, A., Benke, M., Bove, A., Hughes, J., Norell, U.: Verifying Haskell programs using constructive type theory. In: Haskell workshop, pp. 62–73 (September 2005)
53. Honda, K., Yoshida, N.: A compositional logic for polymorphic higher-order functions. In: International ACM Conference on Principles and Practice of Declarative Programming (PPDP), pp. 191–202 (August 2004)
54. Smith, S.F.: Partial Objects in Type Theory. PhD thesis, Cornell University (January 1989)
55. Xu, D.N.: Extended static checking for Haskell. In: Haskell workshop, pp. 48–59. ACM Press, New York (2006)
56. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 42–54 (January 2006)
57. Nanevski, A., Morrisett, G., Birkedal, L.: Polymorphism and separation in Hoare type theory. In: ACM International Conference on Functional Programming (ICFP), pp. 62–73 (September 2006)
58. Barnett, M., Naumann, D.A., Schulte, W., Sun, Q.: 99.44% pure: Useful abstractions in specifications. In: Formal Techniques for Java-like Programs (2004)

59. Gronski, J., Knowles, K., Tomb, A., Freund, S.N., Flanagan, C.: Sage: Hybrid checking for flexible specifications. In: Scheme and Functional Programming (September 2006)
60. Zenger, C.: Indexed types. Theoretical Computer Science 187(1–2), 147–165 (1997)
61. Davies, R.: Practical refinement-type checking. Technical Report CMU-CS-05-110, School of Computer Science, Carnegie Mellon University (May 2005)
62. Pottier, F., Régis-Gianas, Y.: Towards efficient, typed LR parsers. In: ACM Workshop on ML. Electronic Notes in Theoretical Computer Science, vol. 148(2), pp. 155–180 (March 2006)
63. Pottier, F., Régis-Gianas, Y.: Stratified type inference for generalized algebraic data types. In: ACM Symposium on Principles of Programming Languages (POPL) (January 2006)
64. Chen, C., Xi, H.: Combining programming with theorem proving. In: ACM International Conference on Functional Programming (ICFP) (September 2005)
65. Sheard, T.: Putting Curry-Howard to work. In: Haskell workshop (2005)
66. Westbrook, E., Stump, A., Wehrman, I.: A language-based approach to functionally correct imperative programming. In: ACM International Conference on Functional Programming (ICFP), pp. 268–279 (2005)

Synthesis of Optimal Control Policies for Some Infinite-State Transition Systems

Michel Sintzoff

Department of Computing Science and Engineering
Université catholique de Louvain
michel.sintzoff@uclouvain.be

Abstract. We develop a symbolic, logic-based technique for constructing optimal control policies in some transition systems where state spaces are large or infinite. These systems are presented as iterations of finite sets of guarded assignments which have costs. The optimality objective is to minimize the total costs of system executions reaching the set characterized by a given target predicate. Guards are predicates and control policies are expressed by tuples of guards. The optimal control policy refines the control policy of the given system. It is generated from the target predicate by an iteration based on backwards induction. This iterative procedure amounts to a variant of the symbolic algorithm generating the reachability precondition; the latter characterizes the states from which some system execution reaches the target set. The main difference is the introduction of greedy and cost-dependent iteration steps.

1 Introduction

Context. This paper concerns optimality in (discrete) transition systems, a.k.a. discrete-time dynamical systems. Such systems can be analyzed and synthesized using state-based algorithms if the state spaces are finite. When state spaces are infinite or large, symbolic algorithms may prove useful and are available in various cases. In particular, symbolic algorithms may generate reachability preconditions for transition systems [10], and optimal policies for variants of transition systems [7][8][9]. As to the optimal control of systems with no stochastic or continuous transitions, only state-based algorithms are known [6][23]. A symbolic generator of optimal policies for some of these systems is presented.

Approach. An algorithm over states is "state-based", or enumerative, if it handles states one by one. It is "symbolic", or set-based, if states are treated in finitely many, relatively large clusters. To design symbolic algorithms, we express transition systems by guarded-assignment programs [13]. Each assignment expresses a possibly infinite set of state transitions, which have one same given cost. Optimality means that all executions reach the target set at minimum total costs. Guards are predicates characterizing applicability domains of assignments, and control policies are expressed by tuples of guards. The proposed symbolic algorithm generates the optimal control policy. It simply amounts to a set-based form of the state-based greedy algorithm generating shortest paths [12].

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 336–359, 2008.

Contents of the paper. Section 2 introduces action programs. Section 3 recalls the state-based method of optimal control. The symbolic algorithm is developed in Sect. 4. It is put together and its complexity is analyzed in Sect. 5. Section 6 is a comparison with related work. Section 7 presents concluding remarks. Appendix A provides additional proofs. Appendix B contains an index of notations and an index of identifiers.

Writing Conventions. Easy proofs are sketched or understood. Bibliographical references are not complete. Universal quantifiers and domains of variables may be omitted if the context is clear. The following notations are adopted:

$\boldsymbol{B}$ is a tuple $(B_0, \cdots, B_n)$ of predicates, and $\bigvee \boldsymbol{B} \doteq \bigvee_{i=0}^{n} B_i$.
$\#Y$ is the cardinal of Y, $\mathbb{N}$ is the set of naturals, and $\mathbb{N}_\infty \doteq \mathbb{N} \cup \{\#\mathbb{N}\}$.
$Y \to Z$ (resp. $Y \hookrightarrow Z$) is the set of total (resp. partial) maps from Y to Z .
$[a, b] \doteq \{n \mid n \in N_\infty \wedge a \leq n \leq b\}$, and $Y_{[a,b]} \doteq \{y \mid y \in Y \wedge a \leq y \leq b\}$.
$Dom(f)$ is the definition domain of a map f .
$Rng(f)$ is the range of f, namely the image of $Dom(f)$ by f .
$P^{(\sup M)} \doteq \sup_{m \in M}\{P^{(m)}\}$ where $(P^{(m)})_{m \in M}$ is an ascending chain.
E_u^x results from substituting u for x in E, and u *satisfies* $P(x)$ iff $(P(x))_u^x$.
$P(x) \not\equiv \mathbf{false}$ is a *satisfiability expression* standing for $\exists x\!:\!P(x)$.

2 Action Programs, Transition Graphs and Reachability

An elementary, classical framework suffices for the results presented here.

Action programs are guarded-command programs where commands are assignments [3][13]. They determine transition graphs. A reachability precondition characterizes the states from which a target set is reachable using such a graph.

2.1 Action Programs and Control Policies

Action Programs. An *action program* S is a term **do** A **od** expressing the iteration of a set of N guarded assignments where N is any nonzero natural. The sub-term A has the form $A_0 \mathbin{[\!]} \cdots \mathbin{[\!]} A_{N-1}$ where each *action* A_i is a labelled guarded-assignment with a cost w_i. Namely, A_i is

$$B_i(x) \xrightarrow{i} x := f_i(x) \quad <w_i> \quad . \tag{1}$$

The variable x ranges over a set X, the *state space* of S. Each *label* i belongs to a finite alphabet I and labels one action. We use $I = [0, N-1]$ and write $A = [\!]_{i \in I} A_i$. The *action map* is $f_i \in X \hookrightarrow X$. The *(action control) guard* B_i is a predicate characterizing a subset of $Dom(f_i)$. The *action cost* w_i is a strictly positive integer. The maximum action cost is $M_w = \max_{i \in I}\{w_i\}$.

A *target predicate* Q is a distinguished predicate over X and characterizes the *target set.* An *(optimality) problem* is a pair (S, Q), where X is understood.

Comments. If $X = Z_0 \times \cdots \times Z_m$, an action A_i may use simultaneous assignments on the form $z_l := f_{l,i}(z_0, \cdots, z_m)$ where z_l ranges over Z_l ($l \in [0, m]$).

Action programs may be nondeterministic since $B_i \wedge B_j \not\equiv$ **false** is possible. Actually, they amount to recurrence inclusions such as $x^{(n+1)} \in \bigcup_{i \in I} \{f(x^{(n)}, i) \mid i \in \gamma(x^{(n)})\}$ [23]. The control set is I. The transition map $f \in X \times I \hookrightarrow X$ satisfies $f(x,i) = f_i(x)$. The cost of $f(x,i)$ is w_i. The stationary, or memoryless, control (multi)function $\gamma \in X \to 2^I$ satisfies $(i \in \gamma(x)) \equiv B_i(x)$. It is thus represented by the tuple $\boldsymbol{B} = (B_0, \cdots, B_{N-1})$. Clearly, $(\gamma(x) \neq \emptyset) \equiv (\bigvee \boldsymbol{B})(x)$.

Since action programs represent discrete-time systems, they may be obtained from specifications or programs; in fact, the nucleus of the B-method is built on guarded generalized assignments [1]. Of course, action programs may also abstract or approximate dense-time systems [2][25].

Control Policies. A *(symbolic control-)policy* $\boldsymbol{C}$ *(over a set* X*)* is a non-empty tuple of predicates (over X). Let I_m be an alphabet of m symbols, for any natural $m > 0$. An *m-ary policy* $\boldsymbol{C}$ is a tuple $(C_0, \cdots, C_{m-1})$ and denotes the map $c \in I_m \to 2^X$ such that $\bigwedge_{i \in I_m} (x \in c(i) \equiv C_i(x))$.

Let $\boldsymbol{C}$ and $\boldsymbol{C}'$ be two m-ary policies. They are *equal* iff $\bigwedge_{i \in I_m} (C_i \equiv C'_i)$. Moreover, $\boldsymbol{C}$ *refines* $\boldsymbol{C}'$ (or $\boldsymbol{C}'$ is *weaker* than $\boldsymbol{C}$) iff $\bigwedge_{i \in I_m} (C_i \Rightarrow C'_i)$. So, if $\boldsymbol{C}'$ is equal to $\boldsymbol{C}$ then $\boldsymbol{C}'$ is weaker than $\boldsymbol{C}$. A policy $\boldsymbol{C}$ is a *weakest* one in a set $\mathcal{P}$ of policies iff each policy in $\mathcal{P}$ which is weaker than $\boldsymbol{C}$ is equal to $\boldsymbol{C}$. It is the *unique weakest* policy iff any weakest one is equal to it.

Let S be an action program after (1). The *(inherent) policy of* S is the tuple $\boldsymbol{B}$ of action guards B_i in S. A *(refined) policy for* S is a policy which refines the inherent policy of S. Given a refined policy $\boldsymbol{C}$ for S, the action program S *controlled by* $\boldsymbol{C}$ is the result of replacing $\boldsymbol{B}$ by $\boldsymbol{C}$ in S and is denoted by $S{\downarrow}\boldsymbol{C}$. Formally, $S{\downarrow}\boldsymbol{C} \doteq$ **do** $[\!]_{i \in I}\ C_i(x) \xrightarrow{i} x := f_i(x) \quad < w_i > \quad$ **od** .

A policy may ensure reachability, optimality or termination (Sect. 3, Sect. 6). An algorithm which generates policies ensuring such a property is a *(policy) generator (for this property)*.

To sum up, control policies are first-class citizens. They may be constructed or refined. Weakest policies are often preferable: in general, if a control refinement introduces unnecessary determinism, it unduly restricts further refinements.

Running Example. We illustrate the main developments using one same example. The latter is highly simplified, but not trivial, and is defined as follows:

$$\begin{aligned} S &= \textbf{do}\ x \neq 0 \xrightarrow{0} x := 4x \quad <26> \quad [\!]\ \ x \geq 0 \xrightarrow{1} x := x{+}1 \quad <13> \quad \textbf{od} \\ X &= \mathbb{R}, \qquad Q(x) \ \equiv\ 8 \leq x \leq 10\ . \end{aligned} \tag{2}$$

The inherent policy $\boldsymbol{B}$ is $(x \neq 0, x \geq 0)$. Given the comments above, S serves as syntactic sugar for $x^{(n+1)} \in \{4x^{(n)} \mid 0 \in \gamma(x^{(n)})\} \cup \{x^{(n)} + 1 \mid 1 \in \gamma(x^{(n)}))\}$ where $I = \{0, 1\}$ and $((0 \in \gamma(x)), (1 \in \gamma(x)) = \boldsymbol{B}$.

The constants in (2) are chosen so as to allow for lightweight technical explanations. The alphabet $\{0, 1\}$ may be replaced by $\{PowerUp, Steady\}$. The example could be made substantial by enriching the state space, adding actions, and using pertinent action-maps.

A policy is optimal iff each execution that it allows does reach the target set and has a minimum total cost for a given initial state. A unique weakest optimal policy $\boldsymbol{C}$ for (2) exists and is constructed by a symbolic generator (Sect. 4.3). The optimally controlled program is then

$$\begin{aligned} S{\downarrow}\boldsymbol{C} = \mathbf{do}\ & (0.5 \le x \le 0.625) \vee (1.5 < x \le 2.5) \xrightarrow{0} x := 4x \qquad <\!26\!> \\ [\!]\ & (0 \le x \le 0.5) \vee (0.625 < x \le 1.5) \\ & \qquad \vee (2.5 < x < 8) \xrightarrow{1} x := x + 1 \quad <\!13\!>\ \mathbf{od}\ . \end{aligned}$$

2.2 Graphs, Paths and Their Costs

Graphs and Paths. Let S be an action program after (1). The *transition graph* G_S is the labelled, weighted graph (X, I, E_S, w_S) where the set $E_S \subseteq X \times I \times X$ of edges is $\bigcup_{x \in X, i \in I} \{(x, i, f_i(x)) \mid B_i(x)\}$ and the weight function $w_S \in I \to X$ is defined by $\forall i \in I : w_S(i) = w_i$.

If desired, the target set may become a source set by inverting the graph G_S, i.e. by substituting (x', i, x) for each edge (x, i, x') in E_S.

An*(S-)path p* is an alternating sequence of states and compatible labels which contains at least one state and may be infinite; namely, $p = x$ where $x \in X$ or $p = (x_0, i_1, x_1, \cdots, x_{k-1}, i_k, x_k, \cdots)$, viz. $p = x_0 i_1 x_1 \cdots x_{k-1} i_k x_k \cdots$, where each tuple (x_{k-1}, i_k, x_k) is an edge in E_S.

A path is a path *from* $x \in X$ iff its first state is x. A path *reaches* a set $Y \subset X$ iff it is finite and its last state belongs to Y. A path reaches a predicate P characterizing Y iff it reaches Y. A path is *total*, viz. maximal, iff either it is infinite or it reaches $\neg \bigvee \boldsymbol{B}$. So, a finite path is not total iff it can be extended into a strictly longer path.

The set of (S-)paths is denoted by $\mathit{Paths}.S$. The set of paths from x is denoted by $\mathit{Paths}.S.x$. The set of paths from x which reach P is denoted by $\mathit{Paths}.S.x.P$. The set of total paths in $\mathit{Paths}.S.x$ (resp. $\mathit{Paths}.S.x.P$) is denoted by $\mathit{TotalPaths}.S.x$ (resp. $\mathit{TotalPaths}.S.x.P$).

Note. Let Inv be a predicate satisfied by all x and x' such that $\exists i \in I : (x, i, x') \in E_S$. Any guard B_i may be replaced by a guard B'_i satisfying $\mathit{Inv} \wedge B'_i \equiv B_i$ [13]. For brevity, this simplification is not taken into account.

Path Costs. Let $s.s'$ denote the concatenation of sequences s and s'. The *(chop path-) concatenation* of the paths $p = s.x$ and $p' = x.s'$ is $p \frown p' \doteq s.x.s'$. Thus, one same state x ends p and begins p', and it is not repeated in $p \frown p'$.

The function $\mathit{cost} \in \mathit{Paths}.S \to \mathbb{N}_\infty$ yields the sum of the costs of the actions used in a path. Likewise, $\mathit{nb}_{\mathit{edg}} \in \mathit{Paths}.S \to \mathbb{N}_\infty$ yields the number of edge occurrences in a path. Namely, for $x, x' \in X$ and $i \in I$,

$$\begin{aligned} \mathit{cost}(x) &= 0, & \mathit{cost}((x, i, x') \frown p) &= w_i + \mathit{cost}(p) \\ \mathit{nb}_{\mathit{edg}}(x) &= 0, & \mathit{nb}_{\mathit{edg}}((x, i, x') \frown p) &= 1 \ + \mathit{nb}_{\mathit{edg}}(p)\ . \end{aligned}$$

Since $\bigwedge_{i \in I}(1 \leq w_i \leq M_w)$ (Sect. 2.1),

$$\forall p \in \mathit{Paths}.S : \quad \mathit{nb}_{\mathit{edg}}(p) \leq \mathit{cost}(p) \leq \mathit{nb}_{\mathit{edg}}(p) \times M_w \ . \tag{3}$$

2.3 Reachability Precondition

Given an action program S and a target predicate Q, the *reachability precondition* $\mathit{pre}.S.Q$ is defined by $(\mathit{pre}.S.Q)(x) \equiv (\mathit{Paths}.S.x.Q \neq \emptyset)$. So, a state x satisfies $\mathit{pre}.S.Q$ iff some S-path from x reaches Q. The reachability precondition can be derived iteratively [10][11]: for $x \in X$ and $i \in I$,

$$(\mathit{pre}.S.P)(x) \equiv \bigvee_{n \in \mathbb{N}}((\mathit{pre}.A)^n.P)(x) \tag{4}$$

$$(\mathit{pre}.A.P)(x) \equiv \bigvee_{i \in I}(\mathit{pre}.A_i.P)(x), \qquad (\mathit{pre}.A_i.P)(x) \equiv B_i(x) \wedge P^x_{f_i(x)} \tag{5}$$

where $P(x)$ is a predicate over X, $(\mathit{pre}.A)^{n+1}.P \equiv (\mathit{pre}.A)^n.(\mathit{pre}.A.P)$ and $(\mathit{pre}.A)^0.P \equiv P$. Thus, $\mathit{pre}.S.Q$ characterizes a transitive backwards closure. Moreover, it is the least fixpoint of $\lambda w : (Q \vee \mathit{pre}.A.w)$. Hence,

$$\mathit{pre}.A.(\mathit{pre}.S.Q) \Rightarrow \mathit{pre}.S.Q \ . \tag{6}$$

For any refined policy $\boldsymbol{C}$ for S, $\mathit{Paths}.(S{\downarrow}\boldsymbol{C}) \subseteq \mathit{Paths}.S$. Hence, $\mathit{pre}.(S{\downarrow}\boldsymbol{C}).P \Rightarrow \mathit{pre}.S.P$. In short, a policy refinement determines a program refinement [1][3].

Notation. We sometimes write $\breve{S}_Q$ for $\mathit{pre}.S.Q$.

3 State-Based Synthesis of Optimal Policies

A few basic results about optimal control are summarized in terms of the above framework [5][6][21][23].

Let (S, Q) be an optimality problem (Sect. 2.1) and let $\boldsymbol{B}$ be the inherent policy of S. Recall that $\bigvee \boldsymbol{B}$ stands for $\bigvee_{i=0}^{n} B_i$.

Optimality Precondition. The *(state-to-)value function* $V \in X \hookrightarrow \mathbb{N}$, a.k.a. cost-to-go map, yields minimum path-costs: for all x satisfying $\mathit{pre}.S.Q$,

$$V(x) = \min\{\mathit{cost}(p) \mid p \in \mathit{Paths}.S.x.Q\} \ . \tag{7}$$

An *(optimal) value* is a path cost in $\mathit{Rng}(V)$.

For any $x \in X$, an S-path from x is *optimal* iff it reaches Q, has a cost equal to $V(x)$, and is total. Each optimal path from x must be total lest it be extended into a non-optimal path p', viz. $V(x) < \mathit{cost}(p')$: clearly, any optimal system must exclude such a path p'. The set of optimal S-paths from x is thus $\mathit{OptimPaths}.S.x.Q \doteq \{p \mid (p \in \mathit{TotalPaths}.S.x.Q) \wedge (\mathit{cost}(p) = V(x))\}$.

The *optimality precondition* $\mathit{optim}.S.Q$ is satisfied by x iff some S-path from x reaches Q and each total S-path from x reaches Q at a minimum total cost. So, $(\mathit{optim}.S.Q)(x) \equiv (\mathit{pre}.S.Q)(x) \wedge (\mathit{TotalPaths}.S.x = \mathit{OptimPaths}.S.x.Q)$.

The action program S is *optimal* (or ensures optimality) with respect to Q iff $Q \vee \bigvee \boldsymbol{B} \equiv \mathit{optim}.S.Q$. The latter condition is equivalent to $\bigvee \boldsymbol{B} \Rightarrow \mathit{optim}.S.Q$. Indeed, $Q \Rightarrow \mathit{optim}.S.Q$ is immediate while $Q \vee \bigvee \boldsymbol{B} \Leftarrow \mathit{optim}.S.Q$ follows from $Q \vee \bigvee \boldsymbol{B} \Leftarrow \mathit{pre}.S.Q$ and $\mathit{pre}.S.Q \Leftarrow \mathit{optim}.S.Q$.

Optimal Control. *An optimal policy* for (S, Q) is a refined policy $\boldsymbol{C}'$ for S such that $Q \vee \bigvee \boldsymbol{C}' \equiv optim.(S{\downarrow}\boldsymbol{C}').Q$, viz. $S{\downarrow}\boldsymbol{C}'$ ensures optimality.

The *weakest optimal policy* $\boldsymbol{C}$ for (S, Q) is the refined policy for S such that

$$\forall x \in X : (pre.S.Q)(x) \Rightarrow (TotalPaths.(S{\downarrow}\boldsymbol{C}).x = OptimPaths.S.x.Q) \ . \quad (8)$$

So, a total S-path is rejected by $\boldsymbol{C}$ iff it is not optimal. If an optimal policy $\boldsymbol{C}'$ is not equal to $\boldsymbol{C}$, it satisfies (8) where inclusion replaces equality, viz. $\boldsymbol{C}'$ actually rejects some optimal S-paths. Hence, $\boldsymbol{C}$ is weaker than $\boldsymbol{C}'$ and is unique. If the value function V can be computed then $\boldsymbol{C}$ is easily constructed as follows.

A preliminary property results from (6) and (7); recall that $\breve{S}_Q \equiv pre.S.Q$:

$$\forall j \in I, \forall x \in X : (pre.A_j.\breve{S}_Q)(x) \Rightarrow (V(x) \leq w_j + V(f_j(x))) \ . \quad (9)$$

The weakest optimal policy $\boldsymbol{C}$ is then determined by

$$\forall i{\in}I, x \in X : \ C_i(x) \equiv (pre.A_i.\breve{S}_Q)(x) \wedge (V(x) - V(f_i(x)) = w_i) \ . \quad (10)$$

In fact, the equality $V(x) = \min_{j \in I}\{w_j + V(f_j(x)) \mid (pre.A_j.\breve{S}_Q)(x)\}$ formalizes the principle of optimality and is implied by (9) and (10). So, a state x satisfies the optimal guard C_i iff the choice of action A_i is optimal for x.

Since $\boldsymbol{C}$ is the weakest optimal policy and $(V(x) > 0) \Rightarrow \neg(V(x) = 0)$,

$$Q \vee \bigvee \boldsymbol{C} \equiv pre.S.Q \, , \qquad \bigvee \boldsymbol{C} \Rightarrow \neg Q \ . \quad (11)$$

If $\boldsymbol{C}$ is equal to the inherent policy of S then S is optimal with respect to Q.

Convention. An optimal policy for action programs is the weakest one unless otherwise stated.

State-Based Generator. *Assume the state space in* (S, Q) *is finite. A state-based generator for optimality is obtained by the sequential composition of the computation of the value function (7) and the unfolding of the equivalences (10).*

The complexity of this generator is polynomial in $\#X$: shortest paths are produced by efficient algorithms [5][12][14], and they determine V.

4 Development of a Symbolic Generator for Optimality

Our intention is to start from a simple state-based generator, and we choose the greedy algorithm for shortest paths [12]. Each iteration step in the latter generates one state, if any, not yet generated and having the last computed value; otherwise, it computes the next higher value. It is easy, at least in principle, to transform this state-based algorithm into a symbolic one: each symbolic iteration step computes the next higher value and generates predicates (called "guard strata") characterizing states which have this value and satisfy an optimal guard. Clearly, the number of values should be finite. Values are ranked by naturals called levels. At level 0, the value is zero and the basis predicate is the target predicate. The step at a level $n > 0$ constructs one guard stratum for each optimal guard, by a greedy computation which guarantees optimality.

Let (S, Q) be an optimality problem. For the running example, see Sect. 4.3.

4.1 Stratification of Optimal Guards

Let $\boldsymbol{C} = (C_0, \cdots, C_{N-1})$ be the (weakest) optimal policy for (S, Q). Each guard C_i is decomposed into the guard strata $F_i^{(m)}$, one for each value level m. Namely, $C_i \equiv \bigvee_{m \in X_L} F_i^{(m)}$ where X_L is the set of levels. Thus, each guard C_i is the limit of the growing sub-guards $C_i^{(n)} \equiv \bigvee_{m \in [0,n]} F_i^{(m)}$, for $n \in X_L$.

Levels of Values and of States. If $\mathit{Rng}(V)$ is finite then the set X_L of *(value) levels* is $[0, \#\mathit{Rng}(V) - 1]$. Otherwise, it is $\mathbb{N}$. The *level-to-value bijection* $V_L \in X_L \to \mathit{Rng}(V)$ arranges values in increasing order: for $n \in X_L \backslash \{0\}$,

$$V_L(0) = 0, \quad V_L(n) = \min_{x:(pre.S.Q)(x)} \{V(x) \mid V(x) > V_L(n-1)\} \ . \tag{12}$$

The value at level m is $V_L(m)$. The level of a value $v \in \mathit{Rng}(V)$ is $V_L^{-1}(v)$. The level of a state x satisfying $pre.S.Q$ is the level of its value, i.e. $V_L^{-1}(V(x))$. So, the *(state-to-)level function* $L \in X \hookrightarrow X_L$ is given by $L = V_L^{-1} \circ V$. Clearly,

$$V = V_L \circ L, \qquad V(x) \geq L(x), \qquad V(x) > V(x') \equiv L(x) > L(x') \tag{13}$$

where x and x' satisfy $pre.S.Q$.

The *optimality radius* $\rho \in \mathbb{N}_\infty$ is the number of nonzero values:

$$\rho = \sup X_L = \sup \mathit{Dom}(V_L) = \sup \mathit{Rng}(L) \ . \tag{14}$$

The finiteness of this number is crucial (Sect. 6.1). Of course, $\rho \leq \#X$.

Optimality is bounded iff the optimality radius is finite.

Stratification. The *sub-policy* $\boldsymbol{C}^{(n)} = (C_0^{(n)}, \cdots, C_{N-1}^{(n)})$ approximates $\boldsymbol{C}$ from below, by restricting it to the states having a level at most n. So, each *sub-guard* $C_i^{(n)}$ characterizes a level set: for all $n \in X_L$, $i \in I$, and $x \in X$,

$$C_i^{(n)}(x) \ \equiv \ C_i(x) \wedge (L(x) \leq n) \ . \tag{15}$$

The policy strata $\boldsymbol{F}^{(n)} = (F_0^{(n)}, \cdots, F_{N-1}^{(n)})$ stratify the optimal policy $\boldsymbol{C}$. So, the *guard strata* $F_i^{(n)}$ stratify the optimal guard C_i. Namely, each $F_i^{(n)}$ restricts C_i to the states having the level n; it is thus the fringe (or front) of $C_i^{(n)}$:

$$F_i^{(n)}(x) \ \equiv \ C_i(x) \wedge (L(x) = n) \ . \tag{16}$$

Likewise, the auxiliary *sub-domains* $D^{(n)}$ and *domain strata* $H^{(n)}$ respectively approximate and stratify the reachability precondition:

$$D^{(n)}(x) \equiv (pre.S.Q)(x) \wedge (L(x) \leq n) \tag{17}$$

$$H^{(n)}(x) \equiv (pre.S.Q)(x) \wedge (L(x) = n) \ . \tag{18}$$

So, the domain strata $H^{(n)}$ characterize equivalence classes where V is constant.

4.2 Transformations into Computation Formulas

A *computation formula* is a formula which can be evaluated effectively. It yields a number, a Boolean value, or a term which formalizes a predicate introduced in Sect. 4.1. A key formula serves to generate guard strata.

Computation Formulas for Sub-Guards and Sub-Domains. Given (6) and (14), if $\mathit{Rng}(V)$ is finite then a computation formula for ρ is provided by

$$\rho = \max_{n\in\mathbb{N}}\{n+1 \mid \neg D^{(n)} \wedge \mathit{pre}.A.D^{(n)} \not\equiv \mathbf{false}\} \ . \qquad (19)$$

The computation formulas for $F_i^{(0)}, C_i$ and for $C_i^{(m)}, H^{(m)}, D^{(m)}$ $(m \in [0,\rho])$ are given by the following equivalences, where $i \in I$ and $n \in [1,\rho]$:

$$F_i^{(0)} \equiv \mathbf{false}, \quad C_i^{(0)} \equiv F_i^{(0)}, \quad C_i^{(n)} \equiv C_i^{(n-1)} \vee F_i^{(n)}, \quad C_i \equiv C_i^{(\rho)} \qquad (20)$$

$$H^{(0)} \equiv Q, \quad D^{(0)} \equiv H^{(0)}, \quad H^{(n)} \equiv \textstyle\bigvee_{i\in I} F_i^{(n)} \qquad (21)$$

$$D^{(n)} \equiv D^{(n-1)} \vee H^{(n)} \ . \qquad (22)$$

The computation formula for C_i follows from (14) and (15). The one for $H^{(n)}$ results from (11), (16), and (18). The other proofs are obvious.

Greedy Computation Formula for Guard Strata. The formula for computing guard strata implements the greedy principle as follows.

In (16), we unfold C_i using (10) and replace the state-based map V by the level-based bijection V_L. Actually, given (13) and (18), $V(x) - V(f_i(x)) = V_L(n) - V_L(m)$ if $L(x) = n$ and $L(f_i(x)) = m$ or if $H^{(n)}(x)$ and $H^{(m)}(f_i(x))$.

This can be formalized using level-based guards $\mathit{best}_i \in X_L \to \mathit{Bool}$ and maps $g_i \in X_L \hookrightarrow X_L$, defined as follows: for $i \in I$ and $n \in [1,\rho]$,

$$\mathit{best}_i(n) \equiv (V_L(n) - w_i \in \mathit{Rng}(V_L)) \qquad (23)$$

$$\mathit{best}_i(n) \Rightarrow (V_L(n) - w_i = V_L(\,g_i(n))) \ . \qquad (24)$$

So, if $\mathit{best}_i(n)$ is true, the solution for m of the equation $V_L(n) - V_L(m) = w_i$ exists and is equal to $g_i(n)$. Two properties are derived (Appendix A): for $i \in I$, $n \in [1,\rho]$, and $x \in X$,

$$\text{(Greediness)} \qquad \neg D^{(n-1)} \wedge \mathit{best}_i(n) \wedge \mathit{pre}.A_i.H^{(g_i(n))} \Rightarrow H^{(n)} \qquad (25)$$

$$\text{(Abstraction)} \quad C_i(x) \Rightarrow (\mathit{best}_i \circ L)(x) \wedge ((L \circ f_i)(x) = (g_i \circ L)(x)) \ . \qquad (26)$$

In (25), conjunctions are evaluated from left to right. Property (26) asserts that, for all states x satisfying C_i, the state-to-level homomorphism L abstracts the action map f_i into the level-based map g_i [19].

We then obtain the key formula: for $i \in I$ and $n \in [1,n]$,

$$F_i^{(n)} \equiv \neg D^{(n-1)} \wedge \mathit{best}_i(n) \wedge \mathit{pre}.A_i.H^{(g_i(n))} \ . \qquad (27)$$

Proof.

$$
\begin{array}{lll}
 & F_i^{(n)}(x) & \\
\equiv & C_i(x) \wedge (L(x) = n) & [(16)] \\
\equiv & C_i(x) \wedge (pre.S.Q)(x) \wedge best_i(n) & \\
 & \qquad \wedge\, (L(f_i(x)) = g_i(n)) \wedge (L(x) = n) & [(11),\ (26)] \\
\equiv & B_i(x) \wedge (pre.S.Q)(f_i(x)) \wedge (pre.S.Q)(x) \wedge best_i(n) & \\
 & \qquad \wedge\, (L(f_i(x)) = g_i(n)) \wedge (L(x) = n) & [(5), (10), (13), (24)] \\
\equiv & B_i(x) \wedge H^{(g_i(n))}(f_i(x)) \wedge H^{(n)}(x) \wedge best_i(n) & [(18)\text{ twice}] \\
\equiv & pre.A_i.(H^{(g_i(n))}(x)) \wedge H^{(n)}(x) \wedge best_i(n) & [(5)] \\
\equiv & \neg D^{(n-1)}(x) \wedge best_i(n) \wedge pre.A_i.(H^{(g_i(n))}(x)) & [(17),\ (18),\ (25)]
\end{array}
$$

□

A computation step using (27) is asynchronous since usually $g_i(n) \neq g_j(n)$.

Level-based actions $A_i^L = (best_i(n) \xrightarrow{i} n := g_i(n) < w_i >)$ could make up a system S_L abstracting S. Accordingly, $n, best_i, g_i$, and (27) could respectively be written x_L, C_i^L, f_i^L, and $F_i^{(x_L)} \equiv \neg D^{(x_L-1)} \wedge pre.A_i^L.(pre.A_i.H^{(x_L)})$ where $F_i^{(x_L)}(x)$ could become $F_i(x_L, x)$ and likewise for the other iterates.

Sub-Computation Formulas. For $i \in I$ and $n \in [1, \rho]$,

$$V_L(n) = \min_{\substack{i\,\in\, I \\ m\,\in\,[0,\,n-1]}} \{w_i + V_L(m) \mid \neg D^{(n-1)} \wedge pre.A_i.H^{(m)} \not\equiv \textbf{false}\} \tag{28}$$

$$best_i(n) \equiv \bigvee_{m\in[0,n-1]} V_L(n) - w_i = V_L(m) \tag{29}$$

$$g_i(n) = \min_{m\in[0,n-1]} \{m \mid V_L(n) - w_i = V_L(m)\} \quad \text{if } best_i(n)\ . \tag{30}$$

The proof of (28) is found in Appendix A. The proofs of (29) and (30) are easy. The iterates $V_L(m)$ and $H^{(m)}$ in (28) may be recorded in vectors of length ρ. The operator min in (30) serves to compute $V_L^{-1}(V_L(n) - w_i)$.

Implicit Computations. The terms formalizing the predicates from Sect. 4.1 are treated as syntactical expressions. They may be simplified into equivalent ones. Satisfiability expressions, variables such as $F_i^{(n)}$ in (20), integer-valued expressions as in (28), and applications of $best_i$, g_i, and pre are all evaluated.

4.3 Running Example (Continued)

Recall the example (2). For the first three levels, the formulas in Sect. 4.2 yield

$$
\begin{array}{ll}
V_L(0) = 0 & \\
\boldsymbol{F}^{(0)} = (\textbf{false}, \textbf{false}), & \boldsymbol{C}^{(0)} = (\textbf{false},\ \textbf{false}) \\
H^{(0)} \equiv 8 \leq x \leq 10, & D^{(0)} \equiv 8 \leq x \leq 10 \\
V_L(1) = 13 & \\
\boldsymbol{F}^{(1)} = (\textbf{false},\ (7 \leq x < 8)), & \boldsymbol{C}^{(1)} = (\textbf{false},\ (7 \leq x < 8)) \\
H^{(1)} \equiv 7 \leq x < 8, & D^{(1)} \equiv 7 \leq x \leq 10
\end{array}
$$

$$\begin{array}{ll} V_L(2) = 26 & \\ \boldsymbol{F}^{(2)} = ((2 \le x \le 2.5),\ (6 \le x < 7)), & \boldsymbol{C}^{(2)} = ((2 \le x \le 2.5),\ (6 \le x < 8)) \\ H^{(2)} \equiv (2 \le x \le 2.5) \vee (6 \le x < 7), & D^{(2)} \equiv (2 \le x \le 2.5) \vee (6 \le x \le 10)\ . \end{array}$$

Of course, $\boldsymbol{F}^{(n)} = (F_0^{(n)}, F_1^{(n)})$ and $\boldsymbol{C}^{(n)} = (C_0^{(n)}, C_1^{(n)})$. The value at level 1, for instance, is computed easily:

$$\begin{array}{lr} V_L(1) = \min_{i \in \{0,1\}}\{w_i + V_L(0) \mid \neg D^{(0)} \wedge pre.A_i.H^{(0)} \not\equiv \textbf{false}\} & [(28)] \\ \quad = \min\{\ \ \{26 \mid \exists x \in \mathrm{I\!R} : 2 \le x \le 2.5\} & [(5);\text{Logic and} \\ \qquad\qquad \cup \{13 \mid \exists x \in \mathrm{I\!R} : 7 \le x < \ \ 8\ \}\ \} & \text{Arithmetic}] \\ \quad = 13\ . & [\text{Arithmetic}] \end{array}$$

For $n = 2$, the guard stratum in C_0 is obtained as follows. The solution for m of $V_L(m) = V_L(2) - w_0 = 26 - 26$ exists and is zero (29, 30). So, $g_0(2) = 0$ and

$$\begin{array}{lr} F_0^{(2)}(x) \equiv \neg D^{(1)}(x) \wedge best_0(2) \wedge (pre.A_0.H^{(g_0(2))})(x) & [(27)] \\ \qquad\quad \equiv \neg D^{(1)}(x) \wedge (pre.A_0.H^{(0)})(x) & [g_0(2) = 0] \\ \qquad\quad \equiv \neg(7 \le x \le 10) \wedge (x \ne 0) \wedge (8 \le 4x \le 10) & [(5)] \\ \qquad\quad \equiv 2 \le x \le 2.5\ . & [\text{Logic, Arithmetic}] \end{array}$$

The optimality radius ρ, or maximum value-level, is 6. The maximum value $V_L(6)$ is 78. The weakest optimal policy $\boldsymbol{C}$ is the pair (C_0, C_1) where

$$\begin{array}{l} C_0(x) \equiv C_0^{(6)}(x) \equiv (0.5 \le x \le 0.625) \vee (1.5 < x \le 2.5) \\ C_1(x) \equiv C_1^{(6)}(x) \equiv (0 \le x \le 0.5) \vee (0.625 < x \le 1.5) \vee (2.5 < x < 8)\ . \end{array} \tag{31}$$

5 Symbolic Synthesis of Optimal Policies

5.1 Symbolic Generator for Optimality

Schema. Assume that the reachability radius ρ is finite (14) and that the satisfiability expressions in (28) can be evaluated. The optimal policy can then be constructed by a finite iteration where the computation formulas (Sect. 4.2) are used:

$$\begin{array}{lll} \boldsymbol{C}^{(0)} & := (\textbf{false}, \cdots, \textbf{false}), & \text{since } V_L(0) = 0 \text{ given (12)} \\ \boldsymbol{C}^{(n)} & := (C_0^{(n)}, \cdots, C_{N-1}^{(n)}), & \text{for } n \in [1, \rho] \\ \boldsymbol{C} & := \ \boldsymbol{C}^{(\rho)}\ . & \end{array}$$

The proofs in Sect. 4.2 entail the correctness of the schema.

Resulting Generator. It is obtained by unfolding the above schema. The sub-policy $\boldsymbol{C}^{(n)}$ is the tuple $(C_0^{(n)}, \cdots, C_{N-1}^{(n)})$ for $n \in [0, \rho]$.

Input: An optimality problem (S, Q) is given (Sect. 2.1).
Precondition: Optimality is bounded (Sect. 4.1) and the satisfiability expressions in (28) are decidable.
Postcondition: The result $\boldsymbol{C}$ is the weakest optimal policy (Sect. 3).
Invariant: Each equivalence from (15) to (18) is invariant.

Algorithm *OptimPol* $\doteq$

```
begin  $n := 0$; for $i \in I : (F_i^{(0)} := \mathbf{false};\ C_i^{(0)} := F_i^{(0)})$;   (32)
    $V_L(0) := 0;\ \ H^{(0)} := Q;\ \ D^{(0)} := H^{(0)}$;   (33)
  while $\neg D^{(n)} \wedge pre.A.D^{(n)} \not\equiv \mathbf{false}$ do $n := n+1$;   (34)
    $V_L(n) := \min_{\substack{i \in I \\ m \in [0,n-1]}} \left\{ \begin{array}{l} w_i + V_L(m) \mid \\ \neg D^{(n-1)} \wedge pre.A_i.H^{(m)} \not\equiv \mathbf{false} \end{array} \right\}$;   (35)
    for $i \in I : (\ F_i^{(n)} := \neg D^{(n-1)} \wedge best_i(n) \wedge pre.A_i.H^{(g_i(n))}$;   (36)
            $C_i^{(n)} := C_i^{(n-1)} \vee F_i^{(n)}\ )$ ;   (37)
    $H^{(n)} := \bigvee_{i \in I} F_i^{(n)};\quad D^{(n)} := D^{(n-1)} \vee H^{(n)}$        od ;   (38)
  $\rho := n;\ \boldsymbol{C} := \boldsymbol{C}^{(\rho)}$        end .   (39)
```

Correctness. The correctness of *OptimPol* follows from that of the above schema and from a sequential ordering of computations thanks to which the construction of each iterate precedes its use.

Comments. The set of terms generated by *OptimPol* is finite since ρ is finite.

The satisfiability expression in (34) is equivalent to the finite disjunction $\bigvee_{i \in I, m \in [0,n]} (\neg D^{(n)} \wedge pre.A_i.H^{(m)} \not\equiv \mathbf{false})$ of satisfiability expressions from (28). So, it need not be mentioned in the precondition of the algorithm.

The conjuncts in the latter precondition can be established as follows. For one, optimality is bounded in particular families of problems (44). For another, the satisfiability expressions are decidable in restricted formal theories. Of course, *OptimPol* may fail to terminate if its precondition is not guaranteed.

Notation. The term *OptimPol*(S, Q) refers to *OptimPol* with input (S, Q).

5.2 Complexity

Clearly, the complexity of the algorithm *OptimPol* is polynomial with respect to the number of optimal values, the number of actions, and the complexity of evaluating the satisfiability expression in (35), i.e. in (28).

The following notations are used to express complexity. Given an algorithm $\mathcal{A}$, its complexity is denoted by $T(\mathcal{A})$ while $T_{\text{SAT}}(\mathcal{A})$ denotes the complexity of evaluating the satisfiability expressions in $\mathcal{A}$. In addition, $f \in Poly(g)$ stands for $\exists k \in \mathbb{N} : f \in \mathcal{O}(g^k)$. Thus,

$$T(OptimPol) \in Poly(\rho + N + T_{\text{SAT}}(OptimPol)) \ . \tag{40}$$

Therefore, the symbolic generator *OptimPol* is less efficient than the state-based one (Sect. 3) for all problems (S, Q) on a finite state space X such that

$N + T_{\text{SAT}}(\mathit{OptimPol}(S, Q)) \notin \mathit{Poly}(\#X)$. For instance, if the evaluation of a satisfiability expression is *NP*-hard then *OptimPol* may well be less efficient than the state-based generator. If appropriate, the finite transition graph G_S can be extended with the set of edges $\{(x, N, x_q) \mid Q(x)\}$ where x_q is a distinguished new state; the target set then reduces to the singleton $\{x_q\}$.

There are two classes of problems (S, Q) for which *OptimPol* appears useful:

Class I. The set X is infinite and the precondition of *OptimPol* is satisfied by (S, Q). The state-based generator is not intended for this class.

Class II. The set X is finite and $T(\mathit{OptimPol}(S, Q)) \in \mathit{Poly}(\log \#X)$. Then *OptimPol* is exponentially more efficient than the state-based generator.

5.3 Running Example (Continued)

Illustration of Class I above. Since the example in (2) satisfies the precondition of *OptimPol* (Sect. 4.3), it belongs to Class I. The optimal policy $\boldsymbol{C}$ (31) is slightly nondeterministic, given $C_0 \wedge C_1 \equiv (x = 0.5)$. There are two optimal paths from 0.5, namely $(0.5, 0, 2, 0, 8)$ and $(0.5, 1, 1.5, 1, 2.5, 0, 10)$. Their cost is $52 = V_L(4)$. The reachability precondition $\mathit{pre}.S.Q$ is $0 \leq x \leq 10$. The potential complexity of the decomposition of $\neg Q \wedge \mathit{pre}.S.Q$ into C_0 and C_1 (11, 31) may be glimpsed from this highly simplified illustration.

In the case of S and Q in (2), $T(\mathit{OptimPol}(S, Q))$ is low since the predicates $D^{(n-1)}$ and $H^{(m)}$ (35) are little disjunctions of simple intervals (Sect. 4.3).

Illustration of Class II above. Let us approximate the state space $X = \mathbb{R}$ in (2) by a set $X_m \subseteq \mathbb{Q}_{[-50,+50]}$ such that $T(\mathit{OptimPol}(S, Q)) \in \mathit{Poly}(\log \#X_m)$ and $\{2^m \times x \mid x \in X_m\} = \mathbb{Z}_{[-2^m \times 50, +2^m \times 50 - 1]}$ for a natural $m \geq 10$. Thus, $\#X_m = 2^m \times 100$ and $0.625 \in X_m$. The choice of m is not indifferent because the optimal policy (31) is not preserved if $m = 1$ for instance.

Let (S_m, Q_m) be (S, Q) where X_m replaces X. Since X_m is a sufficiently precise approximation of X, $\mathit{OptimPol}(S_m, Q_m)$ faithfully mimics $\mathit{OptimPol}(S, Q)$. In particular, ρ in $\mathit{OptimPol}(S_m, Q_m)$ equals ρ in $\mathit{OptimPol}(S, Q)$, and each iterate in $\mathit{OptimPol}(S_m, Q_m)$ is the corresponding one in $\mathit{OptimPol}(S, Q)$ where X_m replaces X. Hence, $T(\mathit{OptimPol}(S_m, Q_m)) \in \mathit{Poly}(T(\mathit{OptimPol}(S, Q)))$. Hence, $T(\mathit{OptimPol}(S_m, Q_m)) \in \mathit{Poly}(\log \#X_m)$ by definition of X_m. Hence, the problem (S_m, Q_m) belongs to Class II. In this case, for any $m' > m$, $(S_{m'}, Q_{m'})$ also belongs to Class II: $T(\mathit{OptimPol}(S_{m'}, Q_{m'}) \in \mathit{Poly}(\log \#X_{m'})$.

Obviously, $\log \#X_m$ grows polynomially as $\#X_m$ grows exponentially. Consider e.g. $m' = m^h + m$ where $h \in \mathbb{N}$. The approximation is then more precise by m^h orders of magnitude, viz. $\#X_{m'} = 2^{m^h} \times \#X_m$. As a consequence, $\log \#X_{m'} = m^h + \log \#X_m \in \mathit{Poly}(\log \#X_m)$ and $T(\mathit{OptimPol}(S_{m'}, Q_{m'})) \in \mathit{Poly}(T(\mathit{OptimPol}(S_m, Q_m)))$.

Likewise, Class II may be illustrated by a finite-state action program which approximates a continuous system with sufficient precision.

Very Small Target Sets. Let ϵ be a very small and strictly positive real number. If $Q(x)$ becomes $8 \leq x \leq 8{+}2{\times}\epsilon$ and the operation $x{+}1$ in A_1 becomes $x{+}\epsilon$ then the optimality radius ρ is a finite but large integer, there are very many guard strata, and the resulting optimal policy is far more involved. In such a case, the stratification of guards is quite refined and any faithful finite approximation of the state space $\mathbb{R}$ is extremely precise; compare with the illustration of Case II above. The reachability precondition *pre.S.Q*, on the other hand, remains $0 \leq x \leq 10$; it characterizes the interval $\mathbb{R}_{[0,10]}$.

If $Q(x) \equiv (x = 8)$, viz. ϵ is replaced by 0, then ρ is infinite, *OptimPol* is inapplicable, and *pre.S.Q* characterizes a denumerable infinite subset of $\mathbb{R}_{[0,10]}$.

Nonlinear Action-Maps. Action maps may be nonlinear [25]. If the operation $4x$ in A_0 becomes x^2, for instance, then *OptimPol* remains applicable and the reachability precondition does not change. Clearly, the evaluation of satisfiability expressions usually becomes harder as the action maps become more intricate.

5.4 Additional Derivations

This brief, subsidiary Section is included for completeness.

Symbolic Expression of the Value Function. Let x be a state satisfying *pre.S.Q*. Its value $V(x)$ equals $V_L(n)$ for the level n such that x satisfies $H^{(n)}$. Thus, whenever the precondition of *OptimPol* holds (Sect. 5.1), the value function V can be computed using the finite set $V_\sigma = \bigcup_{n \in X_L} \{(V_L(n), H^{(n)})\}$:

$$V(x) = \min_{(v,P) \in V_\sigma} \{v \mid P(x)\} \ .$$

The auxiliary set V_σ can be constructed using a simple variant of *OptimPol* : the steps producing $\boldsymbol{C}^{(0)}$, $\boldsymbol{C}^{(n)}$, and $\boldsymbol{C}$ are respectively replaced by

$$V_\sigma^{(0)} := \{(0, H^{(0)})\}, \quad V_\sigma^{(n)} := V_\sigma^{(n-1)} \cup \{(V_L(n), H^{(n)})\}, \quad V_\sigma := V_\sigma^{(\rho)} \ .$$

Comments. The value $V(x)$, if any, is the length of the shortest (or the cost of the cheapest) S-path from x which reaches Q. Thanks to the above symbolic form, it is possible to compute shortest-paths lengths in the case of infinite graphs and target sets which are defined by action programs and target predicates satisfying the precondition of *OptimPol*; the latter is compared in Sect. 6.1 with the precondition of a related symbolic algorithm.

The auxiliary set V_σ could be generated first, and the optimal policy could then be derived using (10). Here, on the contrary, control policies are synthesized in one go because they are treated as first-class citizens.

Iterative Deduction of the Optimality Precondition. The optimality precondition *optim.S.Q* can be deduced iteratively, as it is the case for the reachability precondition (Sect. 2.3) and the termination one (Sect. 6.2) [13].

The following iteration yields $optim.S.Q \equiv W^{(\rho)}$; the predicates $K^{(m)}$ serve to stratify $W^{(\rho)}$: for $n \in [1, \rho]$,

$$K^{(0)} \equiv Q, \;\; K^{(n)} \equiv \neg W^{(n-1)} \wedge (\bigvee \boldsymbol{B}) \wedge \bigwedge_{i \in I} (B_i \Rightarrow best_i(n) \wedge pre.A_i.K^{(g_i(n))})$$

$$W^{(0)} \equiv Q, \;\; W^{(n)} \equiv W^{(n-1)} \vee K^{(n)} \;\; .$$

The cardinal ρ and the values $best_i(n)$ and $g_i(n)$ are obtained as in *OptimPol*.

6 Related Work

The symbolic algorithm *OptimPol* is compared with related ones which generate the following results: reachability preconditions for discrete-time systems (Sect. 6.1); restricted termination policies for discrete-time systems (Sect. 6.2); optimal policies for Markov decision processes and for hybrid systems (Sect. 6.3).

Let S be an action program, let $\boldsymbol{B}$ be its inherent policy, and let Q be a target predicate.

6.1 Symbolic Synthesis of Reachability Preconditions

Symbolic Algorithm. A classical algorithm constructs the reachability precondition *pre.S.Q* (Sect. 2.3):

```
Algorithm $ReachPre \doteq$
  begin $n := 0;\ D^{(0)} := Q;$
   while $\neg D^{(n)} \wedge pre.A.D^{(n)} \not\equiv$ false do $n := n + 1;$
          $D^{(n)} := D^{(n-1)} \vee \neg D^{(n-1)} \wedge pre.A.D^{(n-1)}$   od ;
   $\rho_R := n;\ D := D^{(\rho_R)}$                                   end .
```

So, $pre.S.Q \equiv D^{(\rho_R)}$. The *reachability radius* $\rho_R \in \mathbb{N}_\infty$ is the least cardinal such that $\forall x \in X : (pre.S.Q)(x) \Rightarrow (\exists p \in Paths.S.x.Q : nb_{edg}(p) \leq \rho_R)$.

For any state x satisfying *pre.S.Q*, its *reachability level* is given by

$$L_R(x) = \min\{nb_{edg}(p) \mid p \in Paths.S.x.Q\} \;\; . \tag{41}$$

Clearly, the reachability radius is the number of nonzero reachability levels:

$$\rho_R = \sup Rng(L_R) \;\; . \tag{42}$$

Reachability is bounded iff ρ_R is finite. Given (41) and (7), the map L_R is akin to V. Therefore, it satisfies (9) mutatis mutandis:

$$\forall j \in I, \forall x \in X : (pre.A_j.\breve{S}_Q)(x) \Rightarrow (L_R(x) \leq 1 + L_R(f_j(x))) \;\; . \tag{43}$$

Precondition of Algorithm ReachPre : Reachability is bounded and the satisfiability expression in *ReachPre* is decidable.

Symbolic Generator for Reachability. Given Sect. 2.3, Q is *reachable* by S (or S ensures reachability) iff $Q \vee \bigvee \boldsymbol{B} \equiv pre.S.Q$.

A *reachability policy* for S and Q is a refined policy $\boldsymbol{C}$ for S such that Q is reachable by $S{\downarrow}\boldsymbol{C}$. Once the reachability precondition has been generated, a reachability policy can be constructed readily. More precisely, a unique weakest reachability policy $\boldsymbol{R}$ exists and is given by the guards $R_i \equiv pre.A_i.\breve{S}_Q$. A symbolic generator *ReachPol* for reachability is thus easily obtained from *ReachDom*.

For the running example (2), $\boldsymbol{R} = (0 < x \leq 2.5, 0 \leq x \leq 9)$.

Unbounded Reachability. Reachability is not bounded only if the state space is infinite. For a significant family of problems over infinite state-spaces, structural properties ensure bounded reachability [16]. Still, bounded reachability is a severe restriction. Assume for instance that $x \leq 40$ replaces the guard $x \geq 0$ of action A_1 in (2); the target predicate $8 \leq x \leq 10$ is then reachable from any negative number and reachability is not bounded.

In case of unbounded reachability, the action program S can be approximated by S_K such that $\forall x \in X : (\forall p \in Paths.S_K.x.Q : nb_{edg}(p) \leq K)$ for some $K \in \mathbb{N}$: it suffices to add a counter $y \in [0, K]$, incremented in each action.

If reachability is unbounded and S has a sufficiently simple structure, the reachability precondition may be obtained by specific decision procedures [17].

Reachability vs. Optimality. The algorithm *ReachPre* is a symbolic form of the state-based algorithm which generates paths having a minimum number of edges [24]. It constructs a precondition determining a qualitative property, namely reachability. The algorithm *OptimPol* is a symbolic form of the greedy state-based algorithm which generates paths with shortest lengths (Sect. 4) [12]. It constructs a policy ensuring a quantitative property, namely optimality.

So, the computation of iterates in *ReachPre* ignores costs and is synchronous, whereas the computation of guard strata in *OptimPol* depends on costs and is asynchronous. As a consequence, each sub-domain $D^{(n)}$ in *ReachPre* is in general larger than the sub-domain $D^{(n)}$ in *OptimPol*. Yet, the respective limits $D^{(\rho_R)}$ and $D^{(\rho)}$ both are equivalent to the reachability precondition $pre.S.Q$. In short, the sub-domains grow faster in *ReachPre* than in *OptimPol*.

What is more, the radiuses ρ_R and ρ are related as follows (Appendix A):

$$\rho_R \leq \rho \leq \rho_R \times M_w \ . \tag{44}$$

Hence, bounded optimality is equivalent to bounded reachability.

Respective Preconditions. Given (44), the precondition of *ReachPre* (above) and that of *OptimPol* (Sect. 5.1) both require bounded reachability and decidable satisfiability. Actually, the precondition of *OptimPol* holds if (i) that of *ReachPre* holds and (ii) the decidability of the satisfiability expression in *ReachPre* implies the decidability of the one in (28).

Condition (ii) is often verified. Nevertheless, the sub-domains $D^{(n)}$ in the algorithm *ReachPre* usually have a simpler structure than those in *OptimPol*.

Respective Complexities. Let us eliminate ρ from (40) using (44). This yields $T(\mathit{OptimPol}) \in \mathit{Poly}(\rho_R + M_w + N + T_{\text{SAT}}(\mathit{OptimPol}))$. As to *ReachPre*, note that $N \in \mathit{Poly}(T_{\text{SAT}}(\mathit{ReachPre}))$ since N is the number of actions in A (5). So, $T(\mathit{ReachPre}) \in \mathit{Poly}(\rho_R + T_{\text{SAT}}(\mathit{ReachPre}))$. Hence,

$$\begin{aligned} & M_w + T_{\text{SAT}}(\mathit{OptimPol}) \in \mathit{Poly}(\rho_R + T_{\text{SAT}}(\mathit{ReachPre})) \\ \Rightarrow\ & T(\mathit{OptimPol}) \in \mathit{Poly}(T(\mathit{ReachPre})) \ . \end{aligned}$$

The premiss may of course be falsified, e.g. by $w_0 = 2^{\rho_R}$ and $w_1 = 1$.

Comments. Given the close relationships above, efficient implementation techniques used for *ReachPre* [10] should be reused for *OptimPol*.

The applicability domain of *OptimPol* appears to be a relatively large part of that of *ReachPre*. The present work does not aim at enlarging the applicability domain of *ReachPre*, which is admittedly limited.

6.2 Symbolic Synthesis of Fast-Termination Policies

Termination Precondition. The *(good-)termination precondition* $wp.S.Q$, a.k.a. weakest precondition, can be defined by $(wp.S.Q)(x) \equiv (pre.S.Q)(x) \wedge (\mathit{TotalPaths}.S.x = \mathit{TotalPaths}.S.x.Q)$. So, a state x satisfies $wp.S.Q$ iff some S-path from x reaches Q and each total S-path from x reaches Q. The termination precondition can be derived iteratively [13].

The preconditions $pre.S.Q, wp.S.Q$, and $optim.S.Q$ characterize three properties of increasing strength: reachability, i.e. the possibility of reaching the target set; good termination, i.e. the certainty of reaching the target set; optimality, i.e. the certainty of reaching the target set at a minimum cost. Firstly, it is clear that $pre.S.Q \Leftarrow wp.S.Q$. Secondly, $wp.S.Q \Leftarrow optim.S.Q$ is proved as follows: for any state x, $(\mathit{TotalPaths}.S.x = \mathit{OptimPaths}.S.x.Q) \Leftarrow (optim.S.Q)(x)$ and $\mathit{OptimPaths}.S.x.Q \subseteq \mathit{TotalPaths}.S.x.Q \subseteq \mathit{TotalPaths}.S.x$ (Sect. 2.3, Sect. 3); hence, $(\mathit{TotalPaths}.S.x = \mathit{TotalPaths}.S.x.Q) \Leftarrow (optim.S.Q)(x)$.

The action program S *terminates in* Q iff $Q \vee \bigvee \boldsymbol{B} \equiv wp.S.Q$.

Termination Control. A *(good-)termination policy* for S and Q is a refined policy $\boldsymbol{C}$ for S such that $S{\downarrow}\boldsymbol{C}$ terminates in Q.

A *fast-termination policy* for (S, Q) is a termination policy $\boldsymbol{C}$ for (S, Q) such that $\forall x \in X : (\forall p \in \mathit{Paths}.(S{\downarrow}\boldsymbol{C}).x.Q : nb_{edg}(p) = L_R(x))$; cf. (41). A unique weakest fast-termination policy exists (see below). It is not always a weakest termination policy because it minimizes the number of edges in paths whereas termination requires no minimization, by definition.

A termination policy can be derived from a termination function [1][3][13][15]. A *termination function* for S and Q is a map $t \in X \hookrightarrow \mathbb{N}$ such that, for any state x, $(x \in \mathit{Dom}(t)) \equiv (pre.S.Q)(x)$ and $(t(x) = 0) \equiv (Q \wedge \neg \bigvee \boldsymbol{B})(x)$; the state space X may be infinite. The t-based termination policy $\boldsymbol{T}^{[t]}$ is the tuple of guards $T_i^{[t]}(x) \equiv (pre.A_i.\breve{S}_Q)(x) \wedge (t(x) - t(f_i(x)) \geq 1)$. If it is equal to the inherent policy of S then S terminates on Q.

Let $\boldsymbol{C}$ be a termination policy. The graph $G_{S\downarrow\boldsymbol{C}}$ is an acyclic subgraph of the graph G_S. Since this acyclic subgraph is not necessarily maximal, $\boldsymbol{C}$ is not always a weakest termination policy. For a finite state space, the construction of a weakest termination policy boils down to that of a maximal acyclic subgraph. As matters stand, no general algorithm *TerminPol* is known for constructing weakest termination policies when state spaces are infinite.

More often than not, the graph G_S contains different maximal acyclic subgraphs, in which case no unique weakest termination policy exists.

Symbolic Generator for Fast Termination. A symbolic generator constructs fast-termination policies $\boldsymbol{T}$ [22], used to prevent starvation in concurrent programs. It can be obtained by dissecting the algorithm *ReachPre* as follows.

Firstly, the sub-domains $D^{(n)}$ are decomposed into domain strata $H^{(m)}$. So, for $n \in [1, \rho]$, $D^{(n)} \equiv D^{(n-1)} \vee H^{(n)}$ and $H^{(n)} \equiv \neg D^{(n-1)} \wedge \mathit{pre}.A.H^{(n-1)}$. Secondly, the domain strata $H^{(n)}$ themselves are decomposed into guard strata $F_i^{(n)}$, for $i \in I$. So, $H^{(n)} \equiv \bigvee_{i \in I} F_i^{(n)}$ and $F_i^{(n)} \equiv \neg D^{(n-1)} \wedge \mathit{pre}.A_i.H^{(n-1)}$. Thirdly, the fast-termination guards T_i are decomposed into guard strata $F_i^{(m)}$. So, $T_i^{(n)} \equiv T_i^{(n-1)} \vee F_i^{(n)}$. The resulting generator is equivalent to that in [22]:

```
Algorithm $\mathit{FastTerminPol} \doteq$
   begin $n := 0$;
    for $i \in I : (F_i^{(0)} := \mathbf{false}; T_i^{(0)} := F_i^{(0)}); H^{(0)} := Q; D^{(0)} := H^{(0)}$;
    while $\neg D^{(n)} \wedge \mathit{pre}.A.H^{(n)} \not\equiv \mathbf{false}$ do $n := n + 1$;
         for $i \in I$ : ( $F_i^{(n)} := \neg D^{(n-1)} \wedge \mathit{pre}.A_i.H^{(n-1)}$;
                    $T_i^{(n)} := T_i^{(n-1)} \vee F_i^{(n)}$ ) ;
         $H^{(n)} := \bigvee_{i \in I} F_i^{(n)}$;  $D^{(n)} := D^{(n-1)} \vee H^{(n)}$        od ;
    $\rho_R := n$; $\boldsymbol{T} := \boldsymbol{T}^{(\rho_R)}$                                  end .
```

The algorithm *FastTerminPol* is structurally similar to *ReachPre*. So, it has the same precondition, does not depend on costs, and uses synchronous steps only. Moreover, the fast-termination policy $\boldsymbol{T}$ is the unique weakest one.

Termination vs. Optimality. Optimality has already been compared with reachability (Sect. 6.1). It is now compared with good termination.

Termination Control vs. Optimal Control. It seems that termination control is more tractable than optimal control because good termination is implied by optimality: it often appears easier to invent an adequate termination function than to invent the uniquely defined value-function. The weakest optimal policy itself is a termination policy, although not a weakest one in general.

Moreover, the value function V per se may serve as a termination function. The resulting V-based termination policy $\boldsymbol{T}^{[V]}$ is weaker than the V-based weakest optimal policy since $(V(x) - V(f_i(x)) \geq 1) \Leftarrow (V(x) - V(f_i(x)) = w_i)$; see (10). Still, $\boldsymbol{T}^{[V]}$ in general is not a weakest termination policy because the value function V may determine a partial order in X which is too restrictive.

FastTerminPol vs. OptimPol. The generator *FastTerminPol* is equivalent to the generator *OptimPol* if all actions have the same cost, viz. the same priority. The asynchronous term $H^{(g_i(n))}$ in (36) then becomes the synchronous term $H^{(n-1)}$ in *FastTerminPol*. Actually, $L = L_R$ and $\rho = \rho_R$ if the action costs are equal. Therefore, the bijection V_L is not needed in the generator *FastTerminPol* and the equal action costs could all be 1.

Running Example (Continued). Let us replace $w_1 = 26$ by $w_1 = 13 = w_0$ in (2). Then the optimal policy generated by *OptimPol* is equal to the fast-termination policy $\boldsymbol{T}$ generated by *FastTerminPol*. Namely,

$$\begin{aligned} T_0(x) &\equiv (0.125 \leq x \leq 0.15625) \vee (0.25 \leq x \leq 0.375) \\ &\quad \vee (0.4375 \leq x \leq 0.625) \vee (1.5 < x \leq 2.5) \\ T_1(x) &\equiv (0 \leq x < 0.5) \vee (0.625 < x \leq 1.5) \vee (2.5 < x < 8) \ . \end{aligned}$$

The optimal policy $\boldsymbol{C}$ in (31) differs from $\boldsymbol{T}$, and is even simpler. For our little running example, a unique weakest termination policy exists and is equal to the weakest reachability policy $(0 < x \leq 2.5,\ 0 \leq x \leq 9)$ (Sect. 6.1).

Path- or State-Completeness of Control Policies. Let ψ stand for any of the following properties: reachability, termination, or optimality. A policy $\boldsymbol{C}$ for S is *path-complete* with respect to ψ iff $Paths.(S{\downarrow}\boldsymbol{C})$ contains each S-path guaranteeing ψ. The weakest reachability policy is path-complete since it allows all paths which reach the target predicate. The weakest optimal policy also is path-complete: it permits all optimal paths. As observed above, a weakest termination policy in general is not path-complete.

A policy $\boldsymbol{C}$ is *state-complete* iff $Q \vee \boldsymbol{C} \equiv pre.S.Q$; see (11). A state-complete termination policy is not always a weakest one; take, as an example, the above policy $\boldsymbol{T}$ qua termination policy.

6.3 Symbolic Generators for Optimality in Related Systems

Markov Decision Processes. A symbolic dynamic-programming generator yields policies which maximize expected total discounted rewards in MDPs [7]. It generates value functions symbolically, using the situation calculus, and then extracts optimal policies; a similar approach is outlined in Sect. 5.4. As in that paper, our aim is to tackle huge state-spaces and our approach is based on value-dependent equivalence classes (18). Here, however, a basic framework is chosen on purpose: transitions are not stochastic, no discount factor is used, and the greedy principle is applied instead of dynamic programming.

The design space of symbolic generators is discussed in Sect. 7.

Hybrid Systems. In the case of continuous or hybrid systems, which are not considered here, state spaces are rarely finite and thus generators must be symbolic. A symbolic algorithm [8] generates optimal strategies for timed game-automata, in a subset of linear hybrid systems, as follows: first, a cost-independent iteration yields candidate strategies; second, cost-dependent polyhedra are constructed; third, a selective search among these polyhedra yields optimal, deterministic strategies. A thorough implementation is provided. Moreover, optimal policies can be generated for a rich class of nonlinear differential equations, provided every discrete transition resets each state to a constant [9].

Further comments are found in Sect. 7.

7 Concluding Remarks

Admissible Problems. The family of optimality problems accepted by the proposed technique is characterized by bounded reachability and decidable satisfiability (Sect. 6.1), and also by the following restrictions (Sect. 2).

Each label identifies a unique action and each assignment is deterministic. Hence, nondeterminism is bounded by the number of actions. Action costs are nonzero natural numbers. Rational costs may be mapped to naturals by a change of scale. Strictly positive real numbers may be used, but then (44) may be falsified; they also may be approximated by rationals. Path costs are defined by a linear function (Sect. 2.2), unlike costs in quadratic-programming control.

Interactive Dynamics. The interaction between controllable and uncontrollable dynamics may be seen as an interplay between the reachability precondition and the termination one [3]. In fact, the reachability precondition presupposes that nondeterminism is controllable, viz. optimistic or angelic, whereas the termination precondition presupposes that nondeterminism is uncontrollable, viz. pessimistic or demonic. Thanks to this logical duality, discrete-time games with two players are readily formalized using interaction programs [3][20]. An interaction program expresses the iteration of the sequential composition of two action sets A and A', which respectively correspond to the controllable proponent and the uncontrollable opponent. So, the action set A is replaced in S by $(A; A')$, the predicate $pre.A.P$ is replaced in (5) by $pre.A.(wp.A'.P)$, the guards in A' may not be refined, and the value function is defined using the min-max pattern [4]. The present work could thus be adapted to the symbolic generation of optimal strategies in certain games.

Methodical Design of Symbolic Generators. Symbolic generators for optimality, different from *OptimPol*, might be inspired by related ones for optimality ([7][8][9]). This specific question needs more study. Furthermore, a systematization of known generators for optimality ([7][8][9] or here) may help in the generic design of symbolic generators for multiple properties, such as termination (Sect. 6.2) and minimization of quadratic or expected total costs [6], taken in the context of discrete-time systems. We discuss this topic in some detail.

The stages in Sect. 4 are the choice of a state-based generator for optimality, the definition of an adequate abstract structure given this generator, and the design of a symbolic generator given the state-based one and the abstract structure; likewise in [7]. This development is easily generalized on paper: the property of optimality and the greedy state-based generator are replaced by a property ψ and a state-based generator ψ_Pol for ψ. It is less easy to invent an abstract structure and to construct a symbolic generator in a particular instance. To carry out this task for multiple properties raises even more difficulties.

An alternative endeavour may be imagined to tackle this issue. First, a typical class of admissible control problems should be identified. One should then design an algorithm A^C transforming concrete transition systems into abstract finite-state "symbolic systems"; see below. Let ψ_Pol be a state-based generator for a property ψ. It does generate an abstract policy from the symbolic system obtained by applying A^C to a given transition system. It remains thus to design a complementary algorithm C^A transforming this abstract policy into a concrete one which ensures ψ in the given transition system. If this endeavour succeeds, the sequential algorithm (A^C; ψ_Pol; C^A) would be a symbolic generator for ψ. So, state-based generators –lock, stock and barrel– would be reused symbolically without more ado, and the present paper would be made redundant.

Symbolic Abstraction. The ideas of abstraction and simulation have been widely used to shed light on the structure of dynamics [2][3][9][10][18][19][25]. In our view, but we may be mistaken, they could also provide an illuminating guide towards a unified approach to the design of symbolic generators. The symbolic dynamical systems above, for instance, could result from a partitioning of concrete state-spaces into equivalence classes determined by formal languages on the alphabet of labels: namely, each concrete state belongs to the equivalence class determined by the set of words abstracting all the simple paths which begin with that state. So, as in future covers [18], states and transitions in symbolic systems would correspond to formal languages and shift maps. A symbolic system would have a finite state-space if it abstracts a transition system where the lengths of simple paths are bounded. Symbolic systems might also abstract continuous systems the dynamics of which can be abstracted symbolically.

Conclusion. A symbolic algorithm constructing optimal control policies for a family of transition systems is presented. Its overall structure is akin to that of the classical symbolic algorithm generating reachability preconditions: simply, synchronous iteration steps used in the latter algorithm are changed into asynchronous, greedy ones ensuring optimality. So, the applicability domain of the proposed symbolic generator for optimality is not much smaller than that of the symbolic method which establishes reachability.

Acknowledgments. We gratefully acknowledge helpful suggestions by J.-F. Raskin, F. Cassez, referees, and participants to meetings of IFIP WG 2.1 (Algorithmic Languages and Calculi) and WG 2.3 (Programming Methodology).

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge Univ. Press, Cambridge (to be published, 2008)
2. Akin, E.: The General Topology of Dynamical Systems. Amer. Math. Soc., Providence (1998)
3. Back, R.-J., von Wright, J.: Refinement Calculus. Springer, New York (1998)
4. Başa, T., Olsder, G.J.: Dynamic Noncooperative Game Theory, 2nd edn. SIAM, Philadelphia (1999)
5. Bellman, R.: Dynamic Programming. Princeton Univ. Press, Princeton (1957)
6. Bertsekas, D.: Dynamic Programming and Optimal Control. Athena Scientific, Belmont (2000)
7. Boutilier, C., Reiter, R., Price, B.: Symbolic dynamic programming for first-order MDPs. In: Proc. 7th Int. Joint Conf. Artificial Intelligence, pp. 690–697. M. Kaufmann, San Francisco (2001)
8. Bouyer, P., Cassez, F., Fleury, E., Larsen, K.: Synthesis of optimal strategies using HyTech. Electronic Notes Theor. Computer Sci. 119, 11–31 (2005)
9. Bouyer, P., Brihaye, T., Chevalier, M.: Weighted O-Minimal Hybrid Systems Are More Decidable Than Weighted Timed Automata! In: Artemov, S.N., Nerode, A. (eds.) LFCS 2007. LNCS, vol. 4514, pp. 69–83. Springer, Heidelberg (2007)
10. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
11. de Bakker, J.W., de Roever, W.P.: A calculus for recursive program schemes. In: Nivat, M. (ed.) Proc. 1st Int. Conf. Automata, Languages and Programming, pp. 167–196. North-Holland, Amsterdam (1973)
12. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
13. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
14. Floyd, R.: Algorithm 97 (Shortest path). Commun. ACM 5, 345 (1962)
15. Floyd, R.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Proc. Symp. Appl. Mathematics, vol. 19, pp. 19–31. Amer. Math. Soc., Providence (1967)
16. Henzinger, T.A., Majumdar, R., Raskin, J.-F.: A classification of symbolic transition systems. ACM Trans. Computational Logic 6, 1–32 (2005)
17. Kupferman, O., Vardi, M.Y.: An Automata-theoretic Approach to Reasoning about Infinite-state Systems. In: Proc. 12th Int. Conf. Computer Aided Verification, LNCS, vol. 1855, pp. 36–52. Springer, Berlin (2006)
18. Lind, D., Marcus, B.: An Introduction to Symbolic Dynamics and Coding. Cambridge Univ. Press, Cambridge (1995)
19. Schmidt, D.A.: Structure-Preserving Binary Relations for Program Abstraction. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 245–265. Springer, Heidelberg (2002)
20. Sintzoff, M.: Iterative Synthesis of Control Guards Ensuring Invariance and Inevitability in Discrete-decision Games. In: Owe, O., Krogdahl, S., Lyche, T. (eds.) From Object-Orientation to Formal Methods. LNCS, vol. 2635, pp. 272–301. Springer, Heidelberg (2004)
21. Sontag, E.D.: Mathematical Control Theory. Springer, New-York (1990)
22. van Lamsweerde, A., Sintzoff, M.: Formal derivation of strongly concurrent programs. Acta Informatica 12, 1–31 (1979)
23. Vinter, R.: Optimal Control. Birkhäuser, Boston (2000)
24. Warshall, S.: A theorem on boolean matrices. J. ACM 9, 11–12 (1962)

25. Wiggins, S.: Introduction to Applied Nonlinear Dynamical Systems and Chaos. Springer, New York (1990)

Appendix A : Additional Proofs

Everywhere, $x \in X, i \in I, m \in \mathbb{N}$, and $n \in [1, \rho]$. Recall that $\breve{S}_Q \equiv pre.S.Q$.

Proof of (25)

$$\begin{aligned}
& \neg D^{(n-1)}(x) \wedge \mathit{best}_i(n) \wedge \mathit{pre}.A_i.(H^{(g_i(n))}(x)) \\
\equiv\ & \neg D^{(n-1)}(x) \wedge \mathit{best}_i(n) \wedge B_i(x) \\
& \qquad\qquad \wedge \breve{S}_Q(f_i(x)) \wedge\ L(f_i(x)) = g_i(n) && [(5), (18)] \\
\Rightarrow\ & \neg D^{(n-1)}(x) \wedge \breve{S}_Q(x) \wedge V(x) \leq w_i + V(f_i(x)) \\
& \quad \wedge V(f_i(x)) = V_L(g_i(n)) \wedge\ w_i + V_L(g_i(n)) = V_L(n) && [(9), (13), (24)] \\
\Rightarrow\ & \neg D^{(n-1)}(x) \wedge \breve{S}_Q(x) \wedge V(x) \leq V_L(n) && [\text{Elim.}\, w_i + V(f_i(x))] \\
\Rightarrow\ & \neg D^{(n-1)}(x) \wedge D^{(n)}(x) && [(13),\ (17)] \\
\Rightarrow\ & H^{(n)}(x) && [(17),\ (18)]
\end{aligned}$$

□

Proof of (26)

$$\begin{aligned}
\text{(26.a)}\quad C_i(x) & \Rightarrow V_L(L(x)) - w_i = V_L(L(f_i(x))) && [(10), (13)] \\
& \Rightarrow\ \mathit{best}_i(L(x)) && [(23)]
\end{aligned}$$

$$\begin{aligned}
\text{(26.b)}\quad C_i(x) & \Rightarrow (V_L(L(f_i(x))) = V_L(L(x)) - w_i) \wedge \mathit{best}_i(L(x)) && [(10), (26.a)] \\
& \Rightarrow\ V_L(L(f_i(x))) = V_L(L(x)) - w_i = V_L(g_i(L(x))) && [(24)] \\
& \Rightarrow V_L(L(f_i(x))) = V_L(g_i(L(x))) && [\text{transitive} =] \\
& \Rightarrow \quad L(f_i(x)) = \quad g_i(L(x)) && [\text{bijective } V_L]
\end{aligned}$$

□

Proof of (28)

$$\begin{aligned}
& V_L(n) \\
=\ & \min_x \{V(x) \mid \breve{S}_Q(x) \wedge V(x) > V_L(n-1)\} && [(12)] \\
=\ & \min_{x,i} \{V(x) \mid \breve{S}_Q(x) \wedge C_i(x) \wedge V(x) > V_L(n-1)\} && [(11), \neg Q(x)] \\
=\ & \min_{x,i} \{V(x) \mid \breve{S}_Q(x) \wedge B_i(x) \wedge \breve{S}_Q(f_i(x)) \wedge (V(x) = w_i + V_L(L(f_i(x)))) \\
& \qquad\qquad \wedge V(x) > V_L(n-1) \wedge L(f_i(x)) < n\} && [(10);\ \text{Prop. min}] \\
=\ & \min_{x,i,m} \{V(x) \mid \breve{S}_Q(x) \wedge B_i(x) \wedge \breve{S}_Q(f_i(x)) \wedge (V(x) = w_i + V_L(m)) \\
& \qquad\qquad \wedge \neg D^{(n-1)}(x) \wedge (m = L(f_i(x))) \wedge m < n\} && [\text{Intro.}\, m; (17)] \\
=\ & \min_{i,m:m<n} \{w_i + V_L(m) \mid \exists x : \neg D^{(n-1)}(x) \wedge B_i(x) \wedge H^{(m)}(f_i(x))\} \\
& && [\text{Elim. } V(x), \text{ Elim. } \breve{S}_Q(x) \text{ by } (6);\ (18)] \\
=\ & \min_{i,m:m<n} \{w_i + V_L(m) \mid \exists x : \neg D^{(n-1)}(x) \wedge (\mathit{pre}.A_i.H^{(m)})(x)\} && [(5)] \\
=\ & \min_{i,m:m<n} \{w_i + V_L(m) \mid \neg D^{(n-1)} \wedge \mathit{pre}.A_i.H^{(m)} \not\equiv \mathbf{false}\} && [\text{Satisfiability}]
\end{aligned}$$

□

Proof of (44)

Let $\sup_x$ stand for $\sup_{x:\breve{S}_Q(x)}$.

$$\begin{array}{llclclr}
\forall x : \breve{S}_Q(x) \Rightarrow & L_R(x) & \leq & L(x) & \leq & L_R(x) \times M_w & [\text{(a),(b) below}] \\
\Rightarrow & \sup_x\{L_R(x)\} & \leq & \sup_x\{L(x)\} & \leq & \sup_x\{L_R(x)\} \times M_w & [\text{Props. sup}] \\
\Rightarrow & \rho_R & \leq & \rho & \leq & \rho_R \times M_w \ . & [(14), (42)]
\end{array}$$

(a) Proof of $\forall x : \breve{S}_Q(x) \Rightarrow L(x) \geq L_R(x)$. Consider any x satisfying $\breve{S}_Q$ and some $p = (x_n, i_{n-1}, x_{n-1}, \cdots, x_1, i_0, x_0) \in \mathit{OptimPaths.S.x.Q}$. So, $x_n = x$ and $Q(x_0)$. Let us prove $\forall k \in [0, n] : L(x_k) \geq L_R(x_k)$ by induction on k :
– $(k = 0)$ $L(x_0) = L_R(x_0) = 0$ $[Q(x_0)$, (13), (41)].
– $(k > 0)$ The hypothesis is $L(x_{k-1}) \geq L_R(x_{k-1})$, for any $k \in [1, n]$. The thesis $L(x_k) \geq L_R(x_k)$ is proved by modus ponens: we have $V(x_k) = w_{k-1} + V(x_{k-1}) \geq 1 + V(x_{k-1})$ because p is optimal and $w_{k-1} \geq 1$, and we prove the following implication:

$$\begin{array}{rlr}
V(x_k) \geq 1 + V(x_{k-1}) & \equiv L(x_k) \geq 1 + L(x_{k-1}) & [(13)] \\
& \Rightarrow L(x_k) \geq 1 + L_R(x_{k-1}) & [\text{Induction hyp.}] \\
& \Rightarrow L(x_k) \geq L_R(x_k) \ . & [(43)]
\end{array}$$

Hence $L(x_n) \geq L_R(x_n)$, for $k = n$. Hence $L(x) \geq L_R(x)$, since $x_n = x$.

(b) Proof of $\forall x : \breve{S}_Q(x) \Rightarrow L(x) \leq L_R(x) \times M_w$. Consider any x satisfying $\breve{S}_Q$. Let $\mathcal{P}_{x,Q}$ stand for $\mathit{Paths.S.x.Q}$.

$$\begin{array}{lcr}
 & \forall p \in \mathcal{P}_{x,Q} : \mathit{cost}(p) \leq \mathit{nb}_{\mathit{edg}}(p) \times M_w & [(3)] \\
\Rightarrow & \min_{p \in \mathcal{P}_{x,Q}}\{\mathit{cost}(p)\} \leq \min_{p \in \mathcal{P}_{x,Q}}\{\mathit{nb}_{\mathit{edg}}(p)\} \times M_w & [\text{Props. min}] \\
\Rightarrow & V(x) \leq L_R(x) \times M_w & [(7), (41)] \\
\Rightarrow & L(x) \leq L_R(x) \times M_w \ . & [(13)]
\end{array}$$

□

Appendix B : Indexes

Notations. The references to the end of Sect. 1 are understood:

$<\!w_i\!>$	action cost, Sect. 2.1	$Y \hookrightarrow Z$	partial functions
$B_i \xrightarrow{i} \ldots$	action labelling, Sect. 2.1	$\mathit{Rng}(f)$	range of a function
do...**od**	action program, Sect. 2.1	$P \not\equiv$ **false**	satisfiability expression
$\#Y$	cardinal of a set	$P^{(\sup M)}$	supremum of a chain
$A_i \,[\!]\, A_j$	choice of actions, Sect. 2.1	E^x_e	substitution
$S{\downarrow}\boldsymbol{C}$	controlled by, Sect. 2.1	$Y \to Z$	total functions
$\mathit{Dom}(f)$	definition domain	$\boldsymbol{B}$	tuple of predicates
$\mathbb{N}_\infty$	naturals and infinity	$\bigvee \boldsymbol{B}$	union of predicates
$[m, n]$	interval		

Selected Identifiers. We write "§" for "Section":

Modal Semirings Revisited

Jules Desharnais[1] and Georg Struth[2]

[1] Département d'informatique et de génie logiciel, Pavillon Adrien-Pouliot,
1065, avenue de la Médecine, Université Laval, Québec, QC, Canada G1V 0A6
Jules.Desharnais@ift.ulaval.ca
[2] Department of Computer Science
University of Sheffield, S1 4DP, United Kingdom
g.struth@dcs.shef.ac.uk

Abstract. A new axiomatisation for domain and codomain on semirings and Kleene algebras is proposed. It is simpler, more general and more flexible than a predecessor, and it is particularly suitable for program analysis and construction via automated deduction. Different algebras of domain elements for distributive lattices, (co-)Heyting algebras and Boolean algebras arise by adapting this axiomatisation. Modal operators over all these domain algebras can then easily be defined. The calculus of the previous axiomatisation arises as a special case. An application in terms of a fully automated proof of a modal correspondence result for Löb's formula is also presented.

1 Introduction

Kleene algebras are foundational structures in computing with applications ranging from program semantics, construction and refinement to rewriting and concurrency control. Most current variants are close relatives of Kozen's elegant axiomatisation [15,16], and share some important features. They focus on the essential operations for modelling programs and similar discrete systems. They support abstract and concise reasoning about such systems within first-order equational logic. They have rich model classes that include relations, languages, paths in graphs, program traces and predicate transformers. And they enable a new kind of automated program analysis and construction that is supported by automated theorem provers (ATP systems) [11,12,23].

To connect the algebraic approach with traditional logics of programs such as dynamic, temporal or Hoare logics, modal operators have been added to semirings and Kleene algebras by axiomatising a notion of domain [8]. In this approach—henceforth referred to as the *DMS approach*—the domain function d maps elements of an idempotent semiring S, which can be assumed to model the actions of some system, to elements of a certain Boolean subalgebra B of S, which models the state space of the system. This two-sorted approach seems very natural since domain elements $d(x)$ represent precisely those states in B at which the action $x \in S$ is enabled. The resulting *modal semirings* and *modal Kleene algebras* have widely been applied since [5,7,11,20,22]. But despite its evident merits, the DMS approach shows some deficiencies.

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 360–387, 2008.

First, the domain algebra B in the range of d cannot be freely chosen: the DMS axioms imply that B must be the *maximal* Boolean subalgebra embedded into S in a certain way [8]. So the axioms determine B rather inelegantly and indirectly. Second, experiments show that the performance of ATP systems can suffer from the intricacies of the two-sorted setting [11] and inhibit verification tasks. But are these deficiencies unavoidable?

This paper proposes a new one-sorted approach to domain semirings and Kleene algebras with domain that overcomes the difficulties mentioned.

- We provide a new one-sorted equational axiomatisation of domain for arbitrary semirings. It induces distributive lattices as domain algebras and supports the definition of modal operators on these lattices.
- We extend domain semirings by an antidomain function that induces a Boolean domain algebra. We then reduce this axiomatisation to three simple equations plus the semiring axioms. We also show that all DMS theorems can be recovered in this simpler setting.
- The flexibility of the approach is further demonstrated by extending domain semirings to Heyting and co-Heyting algebras with modal operators.
- Finally, the improved applicability of the approach follows from proof experiments that include an automated modal correspondence proof for Löb's formula, which has an immediate impact for automated termination analysis in a first-order setting.

By these results, the new approach is simpler, more general and more flexible that the DMS approach without sacrificing the underlying intuitions. Its most important benefit may, however, be its superior suitability for automated program analysis and program construction with ATP systems.

ATP systems were also crucial for the technical development, for analysing dependencies, redundancies and reducibilities of axiom systems and for developing the basic calculi. ATP technology allowed us to greatly accelerate the otherwise tedious and time-consuming search for proofs and counterexamples and to focus entirely on concepts. We used the following tools:

- Waldmeister [3], which is currently the most powerful system for unit equational logic and which outputs equational proofs;
- Prover9 [2], which is currently the most powerful ATP for full first-order reasoning with Kleene algebras (but does not produce readable proofs);
- Mace4 [2], which generates finite (counter)models from first-order axioms.

All calculational proofs and counterexample searches in this paper have been automated with these tools on a standard PC. The input templates in the appendices should allow the interested reader to reproduce them quickly and easily. Unfortunately, the granularity of Waldmeister proofs is too fine for publication. So we include comprehensive proofs of our main theorems in the paper.

The remainder of this text is organised as follows. Section 2 to Section 5 introduce domain semirings which induce distributive lattices as domain algebras and develop their basic calculus. Section 6 provides conditions that relate the domain algebras with Boolean algebras. Section 7 sets up the link between domain semirings, Kleene algebras with domain and distributive lattices with operators. Section 8 to Section 10 introduce antidomain operations that turn domain algebras into Boolean algebras. In Section 11, the domain algebras of domain semirings are extended to (co-)Heyting algebras. Section 12 presents an application of the new axiomatisation in automated termination analysis. Section 13 presents a conclusion. Additional material is collected in four appendices.

2 Domain Semirings

Semirings are essentially rings without subtraction. Formally, a *semiring* is a structure $(S, +, \cdot, 0, 1)$ such that $(S, +, 0)$ is a commutative monoid, $(S, \cdot, 1)$ is a monoid, multiplication distributes over addition from the left and right and 0 is a left and right zero of multiplication. As usual in algebra, variants without 0 and 1 can easily be defined, but they are less interesting for our purpose.

The explicit semiring axioms (as ATP input) can be found in the Appendices A and B. We stipulate that multiplication binds more strongly than addition and we omit the multiplication symbol.

A standard semiring duality is *opposition.* It is obtained by swapping the order of multiplication (or by reading expressions from right to left). The opposite S^o of a semiring S is again a semiring and $S^{oo} = S$.

A semiring S is *idempotent* if $1 + 1 = 1$ or, equivalently, if $x + x = x$ holds for all $x \in S$. For idempotent semirings, the relation $\leq$ defined, for all $x, y \in S$, by $x \leq y \Leftrightarrow x + y = x$ is a partial order; $(S, \leq)$ is a semilattice with addition corresponding to join and with least element 0. Addition and multiplication are isotone with respect to that order.

Idempotent semirings have many computationally meaningful models. We refer to two standard models to motivate our approach:

- *Relation semirings* are idempotent semirings formed by binary relations under union, relational product, the empty relation and the unit relation.
- *Trace semirings* are idempotent semirings formed by sets of traces of a program or transition system under union, complex products based on trace products, the empty set of traces and the set of states from which traces are built. As usual, traces are words over a state alphabet and an action alphabet in which the first and the last letter are state symbols and in which state and action symbols alternate. The trace product is a partial operation that "glues together" traces at their first and last states if these states are equal and that is undefined if they differ.

Language and path semirings arise as special cases of trace semirings.

We axiomatise a domain function on arbitrary semirings. It can be motivated in several ways. First, trace and relation semirings can be taken as starting points and some natural properties of domain can be selected. Alternatively, the DMS axioms, included at the end of this section, can be translated into the one-sorted setting. In addition, the notion axiomatised can be tested on the algebra of domain elements that is induced. We first present the axioms and then motivate them from all these points of view.

A *domain semiring* is a semiring S extended by the *domain operation* $d : S \to S$ which, for all $x, y \in S$, satisfies the following axioms:

$$x + d(x)x = d(x)x, \tag{D1}$$
$$d(xy) = d(xd(y)), \tag{D2}$$
$$d(x) + 1 = 1, \tag{D3}$$
$$d(0) = 0, \tag{D4}$$
$$d(x + y) = d(x) + d(y). \tag{D5}$$

The following fact can easily be shown by an ATP system, and we present the proof generated by Waldmeister as an example in Appendix C.

Proposition 2.1. *Domain semirings are idempotent.*

Hence every domain semiring can be ordered.

Let us discuss the particular choice of axioms. The axioms (D1) and (D2) correspond to the DMS axioms (6) and (8) below. The axioms (D3), (D4) and (D5) follow from the DMS axioms, but, as we will see below, not from (D1) and (D2). DMS axiom (7) is particular to the two-sorted setting and cannot directly be expressed here. This will further be discussed below.

By Proposition 2.1, axiom (D1) can be rewritten as $x \le d(x)x$. We say more generally that an element y of an idempotent semiring is a *left preserver* of an element x if $x \le yx$. So $d(x)$ is a left preserver of x and we call this axiom the *preservation* axiom. Intuitively, $d(x)$ should even be the least left preserver of x. Lemma 4.1(xi) below shows that this is indeed the case. Axiom (D2) says that only the domain of y contributes to the domain of xy; as previously we call it the *locality* axiom. Axiom (D3) can be rewritten as $d(x) \le 1$; it says that all domain elements are below 1 and we call it the *subidentity* axiom. By axiom (D4) and axiom (D5), the domain function is *strict* and *additive*, and we name these axioms accordingly.

The image of the domain operation d on a domain semiring S is denoted by $d(S)$ and elements of $d(S)$ are called a *domain elements* (with respect to S and d). We will consistently use the letters p, q, r to denote domain elements.

The following fixpoint lemma characterises domain elements within the language of domain semirings.

Proposition 2.2. *An element of a domain semiring is a domain element if and only if it is a fixpoint of the domain operation.*

Proof. Let S be a semiring with a mapping d that satisfies (D2). We show that $x \in d(S)$ if and only if $x = d(x)$. First, every $x \in d(S)$ is the image of some $y \in S$, that is, $x = d(y)$. Therefore, $d(x) = d(d(y)) = d(1d(y)) = d(1y) = d(y) = x$ by (D2). Second, $x = d(x)$ trivially implies that $x \in d(S)$. □

Trace and relation semirings with their standard domain operations are models of domain semirings, as can easily be shown. If x is a binary relation, that is, a set of pairs on some given set, then $d(x) = \{(a, a) : (a, b) \in x\}$; if x is a set of traces, then $d(x) = \{p : p = \mathsf{first}(\tau) \text{ and } \tau \in x\}$, as expected.

An even simpler exercise consists in axiomatising codomain semirings as the opposites of domain semirings: A *codomain semiring* is a semiring S extended by the *codomain operation* $d^o : S \to S$ that, for all $x, y \in S$, satisfies

$$x + xd^o(x) = xd^o(x), \tag{1}$$

$$d^o(xy) = d^o(d^o(x)y), \tag{2}$$

$$d^o(x) + 1 = 1, \tag{3}$$

$$d^o(0) = 0, \tag{4}$$

$$d^o(x + y) = d^o(x) + d^o(y). \tag{5}$$

By duality, the opposites of all statements about domain semirings hold in codomain semirings. Therefore only the interaction of domain and codomain deserves further investigation.

At the end of this section, we present the original DMS axioms. They are not needed to understand the further development of this paper, but they illustrate the gain in simplicity obtained.

Let S be an idempotent semiring and let B be the maximal Boolean algebra embedded in S such that 0 becomes the minimal element of B and 1 its maximal element. S is a *DMS domain semiring* if it can be extended by a *domain function* $d : S \to B$ that, for all $x, y \in S$ and $p \in B$, satisfies the axioms

$$x \leq d(x)x, \tag{6}$$

$$d(px) \leq p, \tag{7}$$

$$d(xd(y)) \leq d(xy). \tag{8}$$

It has been shown that (6) and (7) are equivalent to the least left preserver axiom

$$x \leq px \Leftrightarrow d(x) \leq p. \tag{9}$$

3 Irredundancy and Irreducibility of Domain Axioms

In this section we show that the domain axioms of domain semirings are irredundant, that is, no domain axiom is entailed by the semiring axioms and the remaining domain axioms. In fact we show that they are even irredundant with respect to the axioms of idempotent semirings. We also show irreducibility of the axioms in the sense that (D2) and (D5) cannot further be weakened to inequalities, that is, both inequalities that determine these axioms are irredundant.

Formally, a first-order formula ϕ is *irredundant* with respect to a set of first-order formulas Γ if some model of Γ is not a model of ϕ. We say that a set Γ is *irredundant* if each $\phi \in \Gamma$ is irredundant with respect to $\Gamma - \{\phi\}$. Obviously, discarding an irredundant formula from an axiom set changes the theory while discarding a redundant formula does not. In general, there is no guarantee that irredundancy of an axiom set can be established through (small) finite models alone, but, in practice, Mace4 proves very helpful.

Proposition 3.1. *The domain axioms of domain semirings are irredundant.*

Proof. We used Mace4 to find models that satisfy the semiring axioms plus all combinations of four domain axioms except the fifth one.

We first show irredundancy of (D1). Consider the *Boolean semiring* with elements 0 and 1 and with addition and multiplication defined by the tables below. Let also domain be defined as in the following table.

d	0	1
	0	0

$+$	0	1
0	0	1
1	1	1

$\cdot$	0	1
0	0	0
1	0	1

Then (D2)-(D5) hold, but not (D1), since $0 \cdot 1 = 0 \neq 1 = 1 + 0 \cdot 1$.

Irredundancy of (D2) follows by setting $x = y = 2$ in the model

d	0	1	2
	0	1	1

$+$	0	1	2
0	0	1	2
1	1	1	1
2	2	1	2

$\cdot$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	0

Irredundancy of (D3) follows by setting $x = 1$ in the model

d	0	1	2
	0	2	2

$+$	0	1	2
0	0	1	2
1	1	1	2
2	2	2	2

$\cdot$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	2

Irredundancy of (D4) can be established in the model.

d	0	1
	1	1

$+$	0	1
0	0	1
1	1	1

$\cdot$	0	1
0	0	0
1	0	1

Irredundancy of (D5) follows by setting $x = 1$ and $y = 2$ in the model

d	0	1	2	3
	0	1	3	3

$+$	0	1	2	3
0	0	1	2	3
1	1	1	2	1
2	2	2	2	2
3	3	1	2	3

$\cdot$	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	2	2
3	0	3	2	3

□

It usually takes far less time to generate such models with Mace4 than to understand them.

The next lemma shows that the additivity axiom is irreducible.

Lemma 3.2. *Some non-additive function d on an idempotent semiring satisfies (D1)-(D4) and either one of $d(x) + d(y) \leq d(x + y)$ or $d(x + y) \leq d(x) + d(y)$.*

Proof. Mace4 presents a 6-element model for the first claim and a 5-element model for the second claim. □

Lemma 3.2 implies that isotonicity of domain does not entail its additivity, whereas every additive function on a semilattice is isotone, whence domain has this property (cf. Lemma 4.1(vii)). Mace4 also shows that isotonicity of domain does not follow from the domain axioms (D1)-(D4) alone.

The last lemma of this section states that the locality axiom is irreducible.

Lemma 3.3. *Some non-local function d on an idempotent semiring satisfies (D1) and (D3)-(D4) and either one of* $d(xd(y)) \leq d(xy)$ *and* $d(xy) \leq d(xd(y))$.

Proof. Mace4 presents a 3-element model for the first and a 4-element model for the second claim. □

In the DMS axiomatisation, an additivity axiom is not needed and locality can be weakened to $d(xd(y)) \leq d(xy)$.

4 Basic Domain Calculus

We now revisit the basic domain calculus of the DMS axiomatisation [8] in our new setting. It turns out that most of the calculus is preserved. We will later enrich the new axiomatisation to reconstruct precisely the DMS calculus in a much simpler setting.

Lemma 4.1. *Let S be a domain semiring. Let $x, y \in S$ and let $p \in d(S)$. Then*

(i) $d(x)x = x$ *(domain is a left invariant),*
(ii) $d(p) = p$ *(domain is a projection),*
(iii) $d(xy) \leq d(x)$ *(domain increases for prefixes),*
(iv) $x \leq 1 \Rightarrow x \leq d(x)$ *(domain expands subidentities),*
(v) $d(x) = 0 \Leftrightarrow x = 0$ *(domain is very strict),*
(vi) $d(1) = 1$ *(domain is co-strict),*
(vii) $x \leq y \Rightarrow d(x) \leq d(y)$ *(domain is isotone),*
(viii) $d(px) = pd(x)$ *(domain elements can be exported),*
(ix) $d(x)d(x) = d(x)$ *(domain elements are multiplicatively idempotent),*
(x) $d(x)d(y) = d(y)d(x)$ *(domain elements commute),*
(xi) $x \leq px \Leftrightarrow d(x) \leq p$ *(domain elements are least left-preservers),*
(xii) $xy = 0 \Leftrightarrow xd(y) = 0$ *(domain is weakly local).*

All these properties have been proved by Waldmeister and Prover9. Variants hold in the DMS calculus and they all hold on trace and relation semirings, too.

In relation and trace semirings, all subidentities are domain elements. In other domain semirings, this need not be the case.

Lemma 4.2. *Subidentities of domain semirings need not be domain elements.*

Proof. Mace4 presents a 3-element model. □

Lemma 4.1(viii) displays a variant of DMS axiom (7). There, additivity of domain is redundant. Here, the situation is different.

Lemma 4.3. *Some non-additive function d on an idempotent semiring satisfies (D1)-(D4) and $d(d(x)y) = d(x)d(y)$.*

Proof. Mace4 presents a 4-element model. □

The least left preserver property in Lemma 4.1(xi) requires p to be a domain element. More generally, it can easily be shown that $d(x) \leq y$ implies $x \leq yx$ for arbitrary semiring elements. However, the converse implication does not hold.

Lemma 4.4. *On some domain semiring, $x \leq yx \Rightarrow d(x) \leq y$ does not hold.*

Proof. Mace4 presents a 3-element model in which $y \leq 1$. □

Finally, the weak locality property in Lemma 4.1(xii) is also very natural for trace or relation semirings, where it models the pointwise behaviour of relational products and trace products. In the DMS axiomatisation, weak locality is equivalent to locality and it is also equivalent to the fact that $xy = 0 \Leftrightarrow d^o(x)d(y) = 0$. For domain semirings, the situation is slightly weaker.

Lemma 4.5. *Some non-local function on an idempotent semiring satisfies (D1), (D3)-(D5) and weak locality.*

Proof. Mace4 presents a 4-element model. □

Section 10 shows that equivalence holds for Boolean domain semirings.

In sum, the basic domain calculus of the DMS axiomatisation can, up to equivalence of locality and weak locality, be reconstructed in the context of domain semirings.

5 Domain Algebras

This section studies the domain algebra, that is, the algebra of domain elements, which is induced by the domain axioms. By Proposition 2.2, domain elements are fixpoints of the domain operation. This makes their structure easy to analyse.

It is well known that the set of subidentities of an idempotent semiring under addition and multiplication forms an idempotent semiring. Using Proposition 2.2, we can sharpen this fact for domain semirings.

Lemma 5.1. *Let S be a domain semiring. Then $(d(S), +, \cdot, 0, 1)$ is an idempotent semiring.*

Proof. By Proposition 2.2, it suffices to show that 0, 1, $p+q$ and pq are fixpoints of d. The first and second property hold by axiom (D4) and Lemma 4.1(vi). The third property follows from axiom (D5) and Proposition 2.2. The fourth property follows from Lemma 4.1(viii) and again from Proposition 2.2. □

All these proof tasks have been automated with Waldmeister and Prover9.

Lemma 5.2 **([4]).** *A semiring is a distributive lattice if $x + 1 = 1$ and $xx = x$ hold for all elements x.*

Proposition 5.3. *Let S be a domain semiring. Then the tuple $(d(S), +, \cdot, 0, 1)$ is a bounded distributive lattice.*

Proof. Use Lemma 5.1, Lemma 5.2, Lemma 4.1(ix) and Axiom (D3). □

We henceforth call $d(S)$ the *domain algebra* of S. In contrast, the idempotent semiring of subidentities need not form a distributive lattice. Mace4 presents a 4-element domain semiring with subidentities x and y in which $xy = yx$ does not hold; the multiplication table below is not symmetric for $x = 2$ and $y = 3$.

d	0	1	2	3
	0	1	3	3

$+$	0	1	2	3
0	0	1	2	3
1	1	1	1	1
2	2	1	2	3
3	3	1	3	3

$\cdot$	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	0
3	0	3	2	3

In the DMS axiomatisation, the domain algebra is a Boolean subalgebra by definition. For domain semirings, this needs no longer be the case.

Lemma 5.4. *The domain lattices of domain semirings need not be Boolean.*

Proof. Mace4 shows that it is not the case that for all x there exists a y such that $d(x) + d(y) = 1$ and $d(x)d(y) = 0$. Consider $x = 2$ in the following domain semiring:

d	0	1	2
	0	1	2

$+$	0	1	2
0	0	1	2
1	1	1	1
2	2	1	2

$\cdot$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	2

□

Are all domain lattices (co-)Heyting algebras? This cannot not be refuted by a finite counterexample, since all finite distributive lattices are (co-)Heyting algebras [13]. We leave it as an open question.

6 Domain Algebras and Boolean Algebras

We now present a sufficient condition that characterises certain domain elements without even mentioning the domain operation. It links subidentities with domain elements and relates domain algebras with Boolean subalgebras of subidentities. Our criterion uses a weak form of Boolean complementation.

Proposition 6.1. *Let S be a domain semiring. Let $x \in S$ and let there exist some $y \in S$ such that*

$$x + y = 1 \quad \textit{and} \quad yx = 0.$$

Then x is a domain element.

Proof. We first show that $xd(x) = x$ and that $yd(x) = 0$. For the first identity, $xd(x) \leq x$ holds by (D3) and the converse inequality holds, since, by Lemma 4.1(i), $x = (x + y)x = xx + yx = xx = xd(x)x \leq xd(x)$. The second identity holds, since, by Lemma 4.1(i), (D1) and (D4),

$$yd(x) = d(yd(x))yd(x) = d(yx)yd(x) = 0yd(x) = 0.$$

Then $d(x) = (x + y)d(x) = xd(x) + yd(x) = x + 0 = x$. □

An automated proof with Prover9 took less than one second. Waldmeister is not appropriate since reasoning with hypotheses and mixed quantification is needed.

The condition $yx = 0$ is quite sensitive: it does not imply $xy = 0$, although y is a subidentity because of $x + y = 1$. Mace4 presents a 5-element counterexample.

Lemma 6.2. *Some domain semiring S with $x, y \in S$ satisfies $x + y = 1$ and $xy = 0$, but not $d(x) = x$.*

Proof. Mace4 presents the following counterexample, in which $x = 2$ and $y = 3$.

d	0	1	2	3	4
	0	1	1	3	3

$+$	0	1	2	3	4
0	0	1	2	3	4
1	1	1	1	1	1
2	2	1	2	1	2
3	3	1	1	3	3
4	4	1	2	3	4

$\cdot$	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	2	0	0
3	0	3	4	3	4
4	0	4	4	0	0

□

It is, however, straightforward to further constrain the assumptions in order to enforce that both x and y are domain elements.

Corollary 6.3. *Let S be a domain semiring. Let $x \in S$ and let there exist some $y \in S$ such that*

$$x + y = 1, \quad xy = 0 \quad \text{and} \quad yx = 0.$$

Then x and y are domain elements.

Proof. The proof is immediate from Proposition 6.1. □

In particular, we say that an element x of an idempotent semiring S is *complemented* if there exists some $y \in S$ such that $x + y = 1$, $xy = 0$ and $yx = 0$. We denote by B_S the set of all complemented elements in S.

Lemma 6.4. *Let S be an idempotent semiring. Then $(B_S, +, \cdot, 0, 1)$ is a Boolean algebra.*

Proof. (i) All complemented elements $x, y \in S$ satisfy $xx = x$ and $xy = yx$ as Prover9 easily shows. Also, B_S is closed under addition and multiplication. We show that, if x' and y' are complements of x and y (not necessarily unique), then $x'y'$ is a complement of $x + y$ and $x' + y'$ is a complement of xy. First,

$$\begin{aligned} x+y+x'y' &= x(y+y')+y(x+x')+x'y' \\ &= xy+xy'+yx+yx'+x'y' \\ &= xy+xy'+x'y+x'y' \\ &= (x+x')(y+y') \\ &= 1. \end{aligned}$$

Second, $(x+y)x'y' = xx'y'+yx'y' = yy'x' = 0$. The remaining cases then follow by duality.

Therefore, since $x \leq 1$ and by Lemma 5.2, B_S forms a distributive lattice. It is complemented because of the assumptions. But Boolean algebras are complemented distributive lattices. □

Theorem 6.5. *Let S be a domain semiring. Then $d(S)$ contains the greatest Boolean subalgebra of S bounded by* 0 *and* 1*.*

Proof. By Corollary 6.3 and Lemma 6.4, $B_S \subseteq d(S)$. □

Theorem 6.5 provides further interesting insights into the structure of domain algebras. In relation semirings and trace semirings, for instance, where the entire subalgebra of subidentities is a Boolean algebra, the domain algebra is fixed. It is the full algebra of subidentities, as expected.

7 Semiring Modules and Modal Semirings

It is well known that modal operators can be defined via domain and codomain operations [8]. Intuitively, the link is provided by the fact that Kripke frames are relational structures and forward and backward diamond operators correspond to relational preimage and image operations.

Here, more generally, we consider images and preimages with respect to semiring elements and (co)domain elements, that is, we consider modal diamond operators of sort $S \times d(S) \to d(S)$, but we define them more generally as operators of sort $S \times S \to S$ and use the fixpoint condition for explicitly encoding the sort of domain elements.

Let S be a domain semiring. For $x \in S$ and $p \in d(S)$, we define the *preimage* of p under x as $d(xp)$. The *image* of $p \in d^o(S)$ under x can be defined on the opposite semiring as $d^o(px)$ as well. Both $d(S)$ and $d^o(S)$ form a distributive lattice by Proposition 5.3 and its dual statement, but not necessarily the same one, unless $d(S)$ is a Boolean algebra and hence the greatest Boolean subalgebra of S bounded by 0 and 1.

We introduce the standard modal notation and write $\langle x \rangle p = d(xp)$, replacing the preimage operation by a multimodal diamond operator that acts on the domain algebra. In order to justify that these diamonds are indeed modal operators in the sense of the Boolean algebras with operators introduced by Jónsson and Tarski [14], we must show that $\lambda p.\langle x \rangle p$ is strict and additive. To link domain semirings even more strongly with computational algebras and logics, we present a more general result.

A *semiring module* [10] is a structure $(S, L, :)$, such that S is an idempotent semiring, L is a semilattice (with least upper bound operation $+$ and least element 0) and the scalar product : of sort $S \times L \to L$ satisfies, for all $x, y \in S$ and $p, q \in L$, the axioms

$$\begin{aligned}(x+y) : p &= x : p + y : p,\\ x : (p+q) &= x : p + x : q,\\ (xy) : p &= x : (y : p),\\ 1 : p &= p,\\ x : 0 &= 0.\end{aligned}$$

Proposition 7.1. *Let S be a domain semiring. Then $(S, d(S), (\lambda x, p.\langle x\rangle p))$ is a semiring module.*

Proof. It must be shown that $d((x+y)d(z)) = d(xd(z)) + d(yd(z))$ and similarly for the remaining four identities. This is routine work. □

An automated proof with Waldmeister is straightforward. The second and the fifth module axiom are additivity and strictness. So domain semirings indeed induce distributive lattices with operators à la Jónsson and Tarski.

In order to emphasise the fact that domain semirings give rise to modal operators, we also call these structures *modal semirings.*

To link the resulting modal algebras more strongly with logics of programs such as dynamic, temporal and Hoare logics, and in order to prepare the application in Section 12, a notion of iteration is needed.

A *Kleene algebra* is an idempotent semiring S extended by the *star operation* $^* : S \to S$ that satisfies the unfold and the induction axiom

$$1 + xx^* = x^* \qquad \text{and} \qquad y + xz \leq z \Rightarrow x^*y \leq z,$$

and their opposites [15]. A *Kleene algebra with domain* (also called *modal Kleene algebra*) is a domain semiring that is also a Kleene algebra. A *Kleene module* [17] is a semiring module $(K, L, :)$ over a Kleene algebra K that also satisfies, for all $x \in K$ and $p, q \in L$, the induction axiom

$$p + x : q \leq q \Rightarrow x^* : p \leq q.$$

Proposition 7.2. *Let K be a Kleene algebra with domain. Then the structure $(K, d(K), (\lambda x, p.\langle x\rangle p))$ is a Kleene module.*

Proof. By Proposition 7.1, K is a semiring module. It remains to show that

$$d(y) + d(xd(z)) \leq d(z) \Rightarrow d(x^*d(y)) \leq d(z)$$

holds for all Kleene algebras with domain. This proof is again routine and can be carried out along the lines of a similar proof with the DMS axioms [8]. □

An automated proof of the induction law by Prover9 took about 41 minutes and yielded a (resolution) proof with about 150 steps. Previous attempts to automatically prove this law with the DMS axiomatisation within reasonable time failed.

When L is a Boolean algebra, Kleene modules are essentially algebraic variants of propositional dynamic logics and the operators of the temporal logics LTL and CTL can of course be defined in this setting. In this sense, our Kleene algebras with domain yield propositional dynamic logics defined over distributive lattices of propositions. The absence of Boolean complementation certainly increases both the efficiency of proof search and the range of applications.

It is easy to show, using Mace4, that the module axioms are too weak to imply the domain axioms. To this end, of course, we must assume that the semilattice L contains a greatest element 1 in order to define $d(x) = x : 1$. It is then also easy to show that the module laws imply that $d^2(x) = d(x)$, that images of domain are closed under addition, but not under multiplication and that domain elements are not idempotent and do not commute with respect to multiplication. So, in the finite case, the structure induced by the image of domain defined via the scalar product is a complete semilattice, hence a lattice, in which meet and multiplication need not coincide. In the infinite case, the domain algebra induced is only a semilattice.

8 Boolean Domain Semirings

We now show how the domain algebra becomes a Boolean algebra when an appropriate antidomain function is added that simulates the effect of Boolean complementation.

A *Boolean domain semiring* is a domain semiring extended by a mapping $a : S \to S$ that satisfies the axioms

$$d(x) + a(x) = 1, \tag{10}$$

$$d(x)a(x) = 0. \tag{11}$$

Lemma 8.1. *Let S be a Boolean domain semiring. Then, for all $x \in S$,*

(i) $d(a(x)) = a(x)$, whence antidomain elements are domain elements;
(ii) $a(x)d(x) = 0$, whence $a(x)$ is the Boolean complement of $d(x)$;
(iii) $a^2(x) = d(x)$, whence domain can be defined from antidomain.

Proof. (i) We first show that $a(x)d(a(x)) = a(x)$ and that $d(x)d(a(x)) = 0$. For the first identity, $a(x)d(a(x)) \le d(a(x))$ holds since $a(x) \le 1$. The converse inequality holds, since

$$a(x) = (d(x) + a(x))a(x) = a(x)a(x) = a(x)d(a(x))a(x) \le a(x)d(a(x)).$$

The second identity holds, since, by domain export and (11),

$$d(x)d(a(x)) = d(d(x)a(x)) = d(0) = 0.$$

Then $d(a(x)) = (d(x) + a(x))d(a(x)) = d(x)d(a(x)) + a(x)d(a(x)) = a(x)$.

(ii) By (i), commutativity of domain elements and (11),

$$a(x)d(x) = d(a(x))d(x) = d(x)d(a(x)) = d(x)a(x) = 0.$$

(iii) By (10), (11) and (ii), $a(x)$ is the Boolean complement of $d(x)$, whence it satisfies double negation. □

An automated proof with Waldmeister required a few seconds, but the proof produced by the tool is far too long to be displayed.

The following proposition then follows from the facts of Section 6.

Proposition 8.2. *The domain algebra of a Boolean domain semiring is the maximal Boolean subalgebra of the semiring of subidentities.*

Proof. By Theorem 6.5 and Lemma 8.1, the maximal Boolean subalgebra of the subalgebra of subidentities and the domain algebra of a Boolean domain semiring must be the same. □

The above axioms for Boolean domain semirings can still considerably be simplified. First, $a^2 = d$ can be used for eliminating the domain function from the signature and basing the axiomatisation solely on antidomain. Second, the left annihilator property $a(x)x = 0$ of antidomain can be used instead of the left preserver property $a^2(x)x = x$. Experiments with Waldmeister and Mace4 led us to the following theorem.

Theorem 8.3. *A semiring S is a Boolean domain semiring if and only if it can be extended by an antidomain operation $a : S \to S$ that satisfies the axioms*

$$a(x)x = 0, \tag{BD1}$$

$$a(xy) + a(xa^2(y)) = a(xa^2(y)), \tag{BD2}$$

$$a^2(x) + a(x) = 1. \tag{BD3}$$

Again, the entire proof has been automated by Waldmeister and Prover9 and we encourage our readers to verify it using the templates in Appendix A and Appendix B. Again, also, the equational proof provided by Waldmeister is far too long and poorly structured to be displayed. An alternative equational proof can be found in Appendix D.

The first axiom is called *annihilation* axiom, the second one *locality* axiom and the third one *tertium non datur* axiom.

Lemma 8.4. *The axioms (BD1)-(BD3) are irredundant and irreducible.*

Proof. Irredundancy of (BD1) can be verified on the Boolean semiring with the following antidomain function for $x = 1$.

a	0	1
	1	1

$+$	0	1
0	0	1
1	1	1

$\cdot$	0	1
0	0	0
1	0	1

Irredundancy of (BD2) can be verified on the following 3-element semiring for $x = y = 2$.

a	0	1	2
	1	0	0

$+$	0	1	2
0	0	1	2
1	1	1	1
2	2	1	2

$\cdot$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	0

Irredundancy of (BD3) can again be verified again on the Boolean semiring with the antidomain function for $x = 0$.

a	0	1
	0	0

$+$	0	1
0	0	1
1	1	1

$\cdot$	0	1
0	0	0
1	0	1

Irreducibility of the axioms is obvious. □

The present axiomatisation is certainly much simpler and conceptually more appealing than the DMS axiomatisation. Most significantly, the previous need to explicitly specify and embed the Boolean algebra created a significant overhead for ATP systems. Hence the new axiomatisation is certainly the better choice for verification and program construction tasks.

9 Expressivity of Boolean Domain Semirings

We now verify that the new axiomatisation of Boolean domain semirings is precisely as expressive as the DMS axiomatisation. This allows us to reuse all theorems that have previously been derived and proves that the new axiomatisation is at least equally applicable.

Theorem 9.1. *Every theorem of domain semirings in the DMS axiomatisation is a theorem of Boolean domain semirings and vice versa.*

Proof. Let S be a Boolean domain semiring. Then $d(S)$ is a legitimate test algebra for the DMS axiomatisation (in particular, it is the maximal Boolean subalgebra of the algebra of subidentities by Proposition 6.5). Also, d is by definition of type $S \to d(S)$. We have already verified that the DMS domain axioms follow from the new ones via Lemma 4.1 and Theorem 8.3.

Let S be a domain semiring according to the DMS axioms and define, for each $x \in S$, $a(x) = d(x)'$. It has already been shown that (BD1) holds [8]. (BD2) holds by axiom (8). (BD3) holds by definition. □

So the whole development of modal semirings, modal Kleene algebras and Kleene modules that has previously been based on the two-sorted DMS approach can easily be reconstructed and reused.

Here we only mention that Theorem 9.1 also provides some important intuition about the antidomain operation. This is motivated by the fact that in relation and trace semirings, the antidomain $a(x)$ of an element x, which is the complement of the domain of x, is a *left annihilator* of x, that is, $a(x)x = 0$. More precisely, $a(x)$ is the greatest subidentity with that property (0 being the least one). Here, Theorem 9.1 immediately yields the following result.

Lemma 9.2. *For every Boolean domain semiring,*

(i) $y \leq a(x) \Rightarrow yx = 0$*, and*
(ii) $yx = 0 \Rightarrow y \leq a(x)$ *if* $y \leq 1$.

This can readily be checked with Prover9. It is also straightforward to show with Mace4 that the assumption $y \leq 1$ in (ii) cannot be relaxed.

Corollary 9.3. *On Boolean domain semirings, antidomain elements are greatest left annihilators in the set of subidentities.*

10 Antidomain

In this section we further consider the notion of antidomain.

First, some natural properties of antidomain arise by implicitly complementing the domain axioms (D1)-(D5). This yields the identities

$$a(x)x = 0, \tag{12}$$
$$a(xy) = a(xd(y)), \tag{13}$$
$$a(x) \leq 1, \tag{14}$$
$$a(0) = 1, \tag{15}$$
$$a(x + y) = a(x)a(y). \tag{16}$$

The following fact has been shown by Waldmeister.

Lemma 10.1. *The identities (12)-(16) hold in every Boolean domain semiring.*

However, Mace4 shows that the identities (12)-(16) and $d(a(x)) = a(x)$ are strictly weaker than the antidomain axioms.

Lemma 10.2.

(i) Some domain semiring satisfies (12)-(16), but not $d(a(x)) = a(x)$.
(ii) Some domain semiring satisfies (12)-(16), $d(a(x)) = a(x)$*, but not (BD3).*

(BD3) can, however, be proved if the double negation $d(x) = a^2(x)$ (which is also not implied), is added to the assumptions in (ii).

This weaker notion of antidomain characterised by the identities (12)-(16) suffices to prove some interesting properties which then hold a fortiori in Boolean domain semirings. All of them have been checked with Waldmeister.

Lemma 10.3. *Let S be a domain semiring. For all $x, y \in S$, the following properties follow from (12)-(16).*

(i) $a(x) \leq a(xy)$.
(ii) $a(x) = a(d(x))$.
(iii) $a(x)d(x) = 0$.
(iv) $a(x) \leq d(a(x))$.
(v) $a(x)a(x) = a(x)$.

(vi) $a(x)a(y) = a(y)a(x)$.
(vii) $a(x) + a(y) \leq a(d(x)y)$.
(viii) $a(1) = 0$.
(ix) $d(x) = a^2(x) \Rightarrow d(x)a(x) = 0$,
(x) $x \leq y \Rightarrow a(y) \leq a(x)$ *(antidomain is antitone).*

It can further be shown that, in this setting, antidomain elements need not be the Boolean complements of domain elements. Therefore, the interplay between domain and an antidomain operation axiomatised by (12)-(16) might be interesting in its own right and deserves further investigation.

Second, we analyse the relationship between domain semirings with antidomain defined via greatest left annihilation and Boolean domain semirings.

Lemma 10.4. *Some domain semiring satisfies the greatest left annihilator law*

$$y \leq 1 \Rightarrow (yx = 0 \Leftrightarrow y \leq a(x))$$

and $d(a(x)) = a(x)$*, but not* $d(x) = a^2(x)$*.*

This can easily be shown by Mace4. The following fact has been shown with Prover9.

Lemma 10.5. *Every domain semiring that satisfies the greatest left annihilator law,* $d(a(x)) = a(x)$ *and* $d(x) = a^2(x)$ *is a Boolean domain semiring.*

However, the resulting alternative axiomatisation of Boolean domain semirings is not equational and less compact than our standard one.

11 Heyting Domain Semirings

This section further demonstrates the flexibility offered by the new axiomatisation of domain semirings by considering domain algebras that are Heyting algebras. This can easily be enforced by adding the Galois connection

$$pq \leq r \Leftrightarrow p \leq q \rightarrow r, \tag{17}$$

where $p, q, r \in d(S)$, to the axioms of a domain semiring S. As usual, $q \rightarrow r$ is called the *relative pseudocomplement* of r with respect to q and the *pseudocomplement* $\neg p$ of $p \in d(S)$ is defined as $p \rightarrow 0$. Every lattice extended by an operation of relative pseudocomplementation such all elements satisfy (17) is a Heyting algebra by definition. Here, the arrow $\rightarrow$ is a partial function which is only defined on $d(S)$.

Now, since all finite distributive lattices are Heyting algebras, there is no finite domain algebra that violates the closure condition $d(p \rightarrow q) = p \rightarrow q$ for $p, q \in d(S)$. However, for the infinite case and for completeness we explicitly add this condition.

A *Heyting domain semiring* is a domain semiring S extended by an endofunction $\rightarrow$ on domain elements that satisfies the Galois connection (17) and the closure condition $d(p \rightarrow q) = p \rightarrow q$ for all $p, q \in S$.

Unfortunately, the partiality of the arrow makes the encoding in Mace4 rather delicate, since the tool requires that functions be totalised.

The following fact is an immediate consequence of standard properties of relative pseudocomplements [13].

Proposition 11.1. *A domain semiring S is a Heyting domain semiring if and only if all $p, q, r \in d(S)$ satisfy the following equations.*

$$p \to p = 1, \tag{HD1}$$

$$p(p \to q) = pq, \tag{HD2}$$

$$q(p \to q) = q, \tag{HD3}$$

$$p \to qr = (p \to q)(p \to r), \tag{HD4}$$

$$d(p \to q) = p \to q. \tag{HD5}$$

Again, by the results of Section 7, Heyting domain semirings give rise to modal semirings and modal Kleene algebras defined as Heyting algebras with operators; intuitionistic variants of dynamic logics and temporal logics can easily be defined in this setting. This is, however, beyond the scope of this paper.

Alternatively to the development of this section, one might try to replace $q \to r$, which can be expanded to $d(x) \to d(y)$, by $a(x) + d(y)$ in the Galois connection (17) or the identities axiomatising relative pseudocomplementation in Proposition 11.1. This would replace the arrow again by an antidomain function that is implicitly defined through these laws.

So first, let us add, for all elements x, y and z of a domain semiring S, the Galois connection

$$d(x)d(y) \leq d(z) \Leftrightarrow d(x) \leq a(y) + d(z) \tag{18}$$

to the domain semiring axioms. We also need to add the closure condition $d(a(x)) = a(x)$, since otherwise, by Mace4, antidomain elements need not be domain elements. However, these axioms are already too strong.

Proposition 11.2. *Every domain semiring that satisfies (18) and $d(a(x)) = a(x)$ is a Boolean domain semiring.*

This has been proved by Prover9; hence the approach based on (18) is no real alternative.

Second, therefore, we replace arrows by addition and antidomain in the identities of Proposition 11.1. But again, if we do not add $d(a(x)) = a(x)$, antidomain elements might not be domain elements, and, if we do, these axioms are too strong.

Lemma 11.3. *Let S be a domain semiring extended by an antidomain function $a : S \to S$ that, for all $x, y, z \in S$, satisfies the identities*

$$\begin{aligned} a(x) + d(x) &= 1, \\ d(x)(a(x) + d(y)) &= d(x)d(y), \\ d(y)(a(x) + d(y)) &= d(y), \\ a(x) + d(y)d(z) &= (a(x) + d(y))(a(x) + d(z)). \end{aligned}$$

hold. Then S is a Boolean domain semiring and $d(a(x)) = a(x)$.

This has been verified with Prover9.

All results of this section can easily be dualised into facts that hold in co-Heyting algebras.

12 Application: Termination Analysis

One of the most fundamental tasks in program analysis and construction is reasoning about termination and non-termination. Various supporting techniques have already been developed and applied in the context of the DMS approach. In particular, both the relational notion of well-foundedness and the modal notion of termination expressed by Löb's formula have been studied [7]. It has also been shown that termination analysis through Kleene algebras can effectively be automated [23], but the modal Kleene algebras based on the DMS axiomatisation so far present an unfortunate exception due to the general difficulties of automating them [11,23].

This section presents the first fully automated correspondence proof for Löb's formula. Such modal correspondence theorems generally associate validity of modal formulas with (relational) properties that hold on Kripke frames. Löb's formula, in particular, corresponds to the well-foundedness of transitive Kripke frames. Our automation result establishes Löb's formula and related expressions as automatically verified laws that can safely be used for termination analysis. Previous attempts to automate a manual proof based on the DMS axiomatisation [7] did not succeed [11].

It should be no surprise that a statement of comparable complexity cannot be proved in one full sweep from the set of axioms. We therefore use *hypothesis learning* [12]. We start with a basis of axioms from which some potentially dangerous axioms have been discarded. Axioms like $x + y = y + x$ or $1 + xx^* = x^*$, for instance, easily make the search space explode and distract the prover. Also, further potentially useful hypotheses often need to be given to the prover as lemmas. For reasons of consistency, all additional hypotheses should have previously been verified. By Theorem 9.1, we can freely use the large database of automatically verified theorems for the DMS axiomatisation [1]. At the beginning of the hypothesis learning phase, the hypotheses given are often too weak to entail the goal. Then, Mace4 can often find a counterexample indicating that the hypotheses need further strengthening. So more axioms and lemmas need to be added until Mace4 does not find a counterexample within reasonable time. Then Prover9 can be called. If a proof fails within reasonable time limits, another combination can be tried. This procedure is currently being implemented, but for the result of this section, hypotheses have still manually been learned. This is much simpler if, as in the present case, a manual proof is already known.

In its usual form, Löb's formula is written as $\Box(\Box p \to p) \to p$. To represent it algebraically, let K be a modal Kleene algebra and let $d(K)$ be the Boolean domain algebra of K. We first replace $\Box$ by $[x]$ and then dualise it to forward diamonds via $[x]p = (\neg\langle x\rangle\neg p)$, where $\neg$ now denotes Boolean complementation. Writing $p - q = p \cdot \neg q$, Löb's formula now becomes $\langle x\rangle p \leq \langle x\rangle(p - \langle x\rangle p)$.

In order to reason more concisely, we define $\Omega_x(p) = p - \langle x \rangle p$. In relation Kleene algebras, it denotes that subset of p from which no further x-transitions are possible, that is, the *final elements* of p with respect to x. Therefore, we say that an element x of K is *Löbian* [7] if

$$\langle x \rangle p \leq \langle x \rangle \Omega_x(p) \tag{19}$$

holds for all $p \in d(K)$. Intuitively, x is Löbian if all x-transitions lead into sets of x-maximal elements.

To set up the correspondence result, we also need to express well-foundedness and transitivity. We say that x is *well-founded* [7] if

$$\Omega_x(p) = 0 \Rightarrow p = 0 \tag{20}$$

holds for all $p \in d(K)$. Hence only the empty set has no x-maximal elements. Finally, x is *transitive* if $xx \leq x$ and *diamond transitive* (d-transitive) if $\langle x \rangle \langle x \rangle p \leq \langle x \rangle p$ holds for all $p \in d(K)$. Obviously, each transitive element is d-transitive, but the converse need not hold [7].

It has previously been shown that the following two facts hold on a modal Kleene algebra.

Theorem 12.1 ([7]). *Every Löbian element is well-founded.*

With our new axiomatisation, Prover9 could find a proof in a few seconds. The next theorem establishes the converse direction, hence the correspondence result. It is the main statement in this section.

Theorem 12.2. *Every well-founded d-transitive element is Löbian.*

To prove this theorem, the intermediate property $\langle x \rangle p \leq \langle x^+ \rangle \Omega_x(p)$, where $x^+ = xx^*$, has previously been introduced [7]. Here, we base the proof on a simpler property. We say that an element x is *pre-Löbian* if

$$p \leq \langle x^* \rangle \Omega_x(p). \tag{21}$$

Intuitively, if x is pre-Löbian, then every finite iteration of x will lead to x-maximal elements. It is obvious, but not important for the proof, that every pre-Löbian element x satisfies $\langle x \rangle p \leq \langle x^+ \rangle \Omega_x(p)$.

We now compare pre-Löbian elements and well-founded elements.

Proposition 12.3. *An element x is well-founded if and only if it is pre-Löbian.*

Proof. Let x be pre-Löbian and assume that $\Omega_x(p) = 0$. Then

$$p \leq \langle x^* \rangle \Omega_x(p) = \langle x^* \rangle 0 = 0.$$

Let now x be well-founded. x is pre-Löbian if $p - \langle x^* \rangle \Omega_x(p) = 0$, whence, by well-foundedness, it suffices to show that

$$p - \langle x^* \rangle \Omega_x(p) \leq \langle x \rangle (p - \langle x^* \rangle \Omega_x(p)), \tag{22}$$

since $p - q = 0 \Leftrightarrow p \le q$. In the calculation, we use the fact that

$$p - \mathit{\Omega}_x(p) = p - (p - \langle x \rangle p) = p \langle x \rangle p \le \langle x \rangle p. \tag{23}$$

We calculate

$$\begin{aligned} p - \langle x^* \rangle \mathit{\Omega}_x(p) &= p - (\mathit{\Omega}_x(p) + \langle x^+ \rangle \mathit{\Omega}_x(p)) \\ &= (p - \mathit{\Omega}_x(p)) - \langle x^+ \rangle \mathit{\Omega}_x(p) \\ &\le \langle x \rangle p - \langle x^+ \rangle \mathit{\Omega}_x(p) \\ &\le \langle x \rangle (p - \langle x^* \rangle \mathit{\Omega}_x(p)). \end{aligned}$$

The first step uses $x^* = 1 + x^+$ and $\langle 1 \rangle p = p$. The second step uses the fact $p - (q + r) = (p - q) - r$ from Boolean algebra. The third step uses (23). The fourth step uses $f(p) - f(q) \le f(p - q)$ which holds for all additive functions on a Boolean algebra, and $x^+ = xx^*$. □

Prover9 could show in a few seconds that pre-Löbian elements are well-founded; it could show in less than $20s$ that well-founded elements are pre-Löbian. This last proof requires a substantial amount of hypothesis learning, which of course makes the running times given less significant.

We can now prove the main theorem of this section, Theorem 12.2.

Proof. (of Theorem 12.2) Let x be well-founded. Then it is pre-Löbian by Proposition 12.3 and satisfies $\langle x \rangle p \le \langle x^+ \rangle \mathit{\Omega}_x(p) = \langle x \rangle \mathit{\Omega}_x(p)$, since $\langle x \rangle$ is isotone and x is d-transitive. Whence x is Löbian. □

For automating this proof, Prover9 could show almost instantaneously that d-transitivity implies $\langle x^+ \rangle p \le \langle x \rangle p$, when given the additional hypothesis $p + \langle x \rangle q \le q \Rightarrow \langle x^* \rangle p \le q$, which holds in modal Kleene algebras by Proposition 7.2. The converse direction, $\langle x \rangle p \le \langle x^+ \rangle p$, could be proved in less than one second, assuming only isotonicity of domain as an additional hypothesis. The resulting identity is needed in the second step of the proof. Theorem 12.2 could then been proved automatically in about one second, using the fact that well-founded elements are pre-Löbian, d-transitive elements satisfy $\langle x^+ \rangle p = \langle x \rangle p$ and isotonicity of domain. To decrease running times, the Kleene star axioms were also discarded. Proofs without hypothesis learning could possibly have been obtained with more patience.

13 Conclusion

This paper introduced new axiomatisations of domain semirings and Kleene algebras with domain. The approach is more general, simpler, more flexible and better suited for automated reasoning than the previous DMS axiomatisation. It is more general because the domain function is one-sorted. It is simpler because it drastically reduces the number and complexity of axioms needed. It is more flexible because different kinds of domain algebras can easily be obtained by different extensions of the basic set of axioms. Its superior suitability for automated deduction has been demonstrated on a non-trivial example from modal correspondence theory and many smaller proofs in the paper. These results contribute to a new approach to program verification and construction which combines computational algebras with automated deduction and aims at light-weight formal methods with heavy-weight automation.

The flexibility gained by the new axiomatisation opens further directions that remain to be explored.

First, the structure and applications of the distributive lattices and Heyting algebras with modal operators that arise in the setting of domain semirings and Kleene algebras with domain should be studied. Distributive lattices and Heyting algebras with operators have previously found applications in the area of many-valued logics and description logics. A comprehensive survey can be found in an article by Sofronie-Stokkermans [21]. Intuitionistic dynamic logics, which are based on Heyting algebras, have been developed by Degen and Werner [6]. Intuitionistic variants of the linear temporal logic LTL with applications in program verification, in particular for assume-guarantee reasoning and for properties that relate finite and infinite behaviour, have been studied by Maier [18].

In program analysis, domain elements can model tests in control structures such as conditionals and loops. The semiring semantics of the conditional if p then x else y, for instance, is $px + p'y$. On Boolean domain algebras, tests can only evaluate to true or to false. In more fine grained models, tests can also diverge or abort, which violates tertium non datur. Heyting algebras and similar lattices provide the appropriate semantics for these behaviours.

Second, the transfer of our results to weaker variants of semirings and Kleene algebras is important. A successor paper [9] shows that our domain axiomatisations can be reused for demonic refinement algebras [24] and probabilistic Kleene algebras [19] without any changes. Axiomatisations that are appropriate for modelling enabledness conditions for game-based semantics or basic process algebras require additional equations. The whole approach therefore encompasses a wide range of models and applications.

Last but not least, further case studies about programs, processes and other discrete systems need to be carried out to underpin the applicability of modal semirings and modal Kleene algebras in automated program verification, program construction and beyond.

Acknowledgement. We are most grateful to Peter Jipsen and Viorica Sofronie-Stokkermans for interesting discussions and references.

References

1. http://www.dcs.shef.ac.uk/~georg/ka
2. Prover9 and Mace4, http://www.cs.unm.edu/~mccune/prover9
3. Waldmeister, http://www.waldmeister.org
4. Birkhoff, G.: Lattice Theory. Colloquium Publications, vol. 25. American Mathematical Society (reprint, 1984)

5. De Carufel, J.-L., Desharnais, J.: Demonic Algebra with Domain. In: Schmidt, R.A. (ed.) RelMiCS/AKA 2006. LNCS, vol. 4136, pp. 120–134. Springer, Heidelberg (2006)
6. Degen, W., Werner, J.M.: Towards intuitionistic dynamic logic. In: Proceedings of Studia Logica 2006. Logic and Logical Philosophy, vol. 15, pp. 305–324. Nicolaus Copernicus University Press (2007)
7. Desharnais, J., Möller, B., Struth, G.: Termination in modal Kleene algebra. In: Lévy, J.-J., Mayr, E.W., Mitchell, J.C. (eds.) IFIP TCS 2004, pp. 647–660. Kluwer, Dordrecht (2004); Revised version: Algebraic Notions of Termination. Technical Report 2006-23, Institut für Informatik, Universität Augsburg (2006)
8. Desharnais, J., Möller, B., Struth, G.: Kleene algebra with domain. ACM Trans. Computational Logic 7(4), 798–833 (2006)
9. Desharnais, J., Struth, G.: Enabledness conditions for action systems, probabilistic systems, and processes. Technical Report CS-06-08, Department of Computer Science, University of Sheffield (2008)
10. Ésik, Z., Kuich, W.: A Semiring-Semimodule Generalization of ω-Context-Free Languages. In: Karhumäki, J., Maurer, H., Păun, G., Rozenberg, G. (eds.) Theory is Forever. LNCS, vol. 3113, pp. 68–80. Springer, Heidelberg (2004)
11. Höfner, P., Struth, G.: Automated Reasoning in Kleene Algebra. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 279–294. Springer, Heidelberg (2007)
12. Höfner, P., Struth, G.: Can refinement be automated? ENTCS 201, 197–222 (2008)
13. Johnstone, P.J.: Stone Spaces. Cambridge University Press, Cambridge (1982)
14. Jónsson, B., Tarski, A.: Boolean algebras with operators, Part I. American Journal of Mathematics 73, 891–939 (1951)
15. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Information and Computation 110(2), 366–390 (1994)
16. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. 19(3), 427–443 (1997)
17. Leiß, H.: Kleene modules and linear languages. Journal of Logic and Algebraic Programming 66(2), 185–194 (2006)
18. Maier, P.: Intuitionistic LTL and a New Characterization of Safety and Liveness. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 295–309. Springer, Heidelberg (2004)
19. McIver, A.K., Cohen, E., Morgan, C.C.: Using Probabilistic Kleene Algebra for Protocol Verification. In: Schmidt, R.A. (ed.) RelMiCS/AKA 2006. LNCS, vol. 4136, pp. 296–310. Springer, Heidelberg (2006)
20. Möller, B., Struth, G.: Algebras of modal operators and partial correctness. Theoretical Computer Science 351(2), 221–239 (2006)
21. Sofronie-Stokkermans, V.: Automated theorem proving by resolution in non-classical logics. Annals of Mathematics and Artificial Intelligence 49, 221–252 (2007)
22. Solin, K., von Wright, J.: Refinement Algebra with Operators for Enabledness and Termination. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 397–415. Springer, Heidelberg (2006)
23. Struth, G.: Reasoning automatically about termination and refinement. In: S. Ranise, editor, 6th International Workshop on First-Order Theorem Proving, Technical Report ULCS-07-018, Department of Computer Science, pp. 36–51. University of Liverpool (2007)
24. von Wright, J.: Towards a refinement algebra. Science of Computer Programming 51(1-2), 23–45 (2004)

A Axioms for Prover9/Mace4

```
op(500, infix, "+"). % addition
op(490, infix, ";"). % multiplication

formulas(sos).
% semiring axioms
    x+y=y+x.            % additive monoid
    x+0=x.
    x+(y+z)=(x+y)+z.
    x;1=x.              % multiplicative monoid
    1;x=x.
    x;(y;z)=(x;y);z.
    0;x=0.              % annihilation
    x;0=0.
    x;(y+z)=x;y+x;z.    % distributivity
    (x+y);z=x;z+y;z.
   %x+x=x.              % idempotency

% domain axioms
    x+d(x);x=d(x);x.
    d(x;y)=d(x;d(y)).
    d(x)+1=1.
    d(0)=0.
    d(x+y)=d(x)+d(y).

end_of_list.

formulas(goals).

    %add lemma to be proved/refuted

end_of_list.
```

B Axioms for Waldmeister

```
NAME            dsemiring

MODE            PROOF

SORTS           S

SIGNATURE           +: S S -> S
                    0:     -> S
                    *: S S -> S
                    1:     -> S
                    d:   S -> S

                  c1,c2,sk:     -> S
```

```
ORDERING        KBO
                +=1, 0=1, *=1, 1=1, c1=1, c2=1, sk=1, d=1
                * > + > d > 0 > 1 > c1 > c2 > sk

VARIABLES       x,y,z: S

EQUATIONS     % 1. semiring axioms

                    +(+(x,y),z) = +(x,+(y,z))
                         +(x,y) = +(y,x)
                         +(x,0) = x


                    *(*(x,y),z) = *(x,*(y,z))
                         *(x,0) = 0
                         *(0,x) = 0
                         *(x,1) = x
                         *(1,x) = x

                    *(x,+(y,z)) = +(*(x,y),*(x,z))
                    *(+(x,y),z) = +(*(x,z),*(y,z))

              % 2. domain axioms

                 +(x,*(d(x),x)) = *(d(x),x)
                      d(*(x,y)) = d(*(x,d(y)))
                      +(d(x),1) = 1
                           d(0) = 0
                      d(+(x,y)) = +(d(x),d(y))

CONCLUSION  % Equations are implicitly existentially quantified.
            % Universally quantified identities must be skolemised.

                       % add lemma to be proved
```

C Proof of Proposition 2.1 by Waldmeister

The following axioms imply the following theorem:

Axiom 1: $+(x1, x2) = +(x2, x1)$
Axiom 2: $*(x1, 1) = x1$
Axiom 3: $+(x1, *(d(x1), x1)) = *(d(x1), x1)$
Axiom 4: $+(d(x1), 1) = 1$
Theorem 1: $+(1, 1) = 1$

Proof:

Lemma 1: $d(1) = +(1, d(1))$

$d(1)$
$=$ by Axiom 2 RL
$*(d(1), 1)$
$=$ by Axiom 3 RL
$+(1, *(d(1), 1))$
$=$ by Axiom 2 LR
$+(1, d(1))$

Lemma 2: $+(1, d(x1)) = 1$

$+(1, d(x1))$
$=$ by Axiom 1 RL
$+(d(x1), 1)$
$=$ by Axiom 4 LR
1

Lemma 3: $1 = +(1, 1)$

1
$=$ by Lemma 2 RL
$+(1, d(1))$
$=$ by Lemma 1 LR
$+(1, +(1, d(1)))$
$=$ by Lemma 2 LR
$+(1, 1)$

Theorem 1: $+(1, 1) = 1$

$+(1, 1)$
$=$ by Lemma 3 RL
1

D Proof of Theorem 8.3

Proving this theorem automatically with Prover9 or Waldmeister requires only a few minutes. But the proof output of Prover9 is not intended to be readable for humans and the output of Waldmeister for the more involved proofs is far too detailed and badly structured to be revealing. We therefore provide alternative proofs. We set $d(x) = a^2(x)$.

We first show that the axioms (BD1)-(BD3) hold in all domain semirings that satisfy (10) and (11). For (BD1), we calculate

$$a(x)x \leq a(x)d(x)x = 0x = 0,$$

using (D1) in the first and Lemma 8.1(ii) in the second step. For (BD2) we first observe that

$$a(d(x)) = a(a(a(x))) = d(a(x)) = a(x)$$

by Lemma 8.1(iii) and (i). Then (BD2) follows immediately from (D2). Finally, (BD3) is an immediate consequence of Lemma 8.1(iii).

The more difficult part is to show that (D1)-(D5), (10) and (11) follow from (BD1)-(BD3). In particular the proofs of (D2) and (D5) are rather complex and require the development of several auxiliary facts.

(D1), (10) and (11), however, are immediate. For the equational version

$$x = d(x)x \tag{24}$$

of (D1), we use (BD1) and (BD3) to calculate

$$x = (a^2(x) + a(x))x = a^2(x)x + a(x)x = a^2(x)x + 0 = a^2(x)x.$$

(10) follows immediately from (BD3). (11) follows by $a^2(x)a(x) = 0$ from (BD1).

We now note that $a(1) = 0$ follows immediately from (BD1) and, by (BD3) this implies that $a(0) = 1$. Consequently, $d(0) = 0$, which is (D4), and $d(1) = 1$.

This implies that Boolean domain semirings are idempotent since, by (BD2), $1 + 1 = a(x0) + a(xd(0)) = a(xd(0)) = 1$ and thus (D3) follows from (BD3). Another consequence of idempotency and (BD3) is $a(x) \leq 1$, a property we will use freely from hereon.

It now remains to verify (D2) and (D5). We continue by proving some intermediate lemmas. The first one is

$$a(x) = 1 \Rightarrow x = 0, \tag{25}$$

which is direct by (BD1): $a(x) = 1 \Rightarrow a(x)x = x \Rightarrow x = 0$. The second one is

$$a(x)y = 0 \Leftrightarrow a(x) \leq a(y). \tag{26}$$

We prove the implication $\Rightarrow$ in two steps. Firstly, by $a(0) = 1$, (BD2) and (25),

$$a(x)y = 0 \Rightarrow a(a(x)y) = 1 \Rightarrow a(a(x)d(y)) = 1 \Rightarrow a(x)d(y) = 0.$$

Secondly, using (BD3) and the last equality of the previous line, we get

$$a(x) = a(x)(a(y) + d(y)) = a(x)a(y) + a(x)d(y) = a(x)a(y) \leq a(y).$$

The other implication $\Leftarrow$ follows by isotonicity and (BD1): $a(x)y \leq a(y)y = 0$.

We now have enough to prove (D2). By (BD1), $a(x)xy = 0y = 0$. Thus, by (26), $a(x) \leq a(xy)$. Using this with $x := xd(y)$ yields $a(xd(y)) \leq a(xd(y)y)$. By (24), this is just $a(xd(y)) \leq a(xy)$, and combining it with (BD2) gives (D2).

Before proving (D5), we again need two intermediate results. The first one is that antidomain is antitone, which implies that domain is isotone. By (BD1), $a(x+y)x \leq a(x+y)(x+y) = 0$, so that $a(x+y) \leq a(x)$ by (26). Since $x \leq y \Leftrightarrow x+y=y$, this is equivalent to antitonicity of a. The second result is

$$a(a(x)y) \leq d(x) + a(y). \tag{27}$$

We calculate

$$\begin{aligned} a(a(x)y) &= a(a(x)y)d(y) + a(a(x)y)a(y) \\ &\leq a(a(x)y)a(x)d(y) + a(a(x)y)d(x)d(y) + a(y) \\ &\leq a(a(x)d(y))a(x)d(y) + d(x) + a(y) \\ &= 0 + d(x) + a(y) \\ &= d(x) + a(y). \end{aligned}$$

The first and second steps use (BD3). The third uses (BD2) and the fourth (BD1).

We can now prove (D5), that is, $d(x+y) = d(x) + d(y)$. Since $d(x) + d(y) \leq d(x+y)$ holds by isotonicity, we only need to prove $d(x+y) \leq d(x) + d(y)$. Note that $a(x)a(y)(x+y) = 0$, because, by (BD1),

$$a(x)a(y)(x+y) = a(x)a(y)x + a(x)a(y)y \leq a(x)x + a(y)y = 0.$$

But, by (26), (BD2) and again (26),

$$\begin{aligned} a(x)a(y)(x+y) = 0 &\;\Rightarrow\; a(x) \leq a(a(y)(x+y)) \\ &\;\Rightarrow\; a(x) \leq a(a(y)d(x+y)) \\ &\;\Rightarrow\; a(x)a(y)d(x+y) = 0. \end{aligned}$$

Hence, using (BD1) in the first step and $a(x)a(y)d(x+y) = 0$ in the third,

$$\begin{aligned} a(x)a(y) &= a(x)a(y)(a(x+y) + d(x+y)) \\ &= a(x)a(y)a(x+y) + a(x)a(y)d(x+y) \\ &= a(x)a(y)a(x+y) \\ &\leq a(x+y). \end{aligned}$$

This result with antitonicity of a and (27) finally yield the missing part of (D5):

$$d(x+y) \leq a(a(x)a(y)) \leq d(x) + d(y).$$

This tedious development nicely demonstrates the extraordinary power of modern ATP systems, which are able to perform the same proof in a couple of minutes and without any typographical errors.

Asymptotic Improvement of Computations over Free Monads

Janis Voigtländer

Institut für Theoretische Informatik
Technische Universität Dresden
01062 Dresden, Germany
janis.voigtlaender@acm.org

Abstract. We present a low-effort program transformation to improve the efficiency of computations over free monads in Haskell. The development is calculational and carried out in a generic setting, thus applying to a variety of datatypes. An important aspect of our approach is the utilisation of type class mechanisms to make the transformation as transparent as possible, requiring no restructuring of code at all. There is also no extra support necessary from the compiler (apart from an up-to-date type checker). Despite this simplicity of use, our technique is able to achieve true asymptotic runtime improvements. We demonstrate this by examples for which the complexity is reduced from quadratic to linear.

1 Introduction

Monads [1] have become everyday structures for Haskell programmers to work with. Not only do monads allow to safely encapsulate impure features of the programming language [2,3], but they are also used in pure code to separate concerns and provide modular design [4,5]. But, as usual in software construction, modularity comes at a cost, typically with respect to program efficiency. We propose a method to improve the efficiency of code over a large variety of monads. A distinctive feature is that this method is non-intrusive: it preserves the appearance of code, with the obvious software engineering benefits.

Since our approach is best introduced by considering a concrete example, illustrating both the problem we address and our key ideas, that is exactly what we do in the next section. Thereafter, Sect. 3 develops the approach formally, embracing a generic programming style. Further example material is provided in Sects. 4 and 5, where the latter emphasises comparison to related work, before Sect. 6 concludes.

The code that we present throughout requires some extensions over the Haskell 98 standard, in particular rank-2 polymorphism and multi-parameter type constructor classes. It was tested against both GHC (version 6.6, flag `-fglasgow-exts`) and Hugs (version of March 2005, flag `-98`), and is available online at `http://wwwtcs.inf.tu-dresden.de/~voigt/Improve.lhs`.

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 388–403, 2008.

2 A Specific Example

We first study a simple and somewhat artificial example of the kind of transformation we want to achieve. This prepares the ground for the more generic development in the next section, and more practical examples later on.

Consider the following datatype of binary, leaf-labelled trees:

```
data Tree α = Leaf α | Node (Tree α) (Tree α)
```

An important operation on such trees is substituting leaves by trees depending on their labels, defined by structural induction as follows:

```
subst :: Tree α → (α → Tree β) → Tree β
subst (Leaf a) k = k a
subst (Node t₁ t₂) k = Node (subst t₁ k) (subst t₂ k)
```

Note that the type of labels might change during such a substitution.

It is well-known that trees with substitution form a monad. That is,

```
instance Monad Tree where
    return = Leaf
    (>>=) = subst
```

defines an instance of the following type constructor class:

```
class Monad μ where
    return :: α → μ α
    (>>=) :: μ α → (α → μ β) → μ β
```

where the following three laws hold:

$$(\text{return } a \gg\!= k) = k\ a \tag{1}$$

$$(m \gg\!= \text{return}) = m \tag{2}$$

$$((m \gg\!= k) \gg\!= h) = (m \gg\!= (\lambda a \rightarrow k\ a \gg\!= h)) \tag{3}$$

An example use of the monad instance given above is the following program generating trees like those in Fig. 1:

```
fullTree :: Int → Tree Int
fullTree 1 = Leaf 1
fullTree (n+1) =
    do
      i ← fullTree n
      Node (Leaf (n−i)) (Leaf (i+1))
```

Note that the second equation is equivalent to

```
fullTree (n+1) = fullTree n >>= λi → Node (Leaf (n−i)) (Leaf (i+1))
```

and thus to

```
fullTree (n+1) = subst (fullTree n) (λi → Node (Leaf (n−i)) (Leaf (i+1)))
```

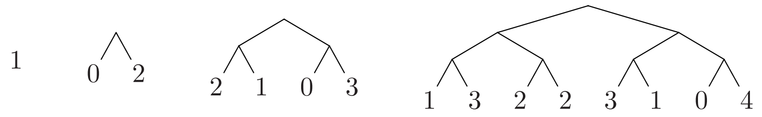


Fig. 1. fullTree 1, fullTree 2, fullTree 3, fullTree 4

This means that to create, for example, the tree fullTree 4, the following expression is eventually evaluated:

$$\mathsf{subst}\ (\mathsf{subst}\ (\mathsf{subst}\ (\textsc{Leaf}\ 1)\cdots)\cdots)\cdots \tag{4}$$

The nested calls to subst mean that prefix fragments of the overall tree structure will be traversed again and again.

In general, the asymptotic time complexity of computing fullTree n is of the order 2^n, which is no surprise as the size of the finally computed output is of that order as well. But a more interesting effect occurs when that output is only partially demanded, such as by the following function:

```
zigzag :: TREE INT → INT
zigzag = zig
   where
     zig (LEAF n) = n
     zig (NODE t₁ t₂) = zag t₁
     zag (LEAF n) = n
     zag (NODE t₁ t₂) = zig t₂
```

Now the expression

$$\mathsf{zigzag}\ (\mathsf{fullTree}\ n) \tag{5}$$

has *quadratic* runtime in n despite exploring only a single tree path of length *linear* in n. To see where the quadratic runtime comes from, consider the following partial reduction sequence for zigzag (fullTree 4), starting from (4) and having reordered a bit the lazy evaluation steps by first, and only, recording those involving subst:

```
      zigzag (subst (subst (subst (LEAF 1) ···) ···) ···)
 ⇒   zigzag (subst (subst (NODE (LEAF 0) (LEAF 2)) ···) ···)
 ⇒²  zigzag (subst (NODE (NODE (LEAF 2) (LEAF 1)) (subst (LEAF 2) ···)) ···)
 ⇒³  zigzag (NODE (NODE (subst (LEAF 2) ···) (NODE (LEAF 2) (LEAF 2))) ···)
 ⇒*  ···
```

The challenge is to bring the overall runtime for (5) down to linear, but to do so without changing the structure of the code for fullTree.

The situation here is similar to that in the well-known definition of naïve list reversal, where left-associatively nested appends cause quadratic runtime. And in fact there is a similar cure. We can create an alternative representation of trees somewhat akin to the “novel representation of lists” of Hughes [6], also known as difference lists. Just as the latter abstract over the end of a list, we abstract over the leaves of a tree as follows:

```
newtype CTree α = CTree (∀β. (α → Tree β) → Tree β)
```

The connection between ordinary trees and their alternative representation is established by the following two functions:

```
rep :: Tree α → CTree α
rep t = CTree (subst t)

abs :: CTree α → Tree α
abs (CTree p) = p Leaf
```

We easily have abs ∘ rep = id. Moreover, the representation type forms itself a monad as follows:

```
instance Monad CTree where
    return a = CTree (λh → h a)
    CTree p >>= k = CTree (λh → p (λa → case k a of CTree q → q h))
```

But to use it as a drop-in replacement for Tree in the definition of fullTree, the type constructor CTree need not only support the monad operations, but also the actual construction of (representations of) non-leaf trees. To capture this requirement, we introduce the following type constructor class:

```
class Monad μ ⇒ TreeLike μ where
    node :: μ α → μ α → μ α
```

For ordinary trees, the instance definition is trivial:

```
instance TreeLike Tree where
    node = Node
```

For the alternative representation, we have to take care to propagate the abstracted-over leaf replacement function appropriately. This is achieved as follows:

```
instance TreeLike CTree where
    node (CTree p1) (CTree p2) = CTree (λh → Node (p1 h) (p2 h))
```

For convenience, we also define an abstract version of the Leaf constructor.

```
leaf :: TreeLike μ ⇒ α → μ α
leaf = return
```

Now, we can easily give a variant of fullTree that is independent of the choice of trees to work with.

```
fullTree′ :: TreeLike μ ⇒ Int → μ Int
fullTree′ 1 = leaf 1
fullTree′ (n+1) =
    do
      i ← fullTree′ n
      node (leaf (n−i)) (leaf (i+1))
```

Note that the code structure is exactly as before. Moreover,

$$\mathsf{zigzag}\ (\mathsf{fullTree}'\ n) \tag{6}$$

still needs quadratic runtime. Indeed, GHC 6.6 with optimisation settings produces exactly the same compiled code for (5) and (6). Nothing magical has happened yet: any overhead related to the type class abstraction in (6) is simply optimised away. So there appears to be neither a gain from, nor a penalty for switching from fullTree to fullTree′. Why the (however small) effort, then?

The point is that we can now switch to an asymptotically more efficient version with almost zero effort. It is as simple as writing

$$\mathsf{zigzag}\ (\mathsf{improve}\ (\mathsf{fullTree}'\ n))\,, \tag{7}$$

where all the “magic” lies with the following function:

```
improve :: (∀ μ. TreeLike μ ⇒ μ α) → Tree α
improve m = abs m
```

In contrast to (5) and (6), evaluation of (7) has runtime only linear in n.

The rationale for the type of improve, as well as the correctness of the above transformation in the sense that (7) always computes the same output as (6), will be discussed in the next section, all for a more general setting than the specific type of trees and the example considered here.

We end the current section by pointing out that (7) is compiled (again by GHC 6.6) to code corresponding to

$$\mathsf{zigzag}\ (\mathsf{fullTree}''\ n\ \textsc{Leaf})\,,$$

where:

```
fullTree'' :: Int → (Int → Tree β) → Tree β
fullTree'' 1 h = h 1
fullTree'' (n+1) h = fullTree'' n (λi → Node (h (n−i)) (h (i+1)))
```

This should make apparent why the runtime is now only of linear complexity.

3 The Generic Setting

To deal with a variety of different datatypes in one stroke, we use the by now folklore approach of two-level types [7,8].

A *functor* is an instance of the following type constructor class:

```
class Functor φ where
    fmap :: (α → β) → φ α → φ β
```

satisfying the following two laws:

$$\mathsf{fmap}\ \mathsf{id}\ t = t \tag{8}$$

$$\mathsf{fmap}\ f\ (\mathsf{fmap}\ g\ t) = \mathsf{fmap}\ (f \circ g)\ t \tag{9}$$

Given such an instance, the corresponding *free monad* (capturing terms containing variables, along with a substitution operation) is defined as follows:

```
data Free φ α = Return α | Wrap (φ (Free φ α))

instance Functor φ ⇒ Monad (Free φ) where
    return = Return
    Return a ≫= k = k a
    Wrap t ≫= k = Wrap (fmap (≫= k) t)
```

Of course, we want to be sure that the laws (1)–(3) hold for the instance just defined. While law (1) is obvious from the definitions, the other two require fixpoint induction and laws (8) and (9).

As an example, consider the following functor:

```
data F β = N β β

instance Functor F where
    fmap h (N x y) = N (h x) (h y)
```

Then Free F corresponds to Tree from Sect. 2, and the monad instances agree.

Back to the generic setting. What was abstraction over leaves in the previous section, now becomes abstraction over the return method of a monad. This abstraction is actually possible for arbitrary, rather than only for free monads. The straight-forward definitions are as follows:

```
newtype C μ α = C (∀β. (α → μ β) → μ β)

rep :: Monad μ ⇒ μ α → C μ α
rep m = C (m ≫=)

abs :: Monad μ ⇒ C μ α → μ α
abs (C p) = p return

instance Monad (C μ) where
    return a = C (λh → h a)
    C p ≫= k = C (λh → p (λa → case k a of C q → q h))
```

Even though the monad laws do hold for the latter instance, we will not need this fact later on. What we will need, however, is the $\mathsf{abs} \circ \mathsf{rep} = \mathsf{id}$ property:

$$\begin{aligned}
&\mathsf{abs}\ (\mathsf{rep}\ m)\\
&\quad= \text{ by definition of } \mathsf{rep}\\
&\mathsf{abs}\ (\mathsf{C}\ (m \gg\!\!=))\\
&\quad= \text{ by definition of } \mathsf{abs}\\
&m \gg\!\!= \mathsf{return}\\
&\quad= \text{ by law (2) for the instance } \mathsf{Monad}\ \mu\\
&m
\end{aligned} \tag{10}$$

We also need to establish connections between the methods of the instances $\mathsf{Monad}\ \mu$ and $\mathsf{Monad}\ (\mathsf{C}\ \mu)$. For return, we have:

$$
\begin{array}{l}
\mathsf{rep}\ (\mathsf{return}\ a)\\
\quad = \ \text{by definition of } \mathsf{rep}\\
\mathsf{C}\ (\mathsf{return}\ a \gg\!=)\\
\quad = \ \text{by definition of sectioning}\\
\mathsf{C}\ (\lambda h \rightarrow \mathsf{return}\ a \gg\!= h)\\
\quad = \ \text{by law (1) for the instance } \mathsf{Monad}\ \mu\\
\mathsf{C}\ (\lambda h \rightarrow h\ a)\\
\quad = \ \text{by definition of } \mathsf{return} \text{ for the instance } \mathsf{Monad}\ (\mathsf{C}\ \mu)\\
\mathsf{return}\ a
\end{array}
\tag{11}
$$

Note that the occurrences of return in the first few lines refer to the instance $\mathsf{Monad}\ \mu$, whereas the return in the last line lives in the instance $\mathsf{Monad}\ (\mathsf{C}\ \mu)$. For the other method of the Monad class, we get the following distribution-like property:

$$
\begin{array}{l}
\mathsf{rep}\ (m \gg\!= k)\\
\quad = \ \text{by definition of } \mathsf{rep}\\
\mathsf{C}\ ((m \gg\!= k) \gg\!=)\\
\quad = \ \text{by definition of sectioning}\\
\mathsf{C}\ (\lambda h \rightarrow (m \gg\!= k) \gg\!= h)\\
\quad = \ \text{by law (3) for the instance } \mathsf{Monad}\ \mu\\
\mathsf{C}\ (\lambda h \rightarrow m \gg\!= (\lambda a \rightarrow k\ a \gg\!= h))\\
\quad = \ \text{by case-of-known}\\
\mathsf{C}\ (\lambda h \rightarrow m \gg\!= (\lambda a \rightarrow \mathbf{case}\ \mathsf{C}\ (k\ a \gg\!=)\ \mathbf{of}\ \mathsf{C}\ q \rightarrow q\ h))\\
\quad = \ \text{by definition of } \mathsf{rep}\\
\mathsf{C}\ (\lambda h \rightarrow m \gg\!= (\lambda a \rightarrow \mathbf{case}\ \mathsf{rep}\ (k\ a)\ \mathbf{of}\ \mathsf{C}\ q \rightarrow q\ h))\\
\quad = \ \text{by definition of } \gg\!= \text{ for the instance } \mathsf{Monad}\ (\mathsf{C}\ \mu)\\
\mathsf{C}\ (m \gg\!=) \gg\!= (\mathsf{rep} \circ k)\\
\quad = \ \text{by definition of } \mathsf{rep}\\
\mathsf{rep}\ m \gg\!= (\mathsf{rep} \circ k)
\end{array}
\tag{12}
$$

Next, we need support for expressing the construction of non-return values in both monads $\mathsf{Free}\ \phi$ and $\mathsf{C}\ (\mathsf{Free}\ \phi)$. To this end, we introduce the following multi-parameter type constructor class:

```
class (Functor φ, Monad μ) ⇒ FreeLike φ μ where
    wrap :: φ (μ α) → μ α
```

As in Sect. 2, one instance definition is trivial:

```
instance Functor φ ⇒ FreeLike φ (Free φ) where
    wrap = Wrap
```

The other one takes a bit more thinking, but will ultimately be justified by the succeeding calculations.

```
instance FreeLike φ μ ⇒ FreeLike φ (C μ) where
    wrap t = C (λh → wrap (fmap (λ(C p) → p h) t))
```

Similarly as for the monad methods before, we would like to prove distribution of rep over wrap, thus establishing a connection between instances FreeLike ϕ μ and FreeLike ϕ (C μ). More specifically, we expect rep (wrap t) $=$ wrap (fmap rep t). However, a straightforward calculation from both sides gets stuck somewhere in the middle as follows:

$$\begin{array}{l}
\mathsf{rep}\ (\mathsf{wrap}\ t)\\
\quad = \ \text{by definition of } \mathsf{rep}\\
\mathsf{C}\ (\mathsf{wrap}\ t \ggg\!=)\\
\quad = \ \text{by definition of sectioning}\\
\mathsf{C}\ (\lambda h \rightarrow \mathsf{wrap}\ t \ggg\!= h)\\
\quad = \ \text{by } \textbf{???}\\
\mathsf{C}\ (\lambda h \rightarrow \mathsf{wrap}\ (\mathsf{fmap}\ (\ggg\!= h)\ t))\\
\quad = \ \text{by definition of sectioning}\\
\mathsf{C}\ (\lambda h \rightarrow \mathsf{wrap}\ (\mathsf{fmap}\ (\lambda m \rightarrow m \ggg\!= h)\ t))\\
\quad = \ \text{by case-of-known}\\
\mathsf{C}\ (\lambda h \rightarrow \mathsf{wrap}\ (\mathsf{fmap}\ (\lambda m \rightarrow (\lambda(\mathsf{C}\ p) \rightarrow p\ h)\ (\mathsf{C}\ (m \ggg\!=)))\ t))\\
\quad = \ \text{by definition of } \mathsf{rep}\\
\mathsf{C}\ (\lambda h \rightarrow \mathsf{wrap}\ (\mathsf{fmap}\ (\lambda m \rightarrow (\lambda(\mathsf{C}\ p) \rightarrow p\ h)\ (\mathsf{rep}\ m))\ t))\\
\quad = \ \text{by law (9) for the instance } \textsf{Functor}\ \phi\\
\mathsf{C}\ (\lambda h \rightarrow \mathsf{wrap}\ (\mathsf{fmap}\ (\lambda(\mathsf{C}\ p) \rightarrow p\ h)\ (\mathsf{fmap}\ \mathsf{rep}\ t)))\\
\quad = \ \text{by definition of } \mathsf{wrap} \text{ for the instance } \textsf{FreeLike}\ \phi\ (\mathsf{C}\ \mu)\\
\mathsf{wrap}\ (\mathsf{fmap}\ \mathsf{rep}\ t)
\end{array}$$

On reflection, this is not so surprising, since it was to be expected that at some point we really need to consider the more specific Free ϕ versus C (Free ϕ) rather than the more general (and thus less informative) μ versus C μ as done for (10)–(12). Here now this point has come, and indeed we can reason for t of type ϕ (Free ϕ α) as follows:

$$\begin{array}{l}
\mathsf{rep}\ (\mathsf{wrap}\ t)\\
\quad = \ \text{as above}\\
\mathsf{C}\ (\lambda h \rightarrow \mathsf{wrap}\ t \ggg\!= h)\\
\quad = \ \text{by definition of } \mathsf{wrap} \text{ for the instance } \textsf{FreeLike}\ \phi\ (\textsf{Free}\ \phi)\\
\mathsf{C}\ (\lambda h \rightarrow \textsf{Wrap}\ t \ggg\!= h)\\
\quad = \ \text{by definition of } \ggg\!= \text{ for the instance } \textsf{Monad}\ (\textsf{Free}\ \phi)\\
\mathsf{C}\ (\lambda h \rightarrow \textsf{Wrap}\ (\mathsf{fmap}\ (\ggg\!= h)\ t))\\
\quad = \ \text{by definition of } \mathsf{wrap} \text{ for the instance } \textsf{FreeLike}\ \phi\ (\textsf{Free}\ \phi)\\
\mathsf{C}\ (\lambda h \rightarrow \mathsf{wrap}\ (\mathsf{fmap}\ (\ggg\!= h)\ t))\\
\quad = \ \text{as above}\\
\mathsf{wrap}\ (\mathsf{fmap}\ \mathsf{rep}\ t)
\end{array} \qquad (13)$$

Our "magic function" is again the same as abs up to typing:

```
improve :: Functor φ ⇒ (∀μ. FreeLike φ μ ⇒ μ α) → Free φ α
improve m = abs m
```

In fact, comparing their types should be instructive. Recall that

$$\mathsf{abs} :: \textsc{Monad}\ \mu \Rightarrow \mathsf{C}\ \mu\ \alpha \to \mu\ \alpha\,.$$

This type is different from that of improve in two ways. The first, and less essential, one is that abs is typed with respect to an arbitrary monad μ, whereas ultimately we want to consider the more specific case of monads of the form Free ϕ. Of course, by simple specialisation, abs admits the following type as well:

$$\mathsf{abs} :: \textsc{Functor}\ \phi \Rightarrow \mathsf{C}\ (\textsc{Free}\ \phi)\ \alpha \to \textsc{Free}\ \phi\ \alpha\,.$$

But, more essentially, the input type of improve is $\forall\mu.\ \textsc{FreeLike}\ \phi\ \mu \Rightarrow \mu\ \alpha$, which puts stronger requirements on the argument *m* than just C (Free ϕ) α would do. And that is what finally enables us to establish the correctness of adding improve at will wherever the type checker allows doing so. The reasoning, in brief, is as follows:

```
improve m
  =  by definition of improve
abs m
  =  by (11)–(13)
abs (rep m)
  =  by (10)
m
```

To understand in more detail what is going on here, it is particularly helpful to examine the type changes that *m* undergoes in the above calculation.

1. In the first line, *m* has the type $\forall\mu.\ \textsc{FreeLike}\ \phi\ \mu \Rightarrow \mu\ \alpha$ (for some fixed instance Functor ϕ and some fixed α), because that is what the type of improve forces it to be.
2. In the second line, *m* has the type C (Free ϕ) α, because that is what the type of abs forces it to be, taking into account that the overall expression in each line must have the type Free ϕ α. When going from left to right in the definition of improve, the type of *m* is thus specialised from $\forall\mu.\ \textsc{FreeLike}\ \phi\ \mu \Rightarrow \mu\ \alpha$ to C (Free ϕ) α. This is possible, and done silently (by the type checker), since an instance FreeLike ϕ (C (Free ϕ)) follows from the existing instance declarations Functor $\phi \Rightarrow$ FreeLike ϕ (Free ϕ) and FreeLike ϕ $\mu \Rightarrow$ FreeLike ϕ (C μ).
3. In the third line, *m* has the type Free ϕ α, because that is what the types of abs and rep force it to be. That type is an alternative specialisation of the original type $\forall\mu.\ \textsc{FreeLike}\ \phi\ \mu \Rightarrow \mu\ \alpha$ of *m*, possible due to the instance declaration Functor $\phi \Rightarrow$ FreeLike ϕ (Free ϕ). The key observation about the second versus third lines is that even though *m* has been type-specialised in two different ways, the *definition* (or value) of *m* is still the same as in the first

line. And since there it has the very general type $\forall \mu.\ \mathsf{FreeLike}\ \phi\ \mu \Rightarrow \mu\ \alpha$, we know that *m* cannot be built from any μ-related operations for any specific μ. Rather, its μ-structure must be made up from the overloaded operations return, $\gg\!=$, and wrap only. And since rep distributes over all of these by (11)–(13), we have

$$\mathsf{rep}\ (m :: \mathsf{Free}\ \phi\ \alpha) = (m :: \mathsf{C}\ (\mathsf{Free}\ \phi)\ \alpha)$$

A more formal proof would require techniques akin to those used for deriving so-called *free theorems* [9,10].

4. In the fourth line, *m* still has the type $\mathsf{Free}\ \phi\ \alpha$.

The essence of all the above is that improve *m* can be used wherever a value of type $\mathsf{Free}\ \phi\ \alpha$ is expected, but that *m* itself must (also) have the more general type $\forall \mu.\ \mathsf{FreeLike}\ \phi\ \mu \Rightarrow \mu\ \alpha$, and that then improve *m* is equivalent to just *m*, appropriately type-specialised. Or, put differently, wherever we have a value of type $\mathsf{Free}\ \phi\ \alpha$ which is constructed in a sufficiently abstract way (via the overloaded operators return, $\gg\!=$, and wrap) that it could also be given the type $\forall \mu.\ \mathsf{FreeLike}\ \phi\ \mu \Rightarrow \mu\ \alpha$, we can apply improve to that value without changing program semantics. Yet another perspective is that improve is simply a type conversion function that can replace an otherwise anyway, but silently, performed type specialisation and has the nice side effect of potentially improving the asymptotic runtime of a program (when left-associatively arranged calls to $\gg\!=$ cause quadratic overhead).

Having studied the generic setting, we can once more return to the specific example from Sect. 2. As already mentioned, the functor F given earlier in the current section yields Free F corresponding to Tree. Moreover, in light of the further general definitions from the current section, F and its functor instance definition also give us all the remaining ingredients of our improvement approach for binary, leaf-labelled trees. In particular, the type constructor C (Free F) corresponds to CTree, and FreeLike F takes the role of TreeLike. There is no need to provide any further definitions, since all the type constructor class instances that are needed are automatically obtained from the mentioned single one, and our generic definition of improve is similarly covering the earlier more specific one. In the next sections we will benefit from this genericity repeatedly.

4 A More Realistic Example

Swierstra and Altenkirch [11] build a pure model of Haskell's teletype IO, with the aim of enabling equational reasoning and automated testing. The monad they use for this corresponds to Free F_IO for the following functor:

```
data F_IO β = GetChar (Char → β) | PutChar Char β

instance Functor F_IO where
    fmap h (GetChar f) = GetChar (h ∘ f)
    fmap h (PutChar c x) = PutChar c (h x)
```

They then provide replacements of Haskell's getChar/putChar functions that produce pure values of this modelling type rather than doing actual IO. We can do so as well, catching up to List. 1 of [11].

```
getChar :: FreeLike F_IO μ ⇒ μ Char
getChar = wrap (GetChar return)

putChar :: FreeLike F_IO μ ⇒ Char → μ ()
putChar c = wrap (PutChar c (return ()))
```

The only differences of note are the more general return types of our versions of getChar and putChar. Just as the original function versions, our versions can be used to specify any interaction. For example, we can express the following computation:

```
revEcho :: FreeLike F_IO μ ⇒ μ ()
revEcho =
    do
      c ← getChar
      when (c ≠ ' ') $
        do
          revEcho
          putChar c
```

Run against the standard Haskell definitions of getChar and putChar (and obviously, then, with the different type signature revEcho :: IO ()), the above code reads characters from the input until a space is encountered, after which the sequence just read is written to the output in reverse order.

The point of Swierstra and Altenkirch's approach is to run the very same code against the pure model instead. Computing its (or similar functions') behaviour is done by a semantics they provide in their List. 2 and which is virtually replicated here (the only differences being two occurrences of Wrap):

```
data Output α = Read (Output α) | Print Char (Output α) | Finish α

data Stream α = Cons {hd :: α, tl :: Stream α}

run :: Free F_IO α → Stream Char → Output α
run (Return a) cs = Finish a
run (Wrap (GetChar f)) cs = Read (run (f (hd cs)) (tl cs))
run (Wrap (PutChar c p)) cs = Print c (run p cs)
```

Simulating a run of revEcho on some input stream, or indeed using QuickCheck [12] to analyse many such runs, takes the following form:

```
run revEcho stream .
```

It turns out that this requires runtime quadratic in the number of characters in *stream* before the first occurrence of a space. This holds both with our definitions and with those of [11]. So these two sets of definitions are not only equivalent with respect to the pure models and associated semantics they provide, but also in terms of efficiency. The neat twist in our setting, however, is that we can simply write

$$\mathsf{run}\ (\mathsf{improve}\ \mathsf{revEcho})\ \mathit{stream} \tag{14}$$

to reduce the complexity from quadratic to linear. The manner in which the quadraticity vanishes here is actually very similar to that observed for the "zigzag after fullTree" example at the end of Sect. 2, so we refrain from giving the code to which (14) is eventually compiled.

It is worth pointing out that the nicely general type of revEcho that makes all this possible could be automatically inferred from the function body if it were not for Haskell's dreaded monomorphism restriction. In fact, in GHC 6.6 we have the option of suppressing that restriction, in which case we need not provide the signature revEcho :: FreeLike F_IO $\mu \Rightarrow \mu$ (), and thus need not even be aware of whether we program against the pure teletype IO model in the incarnation of [11], our "magically improvable" variant of it, or indeed the standard Haskell IO monad.

5 Related Work

In this section we relate our work to two other strands of recent work that use two-level types in connection with monadic datatypes.

5.1 Structuring Haskell IO by Combining Free Monads

We have already mentioned the work by Swierstra and Altenkirch [11] on building pure models of (parts of) the Haskell IO monad. Apart from teletype IO, they also consider mutable state and concurrency. In both cases, the modelling type is a free monad and thus amenable to our improvement method. In a recent pearl [13, Sect. 7], Swierstra takes the modelling approach a step further. The free monad structure is used to combine models for different aspects of Haskell IO, and the models are not just used for reasoning and testing in a pure setting, but also for actual effectful execution. The idea is that the types derived for terms over the pure models are informative about just which kinds of effects can occur during eventual execution. Clearly, there is an interpretative overhead here, and somewhat startlingly this even affects the asymptotic complexity of programs.

For example, for teletype IO the required execution function looks as follows, referring to the original, effectful versions of getChar and putChar:

```
exec :: Free F_IO α → IO α
exec (Return a) = return a
exec (Wrap (GetChar f)) = Prelude.getChar >>= (exec ∘ f)
exec (Wrap (PutChar c p)) = Prelude.putChar c >> exec p
```

Now, main = exec revEcho unfortunately has quadratic runtime behaviour, very evident already via simple experiments with piping to the compiled version a text file with a few thousand initial non-spaces. This is in stark contrast to running revEcho (with alternative type signature revEcho :: IO ()) directly against the IO monad. Quite nicely, simply using main = exec (improve revEcho) recovers the linear behaviour as well. So thanks to our improvement method for free monads, which is orthogonal to Swierstra's "combination by coproducts" approach, we can have it both: pure modelling with informative types and efficient execution without (too big) interpretative overhead. Of course, our improvement also works for other cases of Swierstra's approach, such as his calculator example in Sect. 6. Up to compatibility with Agda's dependent type system, it should also apply to the models Swierstra and Altenkirch provide in [14] for computation on (distributed) arrays, and should reap the same benefits there.

5.2 Short Cut Fusion for Monadic Computations

Ghani et al. [15,16] observe that the augment combinator known from work on short cut fusion [17,18] has a monadic interpretation, and thus enables fusion for programs on certain monadic datatypes. This strand of work is thus the one most closely related to ours, since it also aims to improve the efficiency of monadic computations. An immediate difference is that Ghani et al.'s transformation can at best achieve a linear speedup, but no improvement of asymptotic complexity. More specifically, their approach does not allow for elimination of data structures threaded through repeated layers of monadic binding inside a recursive computation. Since the latter assertion seems somewhat in contradiction to the authors' description, let us elaborate on what we mean here.

First of all, the cases of successful fusion presented in [15,16] as examples all have the very specific form of a single consumer encountering a single producer, that is, eliminating exactly one layer of intermediate data structure. The authors suggest that sequences of ≫= arising from **do**-notation lead to a rippling effect that enables several layers to be eliminated in a row, but we could not reproduce this. In particular, Ghani and Johann [16, Sect. 5] suggest that this happens for the following kind of monadic evaluator:

```
data EXPR = ADD EXPR EXPR | ...

eval (ADD e1 e2) =
    do
      x ← eval e1
      y ← eval e2
      return (x+y)
...
```

But actually, the above right-hand side desugars to

$$\mathsf{eval}\ e_1 \gg\!= (\lambda x \rightarrow \mathsf{eval}\ e_2 \gg\!= (\lambda y \rightarrow \mathsf{return}\ (x{+}y))) \qquad (15)$$

rather than to an expression of the supposed form $(m \gg\!= k_1) \gg\!= k_2$. In fact, not a single invocation of the monadic short cut fusion rule is possible inside (15). In contrast, our improvement approach is quite effective for eval. If, for example, we complete the above to

```
data Expr = ... | Div Expr Expr | Lit Int

...
eval (Div e1 e2) =
     do
       y ← eval e2
       if y = 0 then fail "division by zero" else
         do
           x ← eval e1
           return (div x y)
eval (Lit i) = return i
```

and run it against the exception monad defined as follows:

```
data F_Exc β = Fail String

instance Functor F_Exc where
     fmap h (Fail s) = Fail s

fail s = wrap (Fail s)
```

then we find that while improve does not necessarily always give asymptotic improvements, it still reduces absolute runtimes here. Moreover, it turns out to have a beneficial impact on memory requirements. In particular, for expressions with deeply nested computations, such as

```
deep n = foldl Add (Div (Lit 1) (Lit 0)) (map Lit [2..n])
```

we find that improve (eval (deep n)) :: Free F_Exc Int works fine for n that are orders of magnitude bigger than ones for which eval (deep n) :: Free F_Exc Int already leads to a stack overflow. An intuitive explanation here is that improve essentially transforms the computation into continuation-passing style.

Clearly, just as for eval above, the monadic short cut fusion method proposed by Ghani et al. [15,16] does not help with any of the earlier examples in this paper. Maybe it is possible to bring it to bear on such examples by inventing a suitable worker/wrapper scheme in the spirit of that applied by Gill [17] and Chitil [19] to achieve asymptotic improvements via short cut fusion. If that can be achieved at all for monadic short cut fusion, which is somewhat doubtful due to complications involving polymorphic recursion and higher-kinded types, it would definitely require extensive restructuring of the code to be improved, much in contrast to our near-transparent approach.

On the other hand, Ghani et al.'s work is ahead of ours in terms of the monads it can handle. Their fusion rule is presented for a notion of inductive monads that covers free monads as a special case. More specifically, free monads are inductive monads that arise as fixpoints of a bifunctor that, when applied to one argument, gives the functor sum of the constant-valued functor returning that fixed argument and some other arbitrary, but fixed, functor. In other words, our FREE ϕ corresponds to MU (SUMFUNC ϕ) in the terminology of Ghani and Johann [16, Ex. 14]. Most fusion success stories they report are actually for this special kind of inductive monad and, as we have seen, all models of Swierstra and Altenkirch live in the free monad subspace as well. But still, it would be interesting to investigate a generalisation of our approach to inductive monads other than the free ones, in particular to ones based on functor product instead of functor sum above.

6 Conclusion

We have developed a program transformation that, in essence, makes monadic substitution a constant-time operation and can have further benefits regarding stack consumption. Using the abstraction mechanisms provided by Haskell's type system, we were able to formulate it in such a way that it does not interfere with normal program construction. In particular, programmers need not a priori decide to use the improved representation of free monads. Instead, they can program against the ordinary representation with the only (and non-encumbering) proviso that it be captured as one instance of an appropriate type constructor class. This gives code that is identically structured and equally efficient to the one they would write as usual. When utilisation of the improved representation is desired (for example, because a quadratic overhead is observed), dropping it in a posteriori is as simple as adding a single call to improve at the appropriate place. This transparent switching between the equivalent representations also means that any equational reasoning about the potentially to be improved code can be based on the ordinary representation, which is, of course, beneficial for applications like the ones of Swierstra and Altenkirch, discussed in Sect. 5.1. (Or, formulated in terms of the example from Sect. 2: we can reason and think about fullTree, as special case of fullTree′, even though actually fullTree″ will be run eventually.[1])

The genericity that comes via two-level types is a boon for developing and reasoning about our method, but not an indispensable ingredient. It is always possible to obtain type constructors, classes, and improve-functions tailored to a particular datatype (as in Sect. 2). This is done by bundling and unbundling type isomorphisms as demonstrated by Ghani and Johann [16, App. A].

Acknowledgements. I would like to thank the anonymous reviewers for their comments and suggestions.

[1] An example of a property that is much simpler to prove for fullTree than for fullTree″ is the fact that the output trees produced for input n will only ever contain integers from the interval 0 to n. While this has a straightforward proof by induction for fullTree n, proving it for fullTree″ n LEAF requires a nontrivial generalisation effort to find a good (i.e., general enough) induction hypothesis.

References

1. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991)
2. Peyton Jones, S., Wadler, P.: Imperative functional programming. In: Principles of Programming Languages, Proceedings, pp. 71–84. ACM Press, New York (1993)
3. Launchbury, J., Peyton Jones, S.: State in Haskell. Lisp and Symbolic Computation 8(4), 293–341 (1995)
4. Wadler, P.: The essence of functional programming. In: Principles of Programming Languages, Proceedings, pp. 1–14. ACM Press, New York (1992) (invited talk)
5. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Principles of Programming Languages, Proceedings, pp. 333–343. ACM Press, New York (1995)
6. Hughes, R.: A novel representation of lists and its application to the function "reverse". Information Processing Letters 22(3), 141–144 (1986)
7. Jones, M.: Functional programming with overloading and higher-order polymorphism. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 97–136. Springer, Heidelberg (1995)
8. Sheard, T., Pasalic, E.: Two-level types and parameterized modules. Journal of Functional Programming 14(5), 547–587 (2004)
9. Reynolds, J.: Types, abstraction and parametric polymorphism. In: Information Processing, Proceedings, pp. 513–523. Elsevier, Amsterdam (1983)
10. Wadler, P.: Theorems for free! In: Functional Programming Languages and Computer Architecture, Proceedings, pp. 347–359. ACM Press, New York (1989)
11. Swierstra, W., Altenkirch, T.: Beauty in the beast — A functional semantics for the awkward squad. In: Haskell Workshop, Proceedings, pp. 25–36. ACM Press, New York (2007)
12. Claessen, K., Hughes, R.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: International Conference on Functional Programming, Proceedings, pp. 268–279. ACM Press, New York (2000)
13. Swierstra, W.: Data types à la carte. Journal of Functional Programming (to appear)
14. Swierstra, W., Altenkirch, T.: Dependent types for distributed arrays. In: Trends in Functional Programming, Draft Proceedings (2008)
15. Ghani, N., Johann, P., Uustalu, T., Vene, V.: Monadic augment and generalised short cut fusion. In: International Conference on Functional Programming, Proceedings, pp. 294–305. ACM Press, New York (2005)
16. Ghani, N., Johann, P.: Monadic augment and generalised short cut fusion. Journal of Functional Programming 17(6), 731–776 (2007)
17. Gill, A.: Cheap Deforestation for Non-strict Functional Languages. PhD thesis, University of Glasgow (1996)
18. Johann, P.: A generalization of short-cut fusion and its correctness proof. Higher-Order and Symbolic Computation 15(4), 273–300 (2002)
19. Chitil, O.: Type-Inference Based Deforestation of Functional Programs. PhD thesis, RWTH Aachen (2000)

Symmetric and Synchronous Communication in Peer-to-Peer Networks

Andreas Witzel[1,2]

[1] University of Amsterdam, Plantage Muidergracht 24, 1018TV Amsterdam
[2] CWI, Kruislaan 413, 1098SJ Amsterdam, The Netherlands

Abstract. Motivated by distributed implementations of game-theoretical algorithms, we study symmetric process systems and the problem of attaining common knowledge between processes. We formalize our setting by defining a notion of peer-to-peer networks[1] and appropriate symmetry concepts in the context of Communicating Sequential Processes (*CSP*) [1]. We then prove that *CSP* with input and output guards makes common knowledge in symmetric peer-to-peer networks possible, but not the restricted version which disallows output statements in guards and is commonly implemented. Our results extend [2].

An extended version is available at `http://arxiv.org/abs/0710.2284`.

1 Introduction

1.1 Motivation

Our original motivation comes from the distributed implementation of game-theoretical algorithms (see e.g. [3] for a discussion of the interface between game theory and distributed computing). Two important issues in the domain of game theory have always been knowledge, especially common knowledge, and symmetry between the players, also called anonymity. We will describe these issues and the connections to distributed computing in the following two paragraphs, before we motivate our choice of process calculus and the overall goal of the paper.

Common Knowledge and Synchronization. The concept of common knowledge has been a topic of much research in distributed computing as well as in game theory. When do processes or players "know" some fact, mutually know that they know it, mutually know that they mutually know that they know it, and so on ad infinitum? And how crucial is the difference between arbitrarily, but finitely deep mutual knowledge and the limit case of real common knowledge?

In distributed computing, the classical example showing that the difference is indeed essential is the scenario of Coordinated Attack [4]. The game-theoretical incarnation of the underlying issue is the Electronic Mail Game [5,6].

[1] Please note that we are *not* dealing with fashionable incarnations such as file-sharing networks, but merely use this name for a mathematical notion of a network consisting of directly connected peers "treated on an equal footing", i.e. not having a client-server structure or otherwise pre-determined roles.

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 404–421, 2008.

The basic insight of these examples is that two agents that communicate through an unreliable channel can never achieve common knowledge, and that their behavior under finite mutual knowledge can be strikingly different.

These issues are analyzed in detail in [7], in particular in a separately published part [8], including a variant where communication is reliable, but message delivery takes an unknown amount of time. Even in that variant, it is shown that only finite mutual knowledge can be attained.

However, in a synchronous communication act, sending and receiving of a message is, by definition, performed simultaneously. In that way, the agents obtain not only the pure factual information content of a message, but the sender also knows that the receiver has received the message, the receiver knows that the sender knows that, and so on ad infinitum. The communicated information immediately becomes common knowledge.

Attaining common knowledge and achieving synchronization between processes are thus closely related. Furthermore, synchronization is in itself an important subject, see e.g. [9].

Symmetry and Peer-to-peer Networks. In game theory, players are assumed to be anonymous and treated on an equal footing, in the sense that their names do not play a role and no single player is a priori distinguished from the others [10,11].

In distributed computing, too, this kind of symmetry between processes is desirable to avoid a predetermined assignment of roles to processes and improve fault tolerance, modularity, and load balancing [12].

We will consider symmetry on two levels. Firstly, the communication network used by the processes should be symmetric to some extent in order not to discriminate single processes a priori on a topological level; we will formalize this requirement by defining peer-to-peer networks. Secondly, processes in symmetric positions of the network should have equal possibilities of behavior; this we will formalize in a semantic symmetry requirement on the possible computations.

Communicating Sequential Processes (CSP). Since we are interested in synchronization and common knowledge, a process calculus which supports synchronous communication through primitive statements clearly has some appeal. We will focus on one of the prime examples of such calculi, namely *CSP*, introduced in [1] and revised in [13,14], since it supports synchronous communication through primitive statements. Furthermore, it has been implemented in various programming languages, among the best-known of which is Occam [15]. We thus have at our disposal a theoretical framework and programming tools which in principle could give us synchronization and common knowledge "for free".

However, symmetric situations are a reliable source of impossibility results [16]. In particular, the restricted dialect CSP_{in} which was, for implementation issues [17], chosen to be the theoretical foundation of Occam is provably [2] less expressive than the general form, called $CSP_{i/o}$. CSP_{in} has been used throughout the history of Occam, up to and including its latest variant Occam-π [18]. This generally tends to be the case for implementations of *CSP*, one notable exception being a very recent extension [19] of JCSP[2] to $CSP_{i/o}$.

Some of the resulting restrictions of CSP_{in} can in practice be overcome by using helper processes such as buffers [20]. Our goal therefore is to formalize the concepts mentioned above, extend the notion of peer-to-peer networks by allowing helper processes, and examine whether synchronization is feasible in either of these two dialects of *CSP*. We will come to the result that, while the problem can (straightforwardly) be solved in $CSP_{i/o}$, it is impossible to do so in CSP_{in}. Our setting thus provides an argument in favor of the former's rare and admittedly more complicated implementations, such as JCSP.

1.2 Related Work

This paper builds upon [2], where a semantic characterization of symmetry for *CSP* is given and fundamental possibility and impossibility results for the problem of electing a leader in networks of symmetric processes are proved for various dialects of *CSP*. More recently, this has inspired a similar work on the more expressive π-calculus [21], but the possibility of adding helper processes is explicitly excluded.

There has been research on how to circumvent problems resulting from the restrictions of CSP_{in}. However, solutions are typically concerned only with the factual content of messages and do not preserve synchronicity and the common knowledge creating effect of communication, for example by introducing buffer processes [20].

The same focus on factual information holds for general research on synchronizing processes with asynchronous communication. For example, in [9] one goal is to ensure that a writing process knows that no other process is currently writing; whether this is common knowledge, is not an issue.

The problem of Coordinated Attack has also been studied for models in which processes run synchronously [16]; however, the interesting property of *CSP* is that processes run asynchronously, which is more realistic in physically distributed systems, and synchronize only at communication statements.

Since we focus on the communication mechanisms, the results will likely carry over to other formalisms with synchronous communication facilities comparable to those of *CSP*.

1.3 Overview of the Paper

In Section 2 we give a short description of *CSP* and the dialects that we are interested in, define some basic concepts from graph theory, and recall the required notions and results for symmetric electoral systems from [2].

In Section 3 we formally define the problem of pairwise synchronization that we will examine, give a formalization of peer-to-peer networks which ensures a certain kind of symmetry on the topological level, and describe in what ways

[2] A Java$^{\mathrm{TM}}$ implementation and extension of *CSP*.

we want to allow them to be extended by helper processes. We adapt a concept from [2] to capture symmetry on the semantic level.

Section 4 contains two positive results and the main negative result saying that pairwise synchronization of peer-to-peer networks of symmetric processes is not obtainable in CSP_{in}, even if we allow extensions through buffers or similar helper processes. Section 5 concludes.

2 Preliminaries

2.1 CSP

A *CSP process* consists of a sequential program which can use, besides the usual *local* statements, two *communication* statements:

- $P\,!\,message$ to send (output) the given message to process P;
- $P\,?\,variable$ to receive (input) a message from P into the given local variable.

Communication is *synchronous*, i.e., send and receive instructions block until their counterpart is available, at which point the message is transferred and both participating processes continue execution. Note that the communication partner P is statically defined in the program code.

There are two *control structures* (see Figure 1). Each guard is a Boolean expression over local variables (which, if omitted, is taken to be true), optionally followed by a communication statement. A guard is *open* if its Boolean expression evaluates to true and its communication statement, if any, can currently be performed. A guard is *closed* if its Boolean expression evaluates to false. Note that a guard can thus be neither open nor closed.

```
[  $guard_1 \rightarrow command_1$
□ $guard_2 \rightarrow command_2$
...
□ $guard_k \rightarrow command_k$  ]
```

(a) Non-deterministic selection

```
*[  $guard_1 \rightarrow command_1$
  □ $guard_2 \rightarrow command_2$
  ...
  □ $guard_k \rightarrow command_k$  ]
```

(b) Non-deterministic repetition

Fig. 1. Control structures in *CSP*

The selection statement *fails* and execution is aborted if all guards are closed. Otherwise execution is suspended until there is at least one open guard. Then one of the open guards is selected non-deterministically, the required communication (if any) performed, and the associated command executed.

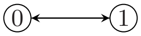
The repetition statement keeps waiting for, selecting, and executing open guards and their associated commands until all guards are closed, and then exits normally; i.e., program execution continues at the next statement.

We will sometimes use the following abbreviation to denote multiple branches of a control structure (for some finite set X): $\Box_{x \in X}\ guard_x \rightarrow command_x$

Various dialects of *CSP* can be distinguished according to what kind of communication statements are allowed to appear in guards. Specifically, in CSP_{in} only input statements are allowed, and in $CSP_{i/o}$ both input and output statements are allowed (within the same control structure). For technical reasons, CSP_{in} has been suggested from the beginning [1] and is indeed commonly used for implementations, as mentioned in Section 1.1.

Definition 1. *A* communication graph *(or* network*) is a directed graph without self-loops. A* process system *(or simply* system*) $\mathcal{P}$ with communication graph $G = (V, E)$ is a set of component processes $\{P_v\}_{v \in V}$ such that for all $v, w \in V$, if the program run by P_v (resp. P_w) contains an output command to P_w (resp. input command from P_v) then $(v, w) \in E$. In that case we say that G* admits *$\mathcal{P}$. We identify vertices v and associated processes P_v and use them interchangeably.*

Example 1. Figure 2 shows a simple network G with the vertex names written inside the vertices, and a $CSP_{i/o}$ program run by two processes which make up a system $\mathcal{P} := \{P_0, P_1\}$. Obviously, G admits $\mathcal{P}$. The intended behavior is that the processes send each other, in non-deterministic order, a message containing their respective process name.



(a) Network G

```
   $recd$ := false
   $sent$ := false
*[ $\neg recd \wedge P_{i+1} \,?\, x \rightarrow recd :=$ true
 □ $\neg sent \wedge P_{i+1} \,!\, i \rightarrow sent :=$ true ]
```

(b) Program of process P_i

Fig. 2. Network and program run by P_0 and P_1 in Example 1. Addition of process names here and in all further example programs is modulo 2.

Definition 2. *A* state *of a system $\mathcal{P}$ is the collection of all component processes' (local) variables together with their current execution positions. A* computation step *is a transition from one state to another, involving either one component process executing a local statement, or two component processes jointly executing a pair of matching (send and receive) communication statements. The valid computation steps are determined by the state of the system.*

A computation *is a maximal sequence of valid computation steps, i.e. a sequence which is not a prefix of any other sequence of valid computation steps. A computation*

- *is* properly terminated *if all component processes have completed their last instruction,*
- diverges *if it is infinite, and*
- *is in* deadlock *if it is finite but not properly terminated.*

2.2 Graph Theory

We state some fundamental notions concerning directed finite graphs, from here on simply referred to as graphs.

Definition 3. *Two vertices $a, b \in V$ of a graph $G = (V, E)$ are* strongly connected *if there are paths from a to b and from b to a; G is strongly connected if all pairs of vertices are. Two vertices $a, b \in V$ are* directly connected *if $(a, b) \in E$ or $(b, a) \in E$; G is directly connected if all pairs of vertices are.*

Definition 4. *An* automorphism *of a graph $G = (V, E)$ is a permutation σ of V such that for all $v, w \in V$, $(v, w) \in E$ implies $(\sigma(v), \sigma(w)) \in E$. The* automorphism group *Σ_G of a graph G is the set of all automorphisms of G. The least $p > 0$ with $\sigma^p = \mathrm{id}$ is called the* period *of σ, where by* id *we denote the identity function defined on the domain of whatever function it is compared to.*

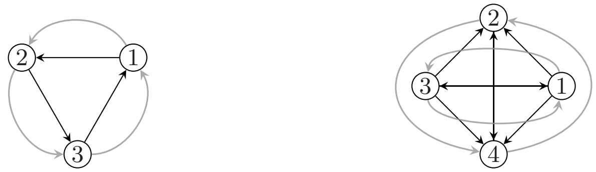
The orbit *of $v \in V$ under $\sigma \in \Sigma_G$ is $O_v^\sigma := \{\sigma^p(v) \mid p \geq 0\}$. An automorphism σ is* well-balanced *if the orbits of all vertices have the same cardinality, or alternatively, if for all $p \geq 0$,*

$$\sigma^p(v) = v \textit{ for some } v \in V \textit{ implies } \sigma^p = \mathrm{id} \quad .$$

We will usually consider the (possibly empty) set $\Sigma_G^{wb} \setminus \{\mathrm{id}\}$ of non-trivial *well-balanced automorphisms of a graph G, that is those with period greater than* 1.

A subset $W \subseteq V$ is called invariant *under $\sigma \in \Sigma_G$ if $\sigma(W) = W$; it is called invariant under Σ_G if it is invariant under all $\sigma \in \Sigma_G$.*

Example 2. Figure 3 shows two graphs G and H and well-balanced automorphisms $\sigma \in \Sigma_G$ with period 3 and $\tau \in \Sigma_H$ with period 2. We have $\Sigma_H = \{\mathrm{id}, \tau\}$, so $\{1, 3\}$ and $\{2, 4\}$ are invariant under Σ_H.



(a) Graph G, $\sigma \in \Sigma_G$

(b) Graph H, $\tau \in \Sigma_H$

Fig. 3. Two graphs with non-trivial well-balanced automorphisms, indicated by gray, bent arrows

2.3 Symmetric Electoral Systems

We take over the semantic definition of symmetry from [2]. As discussed there, syntactic notions of symmetry are difficult to formalize properly; requiring that "all processes run the same program" does not do the job. We will skip the formal details since we are not going to use them. The interested reader is referred to [2].

Definition 5 (adapted from [2, Definition 2.2.2]). *A system $\mathcal{P}$ with communication graph $G = (V, E)$ is* symmetric *if for each automorphism $\sigma \in \Sigma_G$ and each computation C of $\mathcal{P}$, there is a computation C' of $\mathcal{P}$ in which, for each $v \in V$, process $P_{\sigma(v)}$ performs the same steps as P_v in C, modulo changing via σ the process names occurring in the computation (e.g. as communication partners).*

The intuitive interpretation of this symmetry notion is as follows. Any two processes which are not already distinguished by the communication graph G itself, i.e. which are related by some automorphism, must have equal possibilities of behavior. That is, whatever behavior one process exhibits in some particular possible execution of the system (i.e., in some computation), the other process must exhibit in some other possible execution of the system, localized to its position in the graph by appropriate process renaming. Taken back to the syntactic level, this can be achieved by running the same program in both processes, which must not make use of any externally given distinctive features like, for example, an ordering of the process names.

Example 3. The system from Figure 2 is symmetric. It is easy to see that, if we swap all names 0 and 1 in any computation of $\mathcal{P}$, we still have a computation of $\mathcal{P}$. Note that programs are allowed to access the process names, and indeed they do; however, they do not, for example, use their natural order to determine which process sends first.

Example 4. On the other hand, the system $\mathcal{Q} = \{Q_0, Q_1\}$ where each Q_i runs the following program is not symmetric:

```
[  $i = 0 \rightarrow Q_{i+1} \,!\, i$
□ $i = 1 \rightarrow Q_{i+1} \,?\, x$  ]
```

We now recall a classical problem for networks of processes, and then restate the impossibility result which our paper builds on.

Definition 6 (from [2, Definition 1.2.1]). *A system $\mathcal{P}$ is an* electoral system *if*

(i) all computations of $\mathcal{P}$ are properly terminating and
(ii) each process of $\mathcal{P}$ has a local variable `leader`, and at the time of termination all these variables contain the same value, namely the name of some process $P \in \mathcal{P}$.

Theorem 1 (from [2, Theorem 3.3.2]). *Suppose a network G admits some well-balanced automorphism σ different from* id*. Then G admits no symmetric electoral system in CSP$_{in}$.*

3 Setting the Stage

3.1 Pairwise Synchronization

Intuitively, if we look at synchronization as part of a larger system, a process is able to synchronize with another process if it can execute an algorithm such that a direct communication (of any message) between the two processes takes place. This may be the starting point of some communication protocol to exchange more information, or simply be taken as an event creating common knowledge about the processes' current progress of execution.

Communication in *CSP* always involves exactly two processes and facilities for synchronous broadcast do not exist, thus synchronization is inherently pairwise only. This special case is still interesting and has been subject to research, see e.g. [22].

Focusing on the synchronization algorithm, we want to guarantee that it allows all pairs of processes to synchronize. To this end, we require existence of a system where in all computations, all pairs of processes synchronize. Most probably, in a real system not all pairs of processes need to synchronize in all executions. However, if one has an algorithm which in principle allows that, then one could certainly design a system where they actually do; and, vice versa, if one has a system which is guaranteed to synchronize all pairs of processes, then one can obviously use its algorithms to synchronize any given pair. Therefore we use the following formal notion.

Definition 7. *A system $\mathcal{P}$ of processes* (pairwise) synchronizes $\mathcal{Q} \subseteq \mathcal{P}$ *if all computations of $\mathcal{P}$ are finite and properly terminating and contain, for each pair $P_a, P_b \in \mathcal{Q}$, at least one direct communication from P_a to P_b or from P_b to P_a.*

Example 5. The system from Figure 2 synchronizes $\{P_0, P_1\}$.

Note that the program considered so far is not a valid CSP_{in} program, since there an output statement appears within a guard. If we want to restrict ourselves to CSP_{in} (for example, to implement the program in Occam), we have to get rid of that statement. Attempts to simply move it out of the guard fail since the symmetric situation inevitably leads to a system which may deadlock.

To see this, consider the system $\mathcal{P}' = \{P'_0, P'_1\}$ running the following program:

```
   $recd := \text{false}$
   $sent := \text{false}$
$*[$  $\neg recd \wedge P'_{i+1}\,?\,x \rightarrow recd := \text{true}$
   $\square\ \neg sent \rightarrow P'_{i+1}\,!\,i;\ sent := \text{true}$  $]$
```

There is no guarantee that not both processes enter the second clause of the repetition at the same time and then block forever at the output statement, waiting for each other to become ready for input. A standard workaround [20] for such cases is to introduce buffer processes mediating between the main processes, in our case resulting in the extended system $\mathcal{R} = \{R_0, R'_0, R_1, R'_1\}$:

```
   $recd := \text{false}$
   $sent := \text{false}$
$*[$  $\neg recd \wedge R'_{i+1}\,?\,x \rightarrow recd := \text{true}$            $R_i\,?\,y$
   $\square\ \neg sent \rightarrow R'_i\,!\,i;\ sent := \text{true}$  $]$            $R_{i+1}\,!\,y$
 (program of main process $R_i$)                  (program of buffer process $R'_i$)
```

While the actual data transmitted between the main processes remains the same, this system obviously cannot synchronize $\{R_0, R_1\}$, since there is no direct communication between them. This removes the synchronizing and common knowledge creating effects of communication. Mutual knowledge can only be achieved to a finite (if arbitrarily high) level, as discussed in Section 1.1.

The obvious question now is: Is it possible to change the program or use buffer or other helper processes in more complicated and smarter ways to negotiate between the main processes and aid them in establishing direct communications?

To attack this question, in the following Section 3.2 we will formalize the kind of communication networks we are interested in and define how they may be extended in order to allow for helper processes without affecting the symmetry inherent in the original network.

3.2 Peer-to-Peer Networks

The idea of peer-to-peer networks is to have nodes which can communicate with each other directly and on an equal footing, i.e. there is no predetermined client/server architecture or central authority coordinating the communication. We first formalize the topological prerequisites for this, and then adapt the semantic symmetry requirement to our setting.

Definition 8. *A* peer-to-peer network *is a communication graph $G = (V, E)$ with at least two vertices (also called nodes) such that*

(i) G is strongly connected,
(ii) G is directly connected, and
(iii) we have $\Sigma_G^{wb} \setminus \{\mathrm{id}\} \neq \emptyset$.

In this definition, (i) says that each node has the possibility to contact (at least indirectly) any other node, reflecting the fact that there are no predetermined roles; (ii) ensures that all pairs of nodes have a direct connection at least in one direction, without which pairwise synchronization by definition would be impossible; and (iii) requires a kind of symmetry in the network. This last item is implied by the more intuitive requirement that there be some $\sigma \in \Sigma_G$ with only one orbit, i.e. an automorphism relating all nodes to each other and thus making sure that they are topologically on an equal footing. The requirement we use is less restrictive and suffices for our purposes.

Example 6. See Figure 3 for two examples of peer-to-peer networks.

We will consider extensions allowing for helper processes while preserving the symmetry inherent in the network. We view the peers, i.e. the nodes of the original network, as processors each running a main process, while the added nodes can be thought of as helper processes running on the same processor as their respective main process.

Definition 9. *Let $G = (V, E)$ be a peer-to-peer network, then $G' = (V', E')$ is a* symmetry-preserving extension *of G iff there is a collection $\{S_v\}_{v \in V}$ partitioning V' such that*

(i) for all $v \in V$, we have $v \in S_v$;
(ii) all $v \in V$, $v' \in S_v \setminus \{v\}$ are strongly connected (possibly via nodes $\notin S_v$);
(iii) for all $v, w \in V$, $E' \cap (S_v \times S_w) \neq \emptyset$ iff $(v, w) \in E$;
(iv) there is, for each $\sigma \in \Sigma_G$, an automorphism $\iota_\sigma \in \Sigma_{G'}$ extending σ such that $\iota_\sigma(S_v) = S_{\sigma(v)}$ for all $v \in V$.

Remark 1. In general, the collection $\{S_v\}_{v\in V}$ may not be unique. When we refer to it, we implicitly fix an arbitrary one.

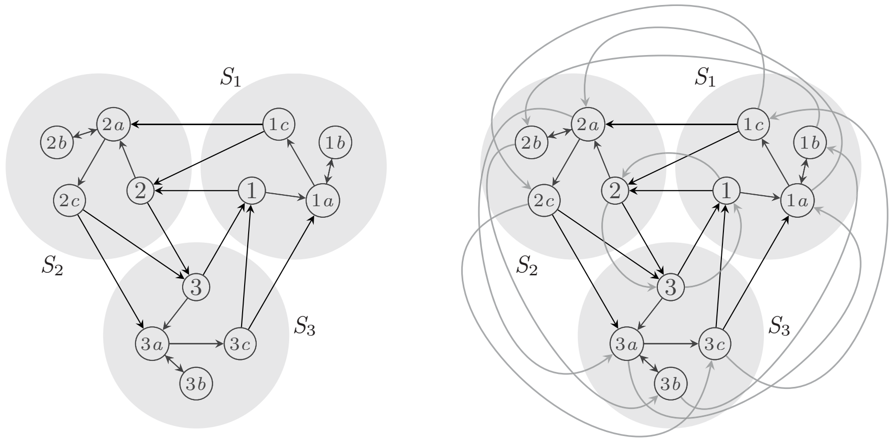
Intuitively, these requirements are justified as follows:

(i) Each S_v can be seen as the collection of processes running on the processor at vertex v, including its main process P_v.
(ii) The main process should be able to communicate (at least indirectly) in both ways with each helper process.
(iii) While communication links within one processor can be created freely, links between processes on different processors are only possible if there is a physical connection, that is a connection in the original peer-to-peer network; also, if there was a connection in the original network, then there should be one in the extension in order to preserve the network structure.
(iv) Lastly, to preserve symmetry, each automorphism of the original network must have an extension which maps all helper processes to the same processor as their corresponding main process.

Example 7. See Figure 4 for an example of a symmetry-preserving extension. Note that condition (iii) of Definition 9 is liberal enough to allow helper processes to communicate directly with processes running on other processors, and indeed, e.g. $2c$ has a link to 3. It also allows several communication links on one physical connection, reflected by the fact that there are three links connecting S_2 to S_3.

We will need the following immediate fact later on.

Fact 1. *As a direct consequence of Definitions 8 and 9, any symmetry-preserving extension of a peer-to-peer network is strongly connected.*



(a) Symmetry-preserving extension of the network from Figure 3(a)

(b) Extended automorphism ι_σ as required by Definition 9

Fig. 4. A symmetry-preserving extension (illustrating Definition 9)

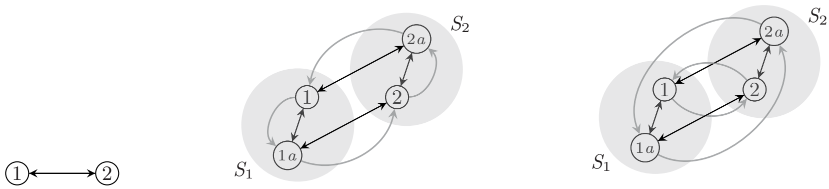
3.3 G-Symmetry

Corresponding to the intuition of processors with main and helper processes, we weaken Definition 5 such that only automorphisms are considered which keep the set of main processes invariant and map helper processes to the same processor as their main process. There are cases (as in Figure 8 later in this paper) where the main processor otherwise would be required to run the same program as some helper process.

Definition 10 (G-symmetry). *A system $\mathcal{P}$ whose communication graph G' is a symmetry-preserving extension of some peer-to-peer network $G = (V, E)$ is called* G-symmetric *if Definition 5 holds with respect to those automorphisms $\sigma \in \Sigma_{G'}$ satisfying, for all $v \in V$, (i) $\sigma(V) = V$ and (ii) $\sigma(S_v) = S_{\sigma(v)}$.*

This is weaker than Definition 5, since there we require the condition to hold for all automorphisms.

Example 8. To illustrate the impact of G-symmetry, Figure 5 shows a network G and an extension where symmetry relates all processes which each other. G-symmetry disregards the automorphism which causes this and considers only those which keep the set of main processes invariant, i.e. the nodes of the original network G, thus allowing them to behave differently from the helper processes.



(a) Network G

(b) Extension of G and an automorphism mixing main and helper processes

(c) Extension of G and the only automorphism taken into account by G-symmetry

Fig. 5. A network G and an extension which has an automorphism mixing main and helper processes, disregarded by G-symmetry

4 Results

4.1 Positive Results

Theorem 2. *Let $G = (V, E)$ be a peer-to-peer network. Then G admits a symmetric system pairwise synchronizing V in $CSP_{i/o}$.*

Proof. A system which at each vertex $v \in V$ runs the program shown below is symmetric and pairwise synchronizes V. Each process simply waits for each other process in parallel to become ready to send or receive a dummy message, and exits once a message has been exchanged with each other process.

```
  for each $w \in V$ do $sync_w := \text{false}$
  $W_{in} := \{w \in V \mid (w, v) \in E\}$
  $W_{out} := \{w \in V \mid (v, w) \in E\}$
*[
  $\Box_{w \in W_{in}}\ \neg sync_w \wedge P_w\,?\,x \rightarrow sync_w := \text{true}$
  $\Box_{w \in W_{out}}\ \neg sync_w \wedge P_w\,!\,0 \rightarrow sync_w := \text{true}$
  ]
```

□

By dropping the topological symmetry requirement for peer-to-peer networks, under certain conditions we get a positive result even for CSP_{in}.

Theorem 3. *Let $G = (V, E)$ be a network satisfying only the first two conditions of Definition 8, i.e. G is strongly connected and directly connected. If G admits a symmetric electoral system and there is some vertex $v \in V$ such that $(v, a) \in E$ and $(a, v) \in E$ for all $a \in V$, then G admits a symmetric system pairwise synchronizing V in CSP_{in}.*

Proof (sketch). First, the electoral system is run to determine a temporary leader v'. When the election has terminated, v' chooses a coordinator v that is directly and in both directions connected to all other vertices, and broadcasts its name. Broadcasting can be done by choosing a spanning tree and transmitting the broadcast information together with the definition of the tree along the tree, as in the proof of [2, Theorem 2.3.1, Phase 2] (the strong connectivity which is required there holds for G by assumption). After termination of this phase, the other processes each send one message to v and then wait to receive commands from v according to which they perform direct communications with each other, while v receives one message from each other process and uses the obtained order to send out the commands. □

Example 9. See Figure 6 for an example of a network which admits a symmetric system pairwise synchronizing all its vertices in CSP_{in}. The fact that the network admits a symmetric electoral system can be established as for [2, Fig. 4] (note that the edges between the lower nodes are only in one direction).

This result could be generalized, e.g. by weakening the conditions on v and taking care that the commands will reach the nodes at least indirectly. Since our main focus is the negative result, we will not pursue this further.

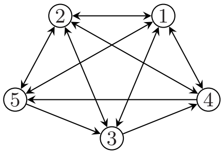


Fig. 6. A network which by Theorem 3 admits a symmetric system pairwise synchronizing all its vertices in CSP_{in}.

4.2 Negative Result

In the following we will establish the main result saying that, even if we extend a peer-to-peer network G by helper processes (in a symmetry-preserving way), it is not possible to obtain a network which admits a G-symmetric system pairwise synchronizing the nodes of G in CSP_{in}.

To this end, we derive a contradiction with Theorem 1 by proving the following intermediate steps (let G denote a peer-to-peer network and G' a symmetry-preserving extension):

- Lemma 1 If G' admits a G-symmetric system pairwise synchronizing the nodes of G in CSP_{in}, it admits a G-symmetric electoral system in CSP_{in}.
- Lemma 2 G' has a non-trivial well-balanced automorphism taken into account by G-symmetry (i.e. satisfying the two conditions of Definition 10).
- Lemma 3 We can extend G' in such a way that there exists a non-trivial well-balanced automorphism (derived from the previous result), G-symmetry is reduced to symmetry, and admittance of an electoral system is preserved.

Lemma 1. *If some symmetry-preserving extension of a peer-to-peer network $G = (V, E)$ admits a G-symmetric system pairwise synchronizing V in CSP_{in}, then it admits a G-symmetric electoral system in CSP_{in}.*

Proof. The following steps describe the desired electoral system (using the fact that under G-symmetry processes of nodes $\in V$ may behave differently from those of nodes $\notin V$):

- All processes run the assumed G-symmetric pairwise synchronization program, with the following modification for the processes in $\mathcal{P} := \{P_v \mid v \in V\}$ (intuitively this can be seen as a kind of knockout tournament, similar to the proof of [2, Theorem 4.1.2, Phase 1]):
  - Each of these processes has a local variable `winning` initialized to `true`.
  - After each communication statement with some other $P \in \mathcal{P}$, insert a second communication statement with P in the same direction:
    - If it was a "send" statement, send the value of `winning`.
    - If it was a "receive" statement, receive a Boolean value, and if the received value is `true`, set `winning` to `false`.

  Note that, since the program pairwise synchronizes V, each pair of processes associated to vertices in V has had a direct communication at the end of execution, and thus there is exactly one process in the whole system which has a local variable `winning` containing `true`.
- After the synchronization program terminates the processes check their local variable `winning`. The unique process that still has value `true` declares itself the leader and broadcasts its name; all processes set their variable `leader` accordingly. As in the proof of Theorem 3, broadcasting can be done using a spanning tree. The required strong connectivity is guaranteed by Fact 1. □

Lemma 2. *For any symmetry-preserving extension $G' = (V', E')$ of a peer-to-peer network $G = (V, E)$, there is $\sigma' \in \Sigma^{wb}_{G'} \setminus \{\mathrm{id}\}$ such that $\sigma'(V) = V$ and $\sigma'(S_u) = S_{\sigma'(u)}$ for all $u \in V$.*

Proof. Take an arbitrary $\sigma \in \Sigma_G^{wb} \setminus \{\mathrm{id}\}$ (exists by Definition 8) and let ι, to save indices, denote the ι_σ required by Definition 9. If $\iota \in \Sigma_{G'}^{wb} \setminus \{\mathrm{id}\}$ we are done; otherwise we construct a suitable σ' (Example 10 illustrates this proof).

Let p denote the period of σ and pick an arbitrary $v \in V$. For simplicity, we assume that σ has only one orbit; if it has several, the proof extends straightforwardly by picking one v from each orbit in parallel.

For all $u \in S_v$ let $p_u := |O_u^\iota|$ and note that for all $t \in O_u^\iota$ we have $p_t = p_u$, and $p_u \geq p$ since ι maps each S_v to $S_{\sigma(v)}$ and these sets are pairwise disjoint. We define $\sigma' : V' \to V'$ and then prove the claims.

$$\sigma'(u) := \begin{cases} \iota^{p_u - p + 1}(u) & \text{if } u \in S_v \\ \iota(u) & \text{otherwise.} \end{cases}$$

- $\sigma'(V) = V$, $\sigma' \neq \mathrm{id}$: Follows from $\iota \restriction_V = \sigma$ and $p_v = p$ and thus $\sigma' \restriction_V = \sigma$ (where $f \restriction_X$ denotes the restriction of a function f to the domain X)
- $\sigma' \in \Sigma_{G'}$: With (iv) from Definition 9 we obtain that, for $u \in S_v$, p_u must be a multiple of p, and $\sigma'(O_u^\iota \cap S_v) = \iota(O_u^\iota \cap S_v)$, thus σ' is a permutation of V' since ι is one. Furthermore, for $t, u \in S_v$, we have $\iota^{p_t(p_u-1)}(t) = t$ and $\iota^{p_u(p_t-1)}(u) = u$ and therefore σ' also inherits edge-preservation from ι by
$$(\sigma'(t), \sigma'(u)) = (\iota^{p_t - p + 1}(t), \iota^{p_u - p + 1}(u)) = (\iota^{p_t p_u - p + 1}(t), \iota^{p_t p_u - p + 1}(u)) \ .$$
- $\sigma'(S_u) = S_{\sigma'(u)}$, σ' well-balanced: The above-mentioned fact that for all $u \in S_v$ we have $\sigma'(O_u^\iota \cap S_v) = \iota(O_u^\iota \cap S_v)$, together with (iv) from Definition 9 implies that also $\sigma'(S_u) = S_{\sigma(u)}$ for all $u \in V$. For all $v' \in V'$, well-balancedness of σ and disjointness of the S_u imply that $\sigma'^q(v') \neq v'$ for $0 < q < p$. On the other hand, since each orbit of σ has size p and contains exactly one element from S_v (namely v), we have that
$$\begin{aligned} \sigma'^p(v') &= \iota^{(p_u - p + 1) + (p-1)}(v') \qquad \text{for some } u \in O_{v'}^\iota \\ &= \iota^{p_u}(v') = \iota^{p_{v'}}(v') = v' \ . \end{aligned}$$
□

Example 10. In Figure 7(a), we have $p = 2$ (the period of $\sigma = \iota_\sigma \restriction_{\{1,2\}}$), and we pick vertex $v = 2$. For the elements of S_2, we obtain $p_2 = p = 2$ and $p_{2a} = p_{2b} = p_{2c} = 6$. Thus σ' is defined as follows:

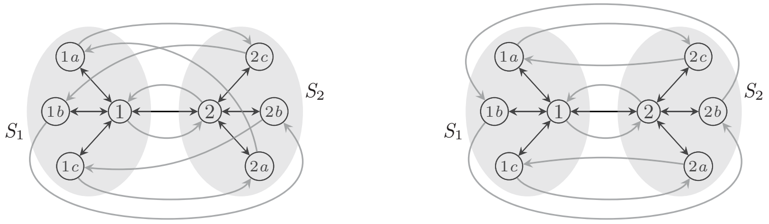
$$\sigma'(u) = \begin{cases} \iota(u) & \text{if } u = 2 \\ \iota^5(u) & \text{if } u \in S_2 \setminus \{2\} \\ \iota(u) & \text{if } u \in S_1 \ . \end{cases}$$

This σ', depicted in Figure 7(b), satisfies the claims of Lemma 2.

Lemma 3. *Any symmetry-preserving extension $G' = (V', E')$ of a peer-to-peer network $G = (V, E)$ can be extended to a network H such that*

(i) $\Sigma_H^{wb} \setminus \{\mathrm{id}\} \neq \emptyset$, *and*

(ii) if G' admits a G-symmetric electoral system in CSP$_{in}$,
then H admits a symmetric electoral system in CSP$_{in}$.



(a) ι_σ as required by Definition 9 (b) σ' constructed from ι_σ as in Lemma 2

Fig. 7. An extended peer-to-peer network G' illustrating Lemma 2

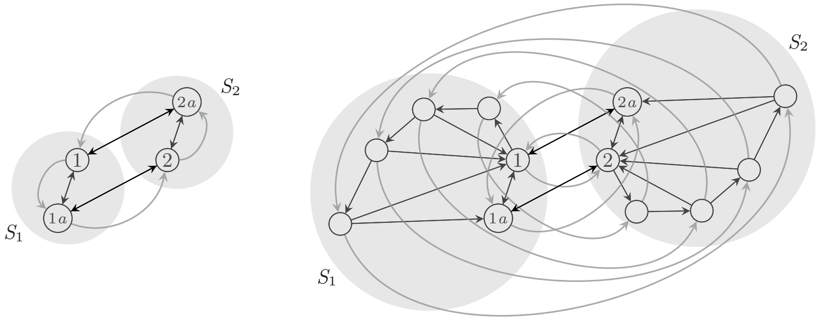


Fig. 8. A network with an automorphism disregarded by G-symmetry, and the extension given in Lemma 3 invalidating automorphisms of this kind shown with the only remaining automorphism

Proof. The idea is to add an "identifying structure" to all elements of V, which forces all automorphisms to keep V invariant and map the S_v to each other correspondingly (see Figure 8). Formally, let $K = |V'|$ and, denoting the inserted vertices by $i_{\cdot,\cdot}$, for each $v \in V$ let $I_v := \bigcup_{k=1}^{K} \{i_{v,k}\}$ and

$$E_v := \{(v, i_{v,1})\} \cup \bigcup_{k=1}^{K-1} \{(i_{v,k}, i_{v,k+1}), (i_{v,k+1}, v)\} \cup \bigcup_{w \in S_v} \{(i_{v,K}, w)\} \ ,$$

and let $H := \left(V' \cup \bigcup_{v \in V} I_v, E' \cup \bigcup_{v \in V} E_v\right)$. Now we can prove the two claims.

(i) Let $\sigma \in \Sigma_{G'}^{wb} \setminus \{\mathrm{id}\}$ with $\sigma(V) = V$ and $\sigma(S_v) = S_{\sigma(v)}$ for all $v \in V$ (such a σ exists by Lemma 2), then $\sigma \cup \bigcup_{v \in V} \bigcup_{k=1}^{K} \{i_{v,k} \mapsto i_{\sigma(v),k}\} \in \Sigma_H^{wb} \setminus \{\mathrm{id}\}$.

(ii) H is still a symmetry-preserving extension of G via (straightforward) extensions of the S_v. The discriminating construction has the effect that Σ_H consists only of extensions, as above, of those $\sigma \in \Sigma_{G'}$ for which $\sigma(V) = V$ and $\sigma(S_v) = S_{\sigma(v)}$ for all $v \in V$. Thus, any G-symmetric system with communication graph H is a symmetric system with communication graph H. Additionally, the set of all $i_{v,k}$ is invariant under Σ_H due to the distinctive structure of the I_v, thus the associated processes are allowed to differ from those of the remaining vertices. A symmetric electoral system in CSP_{in} can thus be obtained by running the original G-symmetric electoral system on all members of G' and having each $v \in V$ inform $i_{v,1}$ about the leader, while all $i_{v,k}$ simply wait for and transmit the leader information. □

Theorem 4. *There is no symmetry-preserving extension of any peer-to-peer network $G = (V, E)$ that admits a G-symmetric system pairwise synchronizing V in CSP_{in}.*

Proof. Assume there is such a symmetry-preserving extension G'. Then by Lemma 1 it also admits a G-symmetric electoral system in CSP_{in}. According to Lemma 3, there is then a network H with $\Sigma_H^{wb} \setminus \{\mathrm{id}\} \neq \emptyset$ that admits a symmetric electoral system in CSP_{in}. This is a contradiction to Theorem 1. □

5 Conclusions

We have provided a formal definition of peer-to-peer networks and adapted a semantic notion of symmetry for process systems communicating via such networks. In this context, we have defined and investigated the existence of pairwise synchronizing systems, which are directly useful because they achieve synchronization, but also because they create common knowledge between processes. Focusing on two dialects of the *CSP* calculus, we have proved the existence of such systems in $CSP_{i/o}$, as well as the impossibility of implementing them in CSP_{in}, even allowing additional helper processes like buffers. We have also mentioned a recent extension to JCSP to show that, while CSP_{in} is less complex and most commonly implemented, implementations of $CSP_{i/o}$ are feasible and do exist.

A way to circumvent our impossibility result is to remove some requirements. For example, we have sketched a construction for non-symmetric systems in CSP_{in}. In general, if we give up the symmetry requirement, $CSP_{i/o}$ can be implemented in CSP_{in} [2, p. 197].

Another way is to weaken the notion of common knowledge or approximate it [8], which may suffice in settings where the impact decreases significantly as the depth of mutual knowledge increases, see e.g. [23].

However, if one is interested in symmetric systems and exact common knowledge, as in the game-theoretical settings described in Section 1.1, then our results show that $CSP_{i/o}$ is a suitable formalism, while CSP_{in} is insufficient. Already in the introducing paper [1], the exclusion of output guards from *CSP* was recognized as reducing expressivity and being programmatically inconvenient, and soon it was deemed technically not justified [17,24] and removed in later versions of *CSP* [13, p. 227].

Some existing proposals for implementations of input and output guards and synchronous communication could be criticized for simply shifting the problems to a lower level, notably for not being symmetric themselves or for not even being strictly synchronous in real systems due to temporal imprecision [8].

However, it is often useful to abstract away from implementation issues on the high level of a process calculus or a programming language (see e.g. [25, Section 10]). For these reasons, we view our setting as an argument for implementing $CSP_{i/o}$ rather than CSP_{in}.

Acknowledgments

I would like to thank my supervisor Krzysztof Apt for his support, helpful comments and suggestions, as well as four anonymous referees for their feedback which helped to improve the paper.

This research was supported by a GLoRiClass fellowship funded by the European Commission (Early Stage Research Training Mono-Host Fellowship MEST-CT-2005-020841).

References

1. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM 21, 666–677 (1978)
2. Bougé, L.: On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes. Acta Informatica 25, 179–201 (1988)
3. Halpern, J.Y.: A computer scientist looks at game theory. Games and Economic Behavior 45, 114–131 (2003)
4. Gray, J.: Notes on Data Base Operating Systems. LNCS, vol. 60, pp. 393–481. Springer, Heidelberg (1978)
5. Rubinstein, A.: The electronic mail game: Strategic behavior under almost common knowledge. The American Economic Review 79, 385–391 (1989)
6. Morris, S.: Coordination, communication, and common knowledge: A retrospective on the electronic-mail game. Oxf Rev Econ Policy 18, 433–445 (2002)
7. Fagin, R., Halpern, J.Y., Vardi, M.Y., Moses, Y.: Reasoning about knowledge. MIT Press, Cambridge (1995)
8. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. Journal of the ACM 37, 549–587 (1990)
9. Schneider, F.B.: Synchronization in distributed programs. ACM Trans. Program. Lang. Syst. 4, 125–148 (1982)
10. Osborne, M.J.: An Introduction to Game Theory. Oxford University Press, New York (2003)
11. Moulin, H.: Axioms of Cooperative Decision Making. Cambridge University Press, Cambridge (1988)
12. Andrews, G.R.: Concurrent Programming: Principles and Practice. Addison-Wesley, Reading (1991)
13. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
14. Schneider, S.: Concurrent and Real Time Systems: The CSP Approach. John Wiley and Sons, Chichester (1999)
15. INMOS Ltd. occam 2 Reference Manual. Prentice-Hall (1988)
16. Fich, F., Ruppert, E.: Hundreds of impossibility results for distributed computing. Distributed Computing 16, 121–163 (2003)

17. Buckley, G.N., Silberschatz, A.: An effective implementation for the generalized input-output construct of csp. ACM Trans. Program. Lang. Syst. 5, 223–235 (1983)
18. Welch, P.: An occam-pi Quick Reference (1996–2007), `https://www.cs.kent.ac.uk/research/groups/sys/wiki/OccamPiReference`
19. Welch, P., Brown, N., Moores, J., Chalmers, K., Sputh, B.: Integrating and extending JCSP. In: McEwan, A.A., Schneider, S., Ifill, W., Welch, P. (eds.) Communicating Process Architectures. IOS Press, Amsterdam (2007)
20. Jones, G.: On guards. In: Muntean, T. (ed.) Parallel Programming of Transputer Based Machines (OUG-7), pp. 15–24. IOS Press, Amsterdam (1988)
21. Palamidessi, C.: Comparing the expressive power of the synchronous and asynchronous pi-calculi. Mathematical Structures in Computer Science 13, 685–719 (2003)
22. Parikh, R., Krasucki, P.: Communication, consensus, and knowledge. Journal of Economic Theory 52, 178–189 (1990)
23. Weinstein, J., Yildiz, M.: Impact of higher-order uncertainty. Games and Economic Behavior 60, 200–212 (2007)
24. Bernstein, A.: Output guards and nondeterminism in Communicating Sequential Processes. ACM Trans. Program. Lang. Syst. 2, 234–238 (1980)
25. Kurki-Suonio, R.: Towards programming with knowledge expressions. In: 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL), pp. 140–149. ACM Press, St. Petersburg Beach (1986)

Author Index