Claude Bolduc
Jules Desharnais
Béchir Ktari (Eds.)

LNCS 6120

Mathematics of Program Construction

10th International Conference, MPC 2010
Québec City, Canada, June 2010
Proceedings

Springer

Claude Bolduc Jules Desharnais
Béchir Ktari (Eds.)

Mathematics of Program Construction

10th International Conference, MPC 2010
Québec City, Canada, June 21-23, 2010
Proceedings

Springer

Volume Editors

Claude Bolduc
Jules Desharnais
Béchir Ktari

Université Laval, Département d'informatique et de génie logiciel
Pavillon Adrien-Pouliot, 1065 Avenue de la Médecine
Québec, QC, G1V 0A6, Canada
E-mail: {Claude.Bolduc, Jules.Desharnais, Bechir.Ktari}@ift.ulaval.ca

Library of Congress Control Number: 2010927075

CR Subject Classification (1998): F.3, D.2, F.4.1, D.3, D.2.4, D.1

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

ISSN	0302-9743
ISBN-10	3-642-13320-7 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-13320-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

This volume contains the proceedings of MPC 2010, the 10th International Conference on the Mathematics of Program Construction. The biennial MPC conferences aim to promote the development of mathematical principles and techniques that are demonstrably practical and effective 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, program analysis, programming paradigms, programming calculi, programming language semantics, security, and program logics.

The conference took place during June 21–23 in Lac-Beauport, a suburb of Québec City, Canada, prior to AMAST 2010 (June 23–26). The previous nine conferences were held in 1989 in Twente, The Netherlands (LNCS 375); in 1992 in Oxford, UK (LNCS 669); in 1995 in Kloster Irsee, Germany (LNCS 947); in 1998 in Marstrand near Göteborg, Sweden (LNCS 1422); in 2000 in Ponte de Lima, Portugal (LNCS 1837); in 2002 in Dagstuhl, Germany (LNCS 2386); in 2004, in Stirling, UK (LNCS 3125); in 2006 in Kuressaare, Estonia (LNCS 4014); and in 2008 in Marseille-Luminy, France (LNCS 5133).

The volume contains one invited paper, the abstracts of two invited talks, and 19 papers selected for presentation by the Program Committee from 37 submissions. Each paper was refereed by at least three reviewers, and on average by four. We are grateful to the members of the Program Committee and the external 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 highly recommend to every Program Chair.

June 2010

Claude Bolduc
Jules Desharnais
Béchir Ktari

Conference Organization

Program Chair

Jules Desharnais	Université Laval, Canada

Program Committee

Philippe Audebaud	École Normale Supérieure de Lyon, France
Ralph-Johan Back	Åbo Akademi University, Finland
Eerke Boiten	University of Kent, UK
Sharon Curtis	Oxford Brookes University, UK
Jeremy Gibbons	University of Oxford, UK
Lindsay Groves	Victoria University of Wellington, New Zealand
Ian J. Hayes	University of Queensland, Australia
Eric Hehner	University of Toronto, Canada
Zhenjiang Hu	National Institute of Informatics, Japan
Johan Jeuring	Utrecht University, The Netherlands
Christian Lengauer	Universität Passau, Germany
Bernhard Möller	Universität Augsburg, Germany
Shin-Cheng Mu	Academia Sinica, Taiwan
David Naumann	Stevens Institute of Technology, USA
José Nuno Oliveira	Universidade do Minho, Portugal
Alberto Pardo	Universidad de la República, Uruguay
Christine Paulin-Mohring	INRIA-Université Paris-Sud, France
Steve Reeves	University of Waikato, New Zealand
Tim Sheard	Portland State University, USA
Georg Struth	Sheffield University, UK
Tarmo Uustalu	Institute of Cybernetics, Estonia

External Reviewers

José Bacelar Almeida
Luís Barbosa
Bruno Barras
Daniel Calegari
Liang-Ting Chen
Olaf Chitil
Robert Colvin
Alcino Cunha
Han-Hing Dang
Simon Doherty
Facundo Domínguez
Brijesh Dongol
Steve Dunne
Jean-Christophe Filliâtre
Roland Glück
Sergei Gorlatch
Makoto Hamana
Christoph Herrmann
Richard Jones
Hiroyuki Kato
Heiko Mantel
Clare Martin
Kazutaka Matsuda
Conor McBride

Local Organizers

Claude Bolduc, Jules Desharnais, and Béchir Ktari (Université Laval, Canada)

Sponsoring Institutions

- Université Laval, Québec, Canada, http://www.ulaval.ca
- Centre de recherches mathématiques, Université de Montréal, Montréal, Canada, http://www.crm.umontreal.ca

Table of Contents

The Algorithmics of Solitaire-Like Games

Roland Backhouse, Wei Chen, and João F. Ferreira[⋆]

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

Abstract. Puzzles and games have been used for centuries to nurture problem-solving skills. Although often presented as isolated brain-teasers, the desire to know how to win makes games ideal examples for teaching algorithmic problem solving. With this in mind, this paper explores one-person solitaire-like games.

The key to understanding solutions to solitaire-like games is the identification of invariant properties of polynomial arithmetic. We demonstrate this via three case studies: solitaire itself, tiling problems and a collection of novel one-person games. The known classification of states of the game of (peg) solitaire into 16 equivalence classes is used to introduce the relevance of polynomial arithmetic. Then we give a novel algebraic formulation of the solution to a class of tiling problems. Finally, we introduce an infinite class of challenging one-person games inspired by earlier work by Chen and Backhouse on the relation between cyclotomic polynomials and generalisations of the seven-trees-in-one type isomorphism. We show how to derive algorithms to solve these games.

Keywords: Solitaire, invariants, tiling problems, polynomials, games on cyclotomic polynomials, seven-trees-in-one, nuclear pennies, algorithm derivation.

1 Introduction

There are two concepts that are basic to all algorithms that process input values using some sort of iterative scheme: invariants and making progress. Although in principle making progress can involve quite complicated theories on well-founded relations, in practice the concept is easy for students to grasp. On the other hand, students are often given misleading information about invariants; they are often taught that invariants are (only) needed for *post-hoc* verification of program correctness and very difficult to formulate. In reality, a good understanding of invariants is crucial to successful algorithm design.

For the last seven years, the module Algorithmic Problem Solving has been taught to first-year Computer Science students at the University of Nottingham.

[⋆] Funded by Fundação para a Ciência e a Tecnologia (Portugal) under grant SFRH/BD/24269/2005.

The module aims to introduce students to effective problem-solving techniques, particularly in the context of solving problems that demand an algorithmic solution; the first technique that is presented is the use of invariants. At a later stage, two-person games are studied in some depth; games are inherently algorithmic in nature (after all, the goal is to win, which means formulating some sort of algorithm) and require little motivation.

There are many puzzles and games in the mathematical literature which have been used for centuries to nurture problem-solving ability. Many are presented as isolated brain-teasers but, for our pedagogical purposes, it is important that they have two qualities. First, any problem that is studied must have a substantial number of variations which can be used to test students' understanding and, second, the solution method should demonstrate effective algorithmic problem-solving rather than being ad hoc or magic.

Recently, we have been studying one-person solitaire-like games in order to try to extract useful examples for study in the module. In this paper, we present our findings so far, including an infinite class of challenging games that we have invented based on insights from type theory.

We begin the paper in section 2 with a brief summary of well-known properties of the game of solitaire. These properties are derived using the algebra of polynomials in a suitably chosen semiring; it is this algebra that is the basis for the novel applications that we discuss in later sections.

Section 3 is about a class of tiling problems. In section 3.1, we show how Golomb's [1] use of colours to solve one such problem is formulated algebraically. (Our solution is simpler than the algebraic formulation proposed by Mackinnon [2].) The solution to the class of tiling problems is discussed in section 3.2.

The so-called "nuclear pennies" game [3] is an example of a game which, until now, has been of isolated interest. The game is based on the theorem attributed to Lawvere that "seven trees are one". That is, if T is the type of binary trees, the type T^7 (the Cartesian product of T with itself 7 times) is isomorphic to T. The game involves moving a checker 6 places to the right on a one-dimensional tape (from position 1 to position 7) following rules that reflect the recursive definition of binary trees. In section 4, we formulate an infinite collection of games, each with different rules, where the goal is to move a checker a certain number of positions from its starting position on a one-dimensional tape. So far as we are aware, these games and their solution are original to this paper. The games were derived from our study of the problem: given a number n, invent an interesting type T such that T^n is isomorphic to T [4].

2 Solitaire and Variations

Solitaire is a well-known game. The game begins with a number of pegs stuck in holes in a board. The holes are arranged in a grid, the shape of which is not relevant to the current discussion. A move, shown diagrammatically in fig. 1, replaces two pegs by one and the game is to remove all pegs bar one, leaving the peg in a designated position. In this section, we show how invariants of polynomial arithmetic are used in the analysis of moves in the game of solitaire and in a variation on solitaire called the solitaire army game.

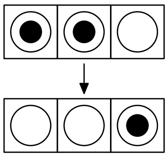


Fig. 1. Move from left to right. Similar moves are allowed from right to left, from top to bottom, and from bottom to top.

2.1 Solitaire

De Bruijn [5] shows that states in the game of solitaire can be divided into 16 equivalence classes in such a way that all moves are between equivalent states. (So the equivalence class of the state is an invariant of each move.) Here is a brief reformulation of De Bruijn's argument[1].

Suppose we assign non-negative integer coordinates (i,j) to each hole in the board. Suppose $R=(A,\mathbf{0},\mathbf{1},+,\cdot)$ is a semiring[2] and suppose p is an element of A. Assign to a peg at position (i,j) the *weight* p^{i+j}. The *total weight* of a state in the game is the sum of the weights of all the pegs on the board in that state. There are four types of move in the game — vertically up and down, and horizontally left and right. A vertical-up move replaces pegs with weights p^{i+j+0} and p^{i+j+1} by a peg with weight p^{i+j+2}. So, if p has the property that $p^0+p^1=p^2$, the total weight is invariant. Similarly, a horizontal-left move replaces pegs with weights p^{i+2+j} and p^{i+1+j} by a peg with weight p^{i+0+j}. So, if $p^2+p^1=p^0$, the total weight remains invariant. A similar analysis applies to the two other types of moves: if $p^2+p^1=p^0$, the total weight is invariant under vertical-down moves and, if $p^0+p^1=p^2$, the total weight is invariant under horizontal-right moves.

The carrier set of the field[3] $GF(4)$ has exactly 4 elements which can be named $\mathbf{0}$, $\mathbf{1}$, p and p^2. Moreover, these elements have the property that $\mathbf{1}+\mathbf{1}=\mathbf{0}$, $\mathbf{1}+p=p^2$ and (hence) $p^2+p=\mathbf{1}$. Thus, if $GF(4)$ is used to compute the weights of states, the weight is invariant under all moves. This divides the states into four equivalence classes, and the initial and final states in the game must be in the same equivalence class.

[1] Berlekamp *et al* [6, pp. 708–710] attribute the theorem to M. Reiss [7] but the argument is different. De Bruijn's argument is more relevant to later sections of this paper. We have not read Reiss's paper but assume that Berlekamp *et al* copy his presentation.

[2] That is, addition in R is associative and commutative and has unit $\mathbf{0}$, product is associative and has unit $\mathbf{1}$ and zero $\mathbf{0}$, and product distributes over addition.

[3] The fact that $GF(4)$ is a field and not just a semiring is not relevant to the argument.

To complete the argument, a symmetrical weighting is used: assign to a peg at position (i,j) the weight p^{i-j}. The total weight is again the sum of the weights of all the pegs on the board in that state. (We call it a "symmetrical" weighting because it is equivalent to turning the board through 90^0.) The same analysis applies, classifying each state into one of 4 equivalence classes.

Combining the two[4], the states are divided into $4{\times}4$ equivalence classes in such a way that the equivalence class is invariant under moves. The game can be solved only if the initial and final states are in the same equivalence class.

2.2 The Solitaire Army

De Bruijn's weighting of a state in a game does not provide a sufficient condition for when it is possible to move from a given initial state to a given final state. Berlekamp *et al* [6, chap. 23] discuss in detail a number of problems, when they can be solved and when they cannot be solved. A tool in their analysis is the notion of a *pagoda function*, which computing scientists would recognise as being similar to a measure of progress. Specifically, a pagoda function is any real-valued function pag on peg-positions that has the property that if a move replaces pegs at positions r and s by a peg at position t then

$$pag.t \le pag.s + pag.r$$

A much-celebrated problem —discussed in several other books and Internet pages— which they solve using a pagoda function is the "solitaire army" problem. This is how they describe the problem.

> A number of Solitaire men stand initially on one side of a straight line beyond which is an infinite empty desert. How many men do we need to send a scout just 0, 1, 2, 3, 4 or 5 paces out into the desert?

The surprising fact is that for 5 paces (or more) there is no solution! The proof (attributed in Wikipedia to John Horton Conway, 1961) resembles De Bruijn's analysis. Suppose peg positions are assigned Cartesian coordinates so that the goal position is given the coordinates $(0,0)$ and the initial positions of the Solitaire men have negative coordinates. Suppose we assign to a peg at position (i,j) the weight $\sigma^{|i+j|}$, where σ is yet to be chosen. Note that $|i{+}j|$ is the distance of (i,j) from $(0,0)$ along a shortest path taken by a peg in moves of the game. The weight of any state in the game is the sum of the weights of all the pegs on the board in that state. Now σ is chosen so that the weighting of pegs is a pagoda function. Specifically, choose $\sigma = \frac{1}{2}(\sqrt{5}-1)$ so that $\sigma^2{+}\sigma = 1$. (This guarantees that the weighting of pegs remains constant when a peg is moved along a shortest path to $(0,0)$ and decreases for other moves.) Then the maximum weight of an initial state in which a finite number

[4] As pointed out to us by Diethard Michaelis, the combination of weights is an element of the semiring $GF(4){\times}GF(4)$ where addition and product are defined componentwise. Note that $GF(4){\times}GF(4)$ is not a field because it has divisors of zero.

of Solitaire men is at a distance at least n from $(0,0)$ is less than σ^{n-5} . For the 5-pace problem, the goal is to reach a state with weight at least σ^{5-5} (i.e. 1) but this is impossible because the weighting is a pagoda function — its value is never increased by a move.

Edsger W. Dijsktra [8] discusses a similar problem (and gives a similar solution).

3 Tiling Problems

Tiling problems involve covering a board without overlapping with a given collection of tiles. Traditionally their solution involves (seemingly ad hoc) colouring arguments. This section is essentially about how to formulate the colouring arguments algebraically.

3.1 The Chessboard Problem

Consider the problem of tiling a chessboard with twenty-one 3×1 rectangles and one 1×1 square. Index each square of the chessboard by a pair of natural numbers (i,j) in the obvious way. For concreteness, we assume that the bottom-left corner is given the label $(0,0)$.

Suppose a chessboard is partially tiled by 3×1 rectangles. As in De Bruijn's analysis of solitaire, give to the square (i,j) two "weights": the *forward* weight is p^{i-j} and the *backward* weight is p^{i+j} , where p is a generator of the field $GF(4)$. Two weights are assigned to the chessboard as follows: the *forward weight* of the chessboard is the sum (in $GF(4)$) of the forward weights of all the individual squares that are tiled, and the *backward weight* of the chessboard is the sum of the backward weights of all the individual squares that are tiled.

Recall that the elements of $GF(4)$ are $\mathbf{0}$, $\mathbf{1}$, p and p^2 and that $\mathbf{1}{+}\mathbf{1}=\mathbf{0}$ and $\mathbf{1}{+}p=p^2$. It follows that

$$(1) \qquad \mathbf{0}=\mathbf{1}{+}p{+}p^2 \quad .$$

In particular, $p^3=\mathbf{1}$, the forward weight of square (i,j) is $p^{(i-j)\,\mathsf{mod}\,3}$ and its backward weight is $p^{(i+j)\,\mathsf{mod}\,3}$. Thus the weights are identical on forward and backward diagonals of the board, respectively. (This is the explanation for our choice of nomenclature).

It is easily checked that when a 3×1 tile is placed on a chessboard, both the forward and backward weights of the chessboard do not change; they are *invariants* of the tiling process. (For example, if a 3×1 tile is placed horizontally on the board with leftmost square at position (i,j) , the weight $p^{i+j}\times(\mathbf{1}{+}p{+}p^2)$ is added to the weight of the board. Because $\mathbf{1}{+}p{+}p^2$ equals $\mathbf{0}$, adding or subtracting this weight has no effect on the total weight.) This is the basis for the choice of $GF(4)$ in weighing squares: it is the simplest possible semiring that satisfies (1) in a non-trivial way.

The forward and backward weights of a completely tiled chessboard are 1 and p, respectively. In order to tile the chessboard completely with twenty-one 3×1 rectangles and one 1×1 square, the 1×1 square must therefore be placed on a square with forward weight 1 and backward weight p. The former are the squares (i,j) with $(i-j \equiv 0) \bmod 3$ and the latter are the squares (i,j) with $(i+j \equiv 1) \bmod 3$. Combined with the requirement that i and j are natural numbers each of which is at most 7, there are just 4 solutions to this pair of equations, namely $(i,j)=(2,2)$, $(i,j)=(5,5)$, $(i,j)=(2,5)$, and $(i,j)=(5,2)$.

The above argument is a simpler presentation of an "algebraic" proof given by Mackinnon [2]. (Our formulation is simpler because Mackinnon takes for p a complex solution of equation (1) — the field of complex numbers is, of course, much more complicated than $GF(4)$.) If colours —say red, white and blue— are assigned to the non-zero elements of $GF(4)$, the argument is essentially the same as Golomb's [1] "colouring" proof. Specifically, Golomb's proof has two components, the colouring of the squares and rotational symmetry of the board. The colouring of the squares is just the assignment of three different values to the squares; this is chosen so that the net "count" of colours on the board —whereby three differently coloured squares "count" as zero— is one. The rotational symmetry is expressed algebraically by the two weights given to squares of the board. The colouring and algebraic proofs are thus in essence identical.

3.2 The Generalisation

In order to demonstrate the effectiveness of the algebraic formulation, let us consider a generalisation. Suppose we have an $m\times m$ board, an unlimited supply of n-ominoes and one 1-omino. (An n-omino is an $n\times 1$ board, i.e. a strip of n squares each of which is the same size as a square of the given $m\times m$ board.) In order to eliminate the trivial case, we assume that $1<m$. We prove that it is possible to cover the $m\times m$ board with the supplied n-ominoes and one 1-omino, without overlapping, precisely when

$$(2) \qquad 1\leq n<m \;\wedge\; (n\backslash(m-1) \vee n\backslash(m+1)) \quad .$$

An obvious necessary condition is

$$1\leq n<m \;\wedge\; n\backslash(m^2-1) \quad .$$

This, however, is not equivalent. For example, it is satisfied by $m=11$ and $n=8$ but it is not the case that $8\backslash(11-1) \vee 8\backslash(11+1)$.

Invariant. Arbitrary Semiring. First, we show that (2) is necessary. Consider any semiring $R=(A,\mathbf{0},\mathbf{1},+,\cdot)$ with an element $x\in A$ that has the property that

$$(3) \qquad \langle \Sigma i : 0\leq i<n : x^i \rangle \;=_R\; \mathbf{0} \quad .$$

(We give examples of such semirings later. The subscript on the equality symbol is necessary later to avoid the confusion that can be caused by overloading.) Now let us assign to each square (i,j) the *weight* x^{i+j} if it is covered and the weight 0 if it is not covered. The weight of a (partially) tiled board is defined to be the sum of the weights of the tiled squares.

On account of (3) above, the placement of an n-omino on the board does not change the weight of the board. Since there is exactly one 1-omino on a completely covered board, a necessary condition is that

$$\left\langle \exists k :: \left\langle \Sigma\, i,j : 0 \leq i < m \wedge 0 \leq j < m : x^{i+j} \right\rangle =_R x^k \right\rangle \quad .$$

Equivalently (calculation left to the reader),

$$(4) \qquad \left\langle \exists k :: \left\langle \Sigma i : 0 \leq i < m : x^i \right\rangle^2 =_R x^k \right\rangle \quad .$$

We show that (4) implies $n \backslash (m-1) \vee n \backslash (m+1)$. Our calculations exploit the following immediate consequences of (3): for all j,

$$(5) \qquad x^j =_R x^{j \,\mathsf{mod}\, n} \quad ,$$

and, hence, for all j,

$$(6) \qquad \left\langle \Sigma i : 0 \leq i < j : x^i \right\rangle =_R \left\langle \Sigma i : 0 \leq i < j \,\mathsf{mod}\, n : x^i \right\rangle \quad .$$

Invariant. Polynomials over $\boldsymbol{GF}(\mathbf{2})$. To complete our argument, we fix the semiring R to be

$$GF(2)[x] \;/\; \left\langle \Sigma i : 0 \leq i < n : x^i \right\rangle$$

That is, R is the set of polynomials in the indeterminate x with coefficients in $GF(2)$ (which is conventionally denoted by $GF(2)[x]$) modulo the polynomial $\left\langle \Sigma i : 0 \leq i < n : x^i \right\rangle$. Thus, in R we have the property (3).

This choice of R is motivated by our goal. Note first the squaring in (4); $GF(2)$ is the simplest example of a semiring in which squaring distributes through addition. This property is easily seen to be inherited by $GF(2)[x]$. That is, for all j,

$$(7) \qquad \left\langle \Sigma i : 0 \leq i < j : x^i \right\rangle^2 =_{GF(2)[x]} \left\langle \Sigma i : 0 \leq i < j : x^{2i} \right\rangle \quad .$$

Hence, the equality also holds in R. Also, the semiring R has 2^{n-1} distinct elements since each element in R has two representations as a polynomial in $GF(2)[x]$ with degree less than n, and there are 2^n such polynomials. (For example, $\mathbf{0}$ is represented by the two polynomials $\left\langle \Sigma i : 0 \leq i < n : 0 \times x^i \right\rangle$ and $\left\langle \Sigma i : 0 \leq i < n : 1 \times x^i \right\rangle$.) In particular (cf (4))

$$x^k =_R \left\langle \Sigma i : 0 \leq i < n \wedge i \neq k : x^i \right\rangle \quad .$$

Consequently, if the function $\#$ of type $GF(2)[x] \rightarrow \mathbb{N}$ counts the number of non-zero coefficients in a given polynomial, then for all k and all P in $GF(2)[x]$ with degree less than n,

$$(8) \qquad (P =_R x^k) \;\Rightarrow\; (\#P = 1) \vee (\#P = n-1) \quad .$$

(It is at this point that the subscript R on the equality sign becomes essential; the left and right side of the equation denote P and x^k, respectively, after injection into the semiring $GF(2)[x]$ modulo the polynomial $\left\langle \Sigma i : 0 \leq i < n : x^i \right\rangle$).

We now have:

$$
\begin{array}{ll}
 & (4) \\
= & \{ \quad (6) \text{ and } (7) \quad \} \\
 & \langle \exists k :: \langle \Sigma i : 0 \le i < m \,\mathsf{mod}\, n : x^{2i} \rangle =_R x^k \rangle \quad .
\end{array}
$$

In order to apply (8), we conduct a case analysis on $m \,\mathsf{mod}\, n$ and on n. There are three cases to consider:

(a) $\quad 2(m \,\mathsf{mod}\, n) < n \quad .$
This is the easiest case. The degree of $\langle \Sigma i : 0 \le i < m \,\mathsf{mod}\, n : x^{2i} \rangle$ is less than n so, applying (8), we have:

$$(4) \quad \Rightarrow \quad (m \,\mathsf{mod}\, n = 1) \vee (m \,\mathsf{mod}\, n = n{-}1) \quad .$$

(b) $\quad 2{\times}(m \,\mathsf{mod}\, n) \ge n \;\wedge\; \mathsf{even}.n \quad .$
Suppose $n = 2q$. The goal is to reduce $\langle \Sigma i : 0 \le i < m \,\mathsf{mod}\, n : x^{2i} \rangle$ to a polynomial with degree less than n. This is done in the following calculation. (Equalities are in R, i.e. in $GF(2)[x] \;/\; \langle \Sigma i : 0 \le i < n : x^i \rangle$.)

$$
\begin{array}{ll}
 & \langle \Sigma i : 0 \le i < m \,\mathsf{mod}\, n : x^{2i} \rangle \\
=_R & \{ \quad \text{assumption, } 2(m \,\mathsf{mod}\, n) \ge n \quad \} \\
 & \langle \Sigma i : 0 \le i < q : x^{2i} \rangle \;+\; \langle \Sigma i : q \le i < m \,\mathsf{mod}\, n : x^{2i} \rangle \\
=_R & \{ \quad (5),\; n = 2q \quad \} \\
 & \langle \Sigma i : 0 \le i < q : x^{2i} \rangle \;+\; \langle \Sigma i : q \le i < m \,\mathsf{mod}\, n : x^{2(i-q)} \rangle \\
=_R & \{ \quad \text{quantifier calculus, in } GF(2)\,,\; [\, a{+}a = 0 \,]\,, \\
 & \quad\;\; m \,\mathsf{mod}\, n \;<\; n \;=\; 2q \quad \} \\
 & \langle \Sigma i \;:\; m \,\mathsf{mod}\, n - q \;\le\; i \;<\; q \;:\; x^{2i} \rangle \quad .
\end{array}
$$

The last line above is a polynomial in $GF(2)$ with degree less than n. Hence, applying (8) to it, we have:

$$(4) \quad \Rightarrow \quad (2q - m \,\mathsf{mod}\, n = 1) \vee (2q - m \,\mathsf{mod}\, n = n{-}1) \quad .$$

Simplifying, using $n = 2q$ (and symmetry of disjunction),

$$(4) \quad \Rightarrow \quad (m \,\mathsf{mod}\, n = 1) \vee (m \,\mathsf{mod}\, n = n{-}1) \quad .$$

(c) $\quad 2{\times}(m \,\mathsf{mod}\, n) \ge n \;\wedge\; \mathsf{odd}.n \quad .$
Suppose $n = 2q - 1$. Then, by a similar calculation, we have:

$$
\begin{array}{ll}
 & \langle \Sigma i : 0 \le i < m \,\mathsf{mod}\, n : x^{2i} \rangle \\
=_R & \{ \quad \text{assumption, } 2(m \,\mathsf{mod}\, n) \ge n\,, \\
 & \quad\;\; (5),\; n = 2q - 1 \quad \} \\
 & \langle \Sigma i : 0 \le i < q : x^{2i} \rangle \;+\; \langle \Sigma i : q \le i < m \,\mathsf{mod}\, n : x^{2(i-q)+1} \rangle \\
=_R & \{ \quad \text{quantifier calculus} \quad \} \\
 & \langle \Sigma i : 0 \le i < m \,\mathsf{mod}\, n : x^{i} \rangle \quad .
\end{array}
$$

The last line above is a polynomial in $GF(2)$ with degree less than n. Hence, applying (8) to it, we again get:

$$(4) \quad \Rightarrow \quad (m \,\mathsf{mod}\, n = 1) \vee (m \,\mathsf{mod}\, n = n{-}1) \quad .$$

We conclude that, in all cases, (4) implies that

$$(m \,\mathsf{mod}\, n \;=\; 1) \;\vee\; (m \,\mathsf{mod}\, n \;=\; n{-}1) \quad .$$

Equivalently,

$$n\backslash(m{-}1) \;\vee\; n\backslash(m{+}1) \quad .$$

Figs. 2(a) and 2(b) show that this condition is also sufficient.

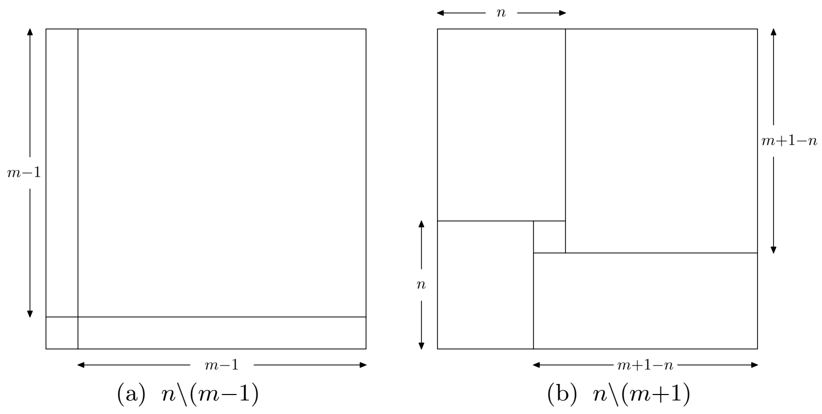


Fig. 2. $n\backslash(m{-}1) \vee n\backslash(m{+}1)$ is sufficient

4 Games on Cyclotomic Polynomials

In this section, we solve a novel class of games played on a one-dimensional tape. The general class is considered in section 4.2; the so-called "nuclear pennies game" [9] based on the "seven trees in one" property [10,11] is used in section 4.1 to introduce the solution method.

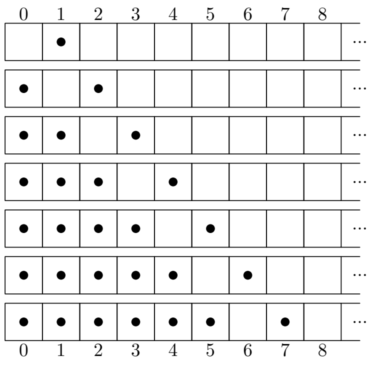
4.1 Seven-Trees-in-One and the Nuclear Pennies Game

Consider the definition of binary trees — a binary tree is an empty tree or an element together with a pair of binary trees. Let us use symbols $+$ and $\times$ to denote disjoint union and Cartesian product respectively and let $\mathbb{1}$ denote the unit type. The type T of binary trees can be characterised by the type isomorphism $T \cong \mathbb{1}{+}T{\times}T$. Surprisingly, it can be shown that there is an isomorphism between seven-tuples of binary trees and binary trees. That is, $T^7 \cong T$. This has been dubbed "seven trees in one" by Blass [10] who attributes the isomorphism to a remark made by Lawvere [12].

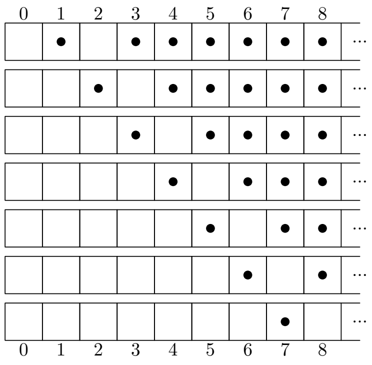
The isomorphism has been turned into a game with checkers called the "nuclear pennies game" [3]. The game is played on a one-dimensional board of infinite extent. A checker is placed on one of the squares and the goal is to move the checker six places to the right. An atomic move is to replace a checker in a square numbered $n{+}1$ by two checkers, one in each of the two adjacent squares n and $n{+}2$, or vice-versa, two checkers, one in square n and one in square $n{+}2$ for some n, are replaced by a checker in square $n{+}1$. The connection with seven-trees-in-one is easy to see if one views a move as replacing $T^n{\times}T$ by $T^n{\times}(\mathbb{1}{+}T{\times}T)$ or vice-versa.

The nuclear-pennies game has an easy solution if one exploits the left-right symmetry of the problem (moving a coin 6 places to the right is the same as moving a coin 6 places to the left). The problem is decomposed into first ensuring that there is a checker in the square 6 places to the right of the starting position and, symmetrically, there is a checker in the square 6 places to the left of the finishing position.

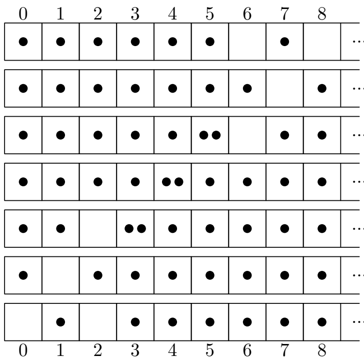
Achieving this first stage is easy. Below we show how it is done. First, six moves are needed to ensure that a checker is added six places to the right of the starting position. (This is shown below using dots to indicate checkers on a square. A blank indicates no checker on the square).



Symmetrically, working from bottom to top, six moves are needed to ensure that a checker is added six places to the left of the finishing position.



Now the goal is to connect these two intermediate states (the bottom state in the top diagram and the top state in the bottom diagram). An appropriate (symmetrical) sequence of states is as follows. (For the reader's convenience, the last and first states in the above figures are repeated as the top and bottom states in the figure below).



The first and last moves make the number of checkers in the leftmost and rightmost positions equal. Then a small amount of creativity is needed to identify the two (symmetrical) moves to the (symmetrical) middle state.

4.2 Cyclotomic Polynomials

Although the nuclear-pennies game is an interesting exercise in the exploitation of symmetry in problem decomposition, it has until recently been an isolated example and appears to have attracted relatively little attention. (The website [9] gives one other example, dubbed the "thermonuclear pennies game".) In a recently submitted paper, Chen and Backhouse [4] posed the problem of, given an arbitrary n, is it possible to invent an "interesting" type T such that $T \cong T^{n+1}$. Likewise, given an arbitrary n, is it possible to invent an "interesting" nuclear-pennies-like game in which the task is to move a checker n places to the right using a sequence of pre-defined atomic moves. They gave an affirmative solution to the first question and a partial solution to the second, both answers being based on the use of so-called cyclotomic polynomials. (The solution to the second problem is partial in the sense that games were invented for an unbounded number of values of n but not all numbers n.) In this section, we present this class of "cyclotomic" games and their solution. (The paper [4] predicts that all cyclotomic games are solvable but does not give an explicit solution).

The games we consider in this section are all based on an equation of the form

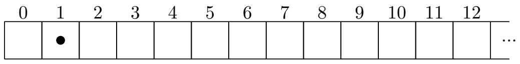
$$(9) \qquad T^1 = T^1 + \Gamma$$

where Γ is a polynomial in T with positive integer coefficients. The atomic moves in such a game are to supplement a checker at position $i+1$ by Γ_k additional checkers at positions $i+k$ where Γ_k is the coefficient of T^k in the polynomial Γ —this corresponds to the use of the equation (9) as a left-to-right replacement rule— or, vice-versa, if there is at least one checker at position $i{+}1$ and additionally Γ_k checkers at each position $i{+}k$, remove the Γ_k checkers at the positions $i{+}k$. The task is to move from an initial state in which there is just one checker at position 1 to a final state where there is just one checker at position n, for some pre-defined n.

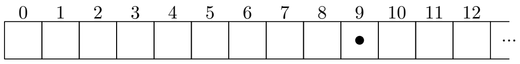
A concrete example is the game based on the equation:

$$(10) \qquad T^1 \;=\; T^1 + (T^0{+}T^4) \;\; .$$

In this game, we are given the following board with one checker on position 1:



Now, whenever there is a checker in position $i{+}1$, we are allowed to add two checkers to the board —one in position i and the other in position $i{+}4$— and, vice-versa, whenever there is at least one checker in positions i, $i{+}1$, and $i{+}4$, we are allowed to remove one checker from positions i and $i{+}4$. The question is to determine if, from the initial state shown above, it is possible to move the checker 8 places to the right, i.e., to obtain the following state:



A necessary condition for a game based on (9) to be solvable is easily determined. Suppose we represent any state of the board by a polynomial in $\mathbb{Z}[T]$. Then, the moves are so designed that the polynomial modulo Γ is an invariant. The initial state is represented by T and the desired final state is represented by T^{n+1}. A necessary condition is thus that

$$(11) \qquad T \,\mathsf{mod}\, \Gamma \;\;=\;\; T^{n+1} \,\mathsf{mod}\, \Gamma \;\; .$$

Equivalently, the polynomial $T^{n+1}{-}T$ must be divisible by Γ.

A well-known result is that, in $\mathbb{Z}[T]$, a polynomial is a divisor of $T^n{-}1$ equivales it is a product of so-called cyclotomic polynomials. (See Wikipedia or [4] for further information on cyclotomic polynomials.) For our purpose of inventing games with checkers, we restrict attention to products of cyclotomic polynomials in $\mathbb{N}[T]$. Specifically, we consider the polynomials $\psi_{a,n}$ where

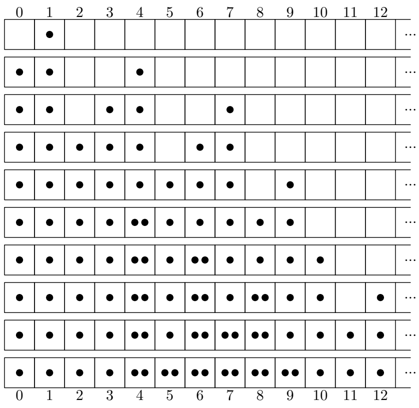
$$\psi_{a,n} \;\;=\;\; \langle \Sigma i \,{:}\, 0 \le i < a \,{:}\, T^{i\times a^{n-1}} \rangle \qquad .$$

We assume that a and n are so chosen that the degree of $\psi_{a,n}$ is at least 2. Equivalently, we assume that

$$(12) \qquad (2 \le a \wedge 2 \le n) \;\vee\; (3 \le a \wedge 1{=}n) \;\; .$$

(When a is a prime number, $\psi_{a,n}$ is a cyclotomic polynomial and is commonly denoted by Φ_{a^n}. When a is not prime, $\psi_{a,n}$ is a product of cyclotomic polynomials.) For a game based on $\psi_{a,n}$ the goal is to move a checker a^n places to the right. For example, $\psi_{2,3} = 1+T^4$; it is this polynomial that is used in the game defined by (10) with goal to move the checker 2^3 (i.e. 8) places to the right.

Before considering the general case, let us see how we could move the checker in the above example. The first trivial observation is that we have to add new checkers to the board (whenever we add checkers, we say that we perform an *expansion*). Another trivial observation is that, in order to leave a single checker in position 9 , we have to apply the rule in reverse, i.e., we have to remove checkers (whenever we remove checkers, we say that we perform a *contraction*). From these two observations, we propose constructing an algorithm that is divided into two phases: first, we perform a sequence of expansions that place at least one checker in the desired position; second, we perform a sequence of contractions until we have exactly one checker in the desired position. Figure 3 shows a solution. Note the division into two distinct phases: fig. 3(a) shows a sequence of expansions and fig. 3(b) shows a sequence of contractions. (For the reader's convenience, the middle state is repeated at the bottom of fig. 3(a) and the top of fig. 3(b)).


(a) First phase: sequence of expansions (positions 1 , 4 , 3 , 6 , 5 , 7 , 9 , 8 , and 6)

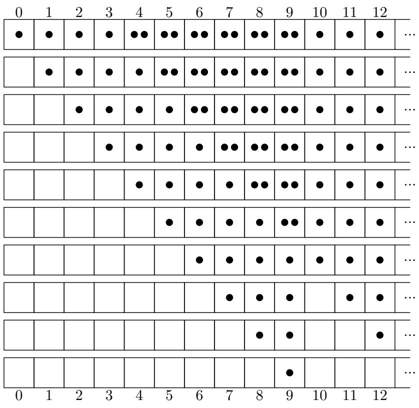

(b) Second phase: sequence of contractions (all positions from 1 upto 9)

Fig. 3. Moving a checker 8 places to the right when the move is given by $T = T + \psi_{2,3}$

We now consider the general problem. As it turns out, our solution involves a case analysis on the values of a and n . There are two cases that exhibit the same sort of symmetry as the "nuclear pennies game" and can be solved using the same strategy as was used for that game. These are (a) $2{=}a \wedge 2{=}n$ and (b) $3{=}a \wedge 1{=}n$. In case (a), $\psi_{a,n} = \psi_{2,2} = 1+T^2$. In case (b), $\psi_{a,n} = \psi_{3,1} = 1+T+T^2$. We leave these cases as elementary exercises for the reader. (Exploit the symmetry in the polynomials about T^1 and the left-right symmetry of the task. See section 4.1 for the strategy.) We split the remaining cases into (c) $4 \leq a \wedge 1 = n$ and (d) $2 \leq a \wedge 2 \leq n \wedge (2{\neq}a \vee 2{\neq}n)$.

Algorithm Decomposition. Considering the discussion in the previous section and motivated by the example shown in figure 3, we specify the algorithm we want to develop as follows:

```
{ $s = T$ }
perform a sequence of expansions
{ $s =$ “some intermediate state” } ;
perform a sequence of contractions
{ $s = T^{a^n+1}$ } .
```

Because moves are given by the equation $T = T + \psi_{a,n}$, we can model the *expansion* of position $k+1$ as

$$s \ := \ s + T^k \times \psi_{a,n} \quad ,$$

and we can model the *contraction* of position $k+1$ as

$$s \ := \ s - T^k \times \psi_{a,n} \quad .$$

This allows us to refine the specification:

```
{ $s = T$ }
do choose appropriate $k$;
    $s := s + T^k \times \psi_{a,n}$
od
{ $s =$ “some intermediate state” } ;
do choose appropriate $k$;
    $s := s - T^k \times \psi_{a,n}$
od
{ $s = T^{a^n+1}$ } .
```

The Intermediate State. Let us explore the “intermediate state”. An invariant of the first phase is that $s - T$ is divisible by $\psi_{a,n}$; an invariant of the second phase is that $s - T^{a^n+1}$ is divisible by $\psi_{a,n}$. So the “intermediate state” must be a polynomial s such that both $s - T$ and $s - T^{a^n+1}$ are divisible by $\psi_{a,n}$. Now,

$$\begin{aligned}
& T^{a^n+1} - T \\
= \quad & \{ \quad \text{geometric series} \quad \} \\
& (T^{a^{n-1}+1} - T) \times \langle \Sigma i : 0 \le i < a : T^{i \times a^{n-1}} \rangle \\
= \quad & \{ \quad \text{definition of } \psi_{a,n} \quad \} \\
& (T^{a^{n-1}+1} - T) \times \psi_{a,n} \quad .
\end{aligned}$$

It follows that, for all γ in $\mathbb{N}[T]$,

$$(13)\qquad T \;+\; (\gamma + T^{a^{n-1}+1}) \times \psi_{a,n} \;\;=\;\; T^{a^n+1} \;+\; (\gamma + T) \times \psi_{a,n} \;\; .$$

The left and right sides of this equation express our "intermediate state": the left side is the postcondition of the expansion phase and the right side is the precondition of the contraction phase. Our task is to choose γ in such a way that the intermediate state can be reached both by a sequence of expansions and the reverse of a sequence of contractions.

Contraction Phase. We start with the contraction phase which is much simpler. Note that 1 is a term in the polynomial $\psi_{a,n}$. This means that if an expansion is applied to position $k{+}1$ a checker is introduced at position k. Then an expansion can be applied to position k, introducing a checker at position $k{-}1$. And, of course, so on ad infinitum. In this way, a sequence of expansions starting from the state T^{a^n+1} (a checker in position $a^n{+}1$) and applied to positions $a^n{+}1$, a^n, ..., 1 will yield the state

$$T^{a^n+1} \;+\; \langle \Sigma i : 0 \le i < a^n{+}1 : T^i \rangle \times \psi_{a,n}$$

which has the form of the right side of (13). Reversing this process gives us the contraction phase:

```
{   $s \;=\; T^{a^n+1} + \langle \Sigma i : 0 \le i < a^n{+}1 : T^i \rangle \times \psi_{a,n}$   }
$k := 0$;
{ **Loop invariant:** $s = T^{a^n+1} + \langle \Sigma i : k \le i < a^n{+}1 : T^i \rangle \times \psi_{a,n}$   }
do   $k < a^n{+}1 \;\rightarrow\; s, k \;:=\; s - T^k \times \psi_{a,n} \,,\, k{+}1$
od
{   $s = T^{a^n+1}$   }
```

Expansion Phase. We now have to construct the loop corresponding to the expansion phase. The postcondition of the expansion phase is the precondition of the contraction phase. Because of (13), this is

$$s \;=\; T \;+\; (\langle \Sigma i \;:\; 0 \le i < a^n{+}1 \wedge i \ne 1 \;:\; T^i \rangle + T^{a^{n-1}+1}) \times \psi_{a,n} \;\; .$$

The precondition is $s = T$. Recalling the definition of an expansion of position $k{+}1$, we are required to expand each position in $\{1\} \cup \{i : 2 \le i < a^n{+}1 : i{+}1\}$ once together with the position $a^{n-1}{+}2$ a second time. Of course, since the initial state is that there is one checker at position 1, the first move is to expand position 1. Let $\mathcal{E}$ denote the bag of positions remaining to be expanded. That is,

$$\mathcal{E} \;=\; \{i : 2 \le i < a^n{+}1 : i{+}1\} \uplus \{a^{n-1}{+}2\} \;\; .$$

(The symbol "$\uplus$" denotes bag union.) Then, the expansion phase after the first move has been completed is implemented as follows.

```
{   $s \;=\; T{+}\psi_{a,n}$   }
$A\,,B \;:=\; \mathcal{E}\,,\emptyset$ ;
{ **Loop invariant:**        $s = T + \langle\Sigma i : i\in\{1\}\uplus B : T^{i-1}\rangle\times\psi_{a,n}$
                         $\wedge \quad A\uplus B = \mathcal{E}$   }
*do*  $A\neq\emptyset\rightarrow$      choose $j$ such that $j\in A$ and there is a checker
                  in position $j$ ;
                  $s\,,A\,,B \;:=\; s+T^{j-1}\times\psi_{a,n}\;,\;A-\{j\}\;,\;B\uplus\{j\}$
*od*
{   $s = T + (T^{a^{n-1}+1} + \langle\Sigma i : 0\leq i < a^n{+}1 \wedge i\neq 1 : T^i\rangle)\times\psi_{a,n}$   }
```

In order to expand a position it is required that there be a checker on that position. The correctness of the algorithm depends therefore on showing that, at each iteration, it is possible to choose a suitable value of j . (Formally, the "choose" statement is a conditional statement which will abort if j cannot be chosen.) Since the expansion phase never removes checkers (unlike the nuclear pennies game), it suffices to show that there is at least one way of ordering the elements of the bag $\mathcal{E}$ that guarantees that the position is occupied when the element is chosen. Such an ordering we called a *valid* ordering.

In the case that $1 = n$, the correctness of the algorithm is obvious. The bag $\mathcal{E}$ is then

$$\{i : 3\leq i < a{+}2 : i\} \uplus \{3\} \quad .$$

Position 3 has to be expanded twice (since $2\leq a$) and each of the positions 4 , ... , $a{+}1$ have to be expanded once. Assuming that $4\leq a$, there is indeed a checker at position 3 and the position can be expanded. This ensures that there are checkers at all positions 0 , 1 , ... , $a{+}1$. Subsequently, the positions 3 , ... , $a{+}1$ can be expanded in an arbitrary order. (In other words, any ordering that places position 3 first is a valid ordering).

The second case, when $2\leq a \wedge 2\leq n \wedge (2\neq a \vee 2\neq n)$, is the most difficult. The construction of a valid ordering reuses the central idea of the contraction phase, namely that once a checker has been introduced at position $k{+}1$ we can always expand positions k , $k{-}1$, $k{-}2$, etc.

In the following calculation, we determine a valid ordering for the expansions. Note the assumption $3\leq a^{n-1}$ in the first step . This is why we perform a case analysis. (The assumption is indeed satisfied when $2\leq a \wedge 2\leq n \wedge (2\neq a \vee 2\neq n)$.)

$$T + T^0\times\psi_{a,n}$$

$=\quad$ { assuming $3\leq a^{n-1}$, the coefficient of $T^{a^{n-1}}$ in $T^0\times\psi_{a,n}$ is 1
so we can expand positions a^{n-1} , $a^{n-1}{-}1$, ... , 4 , 3 }

$$T + T^0\times\psi_{a,n} + \langle\Sigma i : 2\leq i < a^{n-1} : T^i\rangle\times\psi_{a,n}$$

$=$ { now the coefficient of $T^{a^{n-1}+2}$ in $T^2 \times \psi_{a,n}$ is 1
so we can expand positions $a^{n-1}+2$ and $a^{n-1}+1$ }

$T + T^0 \times \psi_{a,n} + \langle \Sigma i : 2 \leq i < a^{n-1}+2 : T^i \rangle \times \psi_{a,n}$

$=$ { $\langle \Sigma i : 2 \leq i < a^{n-1}+2 : T^i \rangle \times \psi_{a,n} = \langle \Sigma i : 0 \leq i < a^n : T^{i+2} \rangle$ }

$T + T^0 \times \psi_{a,n} + \langle \Sigma i : 2 \leq i < a^n+2 : T^i \rangle$

$=$ { now expand positions $a^{n-1}+2$, $a^{n-1}+3$, $\ldots$, a^n+1 }

$T + T^0 \times \psi_{a,n} + \langle \Sigma i : 2 \leq i < a^n+2 : T^i \rangle$

$+ \quad \langle \Sigma i : a^{n-1}+1 \leq i < a^n+1 : T^i \rangle \times \psi_{a,n}$

$=$ { $\langle \Sigma i : 0 \leq i < a^n : T^{i+2} \rangle = \langle \Sigma i : 2 \leq i < a^{n-1}+2 : T^i \rangle \times \psi_{a,n}$ }

$T + T^0 \times \psi_{a,n} + \langle \Sigma i : 2 \leq i < a^{n-1}+2 : T^i \rangle \times \psi_{a,n}$

$+ \quad \langle \Sigma i : a^{n-1}+1 \leq i < a^n+1 : T^i \rangle \times \psi_{a,n}$

$=$ { range splitting }

$T + (\langle \Sigma i : 0 \leq i < a^n+1 \wedge i \neq 1 : T^i \rangle + T^{a^{n-1}+1}) \times \psi_{a,n}$.

In summary, a valid ordering of expansions is, first, position 1 , then positions a^{n-1} , $a^{n-1}-1$, $\ldots$, 4 , 3 , then positions $a^{n-1}+2$ and $a^{n-1}+1$ and finally positions $a^{n-1}+2$, $a^{n-1}+3$, $\ldots$, a^n+1 .

This completes the expansion phase and the algorithm.

5 Conclusion

The games presented here were developed in order to support teaching of invariant properties in introductory, university-level courses. The current presentation of their solution is possibly too difficult for that level but they could be used to support more advanced algorithmically oriented courses that depend on polynomial arithmetic. (Coding theory is an obvious example).

The common theme of all our examples is the exploitation of simple invariant properties of polynomials. It is this insight that enabled us to invent the cyclotomic games in section 4.2 (which we believe to be original to this paper). We are currently trying to find a complete characterisation of games based on the equation

$$T^k = T^k + \Gamma$$

where k is a positive number and Γ is a polynomial in T with positive integer coefficients. (The generalisation from 1 to k makes the contraction phase harder).

Some improvement on our derivations would be welcome. We have been obliged to use case analyses in both the solution of the tiling problem (section 3.2) and the solution of the cyclotomic games (section 4.2). This is unfortunate but apparently unavoidable.

The class of tiling problems we have discussed assumes that all "ominoes" are straight. Golomb [1] extends his colouring argument to other problems where the ominoes are not straight. We haven't explored these problems. It would be interesting to see whether the algebraic formulation can be extended to such problems in a uniform way.

The interested reader may want to prove that our solutions to the cyclotomic games minimise the number of moves. The generalisation of the games to higher dimensions may also be of interest.

Acknowledgement. Our thanks to Diethard Michaelis for his comments on earlier drafts of this paper.

References

1. Golomb, S.W.: Polyominoes. George Allen & Unwin Ltd. (1965)
2. Mackinnon, N.: An algebraic tiling proof. The Mathematical Gazette 73(465), 210–211 (1989)
3. Piponi, D.: Arboreal isomorphisms from nuclear pennies (September 2007), Blog post available at, `http://blog.sigfpe.com/2007/09/arboreal-isomorphisms-from-nuclear.html`
4. Chen, W., Backhouse, R.: From seven-trees-in-one to cyclotomics (2010) (submitted for publication), `http://cs.nott.ac.uk/~wzc`
5. de Bruijn, N.G.: A Solitaire game and its relation to a finite field. J. of Recreational Math. 5, 133–137 (1972)
6. Berlekamp, E.R., Conway, J.H., Guy, R.K.: Winning Ways, vol. I, II. Academic Press, London (1982)
7. Reiss, M.: Beitrage zur Theorie der Solitär-Spiels. Crelle's J. 54, 344–379 (1857)
8. Dijkstra, E.W.: The checkers problem told to me by M.O. Rabin (September 1992), `http://www.cs.utexas.edu/users/EWD/ewd11xx/EWD1134.PDF`
9. Piponi, D.: Using thermonuclear pennies to embed complex numbers as types (October 2007), Blog post available at, `http://blog.sigfpe.com/2007/10/using-thermonuclear-pennies-to-embed.html`
10. Blass, A.: Seven trees in one. Journal of Pure and Applied Algebra 103(1), 1–21 (1995)
11. Fiore, M.: Isomorphisms of generic recursive polynomial types. In: Proceedings of the 31st Annual ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages, pp. 77–88. ACM Press, New York (2004)
12. Lawvere, F.W.: Some thoughts on the future of category theory. Lecture Notes in Mathematics, vol. 1488, pp. 1–13 (1991)

Compositionality of Secure Information Flow

Catuscia Palamidessi

INRIA and LIX, École Polytechnique, Palaiseau, France

One of the concerns in the use of computer systems is to avoid the leakage of confidential information through public outputs. Ideally we would like systems to be completely secure, but in practice this goal is often impossible to achieve. Therefore it is important to have a way to quantify the amount of leakage, so to be able to assess that a system is better than another, although they may both be insecure. Recently there have been various proposals for quantitative approaches. Among these, there is a rather natural one which is based on the Bayes risk, namely (the converse of) the probability of guessing the right value of the secret, once we have observed the output [1]. The main other quantitative approaches are those based on Information Theory: intuitively indeed the information leakage can be thought of as the certainty we gain about the secret by observing the output, and the (un)certainty of a random variable is represented by its *entropy*. The information-theoretic approaches, in the early proposals (see for instance [2,3,4]), were based on the most common notion of entropy, namely Shannon entropy. However Smith has argued in [5] that Shannon entropy, due to its averaging nature, is not very suitable to represent the vulnerability of a system, and he has proposed to use Rényi's min entropy [6] instead. In the same paper, Smith has also shown that the approach based on Rényi's min entropy is equivalent to the one based on the Bayes risk.

In this work, which continues a line of research initiated in [7], we consider a formalism for the specification of systems composed by concurrent and probabilistic processes, and we investigate "safe constructs", namely constructs which do not increase the vulnerability.

References

1. Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: On the Bayes risk in information-hiding protocols. Journal of Computer Security 16(5), 531–571 (2008)
2. Clark, D., Hunt, S., Malacaria, P.: Quantitative information flow, relations and polymorphic types. J. of Logic and Computation 18(2), 181–199 (2005)
3. Zhu, Y., Bettati, R.: Anonymity vs. information leakage in anonymity systems. In: Proc. of ICDCS, pp. 514–524. IEEE, Los Alamitos (2005)
4. Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: Anonymity protocols as noisy channels. Inf. and Comp. 206(2-4), 378–401 (2008)
5. Smith, G.: On the foundations of quantitative information flow. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 288–302. Springer, Heidelberg (2009)
6. Rényi, A.: On Measures of Entropy and Information. In: Proc. of the 4th Berkeley Symposium on Mathematics, Statistics, and Probability, pp. 547–561 (1960)
7. Braun, C., Chatzikokolakis, K., Palamidessi, C.: Compositional methods for information-hiding. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 443–457. Springer, Heidelberg (2008)

Process Algebras for Collective Dynamics
(Extended Abstract)

Jane Hillston

Laboratory for Foundations of Computer Science,
The University of Edinburgh, Scotland

Quantitative Analysis

Stochastic process algebras extend classical process algebras such as CCS [1] and CSP [2] with quantified notions of time and probability. Examples include PEPA [3], EMPA [4], MoDeST [5] and IMC [6]. These formalisms retain the compositional structure of classical process algebras and the additional information captured within the model allows analysis to investigate additional properties such as dynamic behaviour and resource usage.

Stochastic process algebras have been successfully applied to quantitative evaluation of systems for over a decade. For example, in the context of performance analysis, PEPA has been used to describe both software and hardware systems and has helped to incorporate early performance prediction into the design process. Moreover, recently there has been considerable interest in using stochastic process algebras for modelling intracellular processes in systems biology.

In all these models the entities in the system under study are represented as components in the process algebra. The structured operational semantics of the language is used to identify all possible behaviours of the system as a labelled transition system. With suitable restrictions on the form of random variables used to govern delays within the model to be negative exponentially distributed this labelled transition system can be interpreted as a continuous time Markov chain (CTMC). This provides access to a wide array of analysis techniques, usually in terms of the evolution of the probability distribution over states of the model over time.

This has the advantage that it is a fine-grained view of the system, allowing the quantitative characteristics of individual entities to be derived. Unfortunately it has the disadvantage that generation and manipulation of the necessary CTMC can be very computationally expensive, or even intractable, due to the well-known *state space explosion* problem. This problem becomes particularly acute in situations where there are large numbers of entities exhibiting similar behaviour interacting within a system. Often in these situations whilst it is important to capture the behaviour of individual entities accurately the dynamics of the system are most fruitfully considered at a population level. Examples include the spread of disease through a population, the behaviour of crowds during emergency evacuation of a building or scalability studies involving a large number of users trying to access a service. In these cases we are interested in the *collective* rather than the *individual* dynamics.

Collective Dynamics

Process algebras have several attractions for modelling for collective dynamics. The behaviour of individuals, and particularly their interactions are important for such systems, and the compositional approach of the process algebra allows the modeller to capture the exact form of interactions and constraints between entities. However, standard approaches to analysis of process algebra models remain focused on the behaviour of individuals and are inherently discrete event-based. As explained above, this leads to the state space explosion problem and makes it difficult to construct models large enough to exhibit the population level effects which we are interested in.

Thus at Edinburgh we have been investigating the use of process algebras for collective dynamics based on alternative semantics for the constructed models, which consider the population rather than the individuals. As observed above, the semantics of individual-oriented stochastic process algebra models generally gives rise to a discrete state space with dynamics captured by a continuous time Markov chain. In the context of collective dyanamics, an alternative mathematical framework based on sets of ordinary differential equations is used. This may be regarded as a fluid approximation of the discrete state model [7] and recent work has shown how this may be accessed directly via a novel symbolic structured operational semantics [8]. This provides a framework in which to establish the relationship between the two alternative forms of representation.

References

1. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
2. Hoare, C.: Communicating Sequential Processes. Prentice Hall, Englewood Cliffs (1985)
3. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press, Cambridge (1996)
4. Bernardo, M., Gorrieri, R.: A tutorial on EMPA: a theory of concurrent processes with nondeterminism, priorities, probabilities and time. TCS 202, 1–54 (1998)
5. D'Argenio, P., Hermanns, H., Katoen, J.P., Klaren, R.: Modest — a modelling and description language for stochastic timed systems. In: de Luca, L., Gilmore, S. (eds.) PROBMIV 2001, PAPM-PROBMIV 2001, and PAPM 2001. LNCS, vol. 2165, p. 87. Springer, Heidelberg (2001)
6. Hermanns, H.: Interactive Markov Chains. LNCS, vol. 2428, p. 57. Springer, Heidelberg (2002)
7. Hillston, J.: Fluid flow approximation of PEPA models. In: Proc. of the 2nd International Conference on Quantitative Evaluation of Systems (2005)
8. Tribastone, M., Gilmore, S., Hillston, J.: Scalable differential analysis of process algebra models. IEEE Transactions on Software Engineering (to appear, 2010)

On Automated Program Construction and Verification

Rudolf Berghammer[1] and Georg Struth[2]

[1] Institute of Computer Science, Christian-Albrechts-University of Kiel, Germany
rub@informatik.uni-kiel.de
[2] Department of Computer Science, University of Sheffield, UK
g.struth@dcs.shef.ac.uk

Abstract. A new approach for automating the construction and verification of imperative programs is presented. Based on the standard methods of Floyd, Dijkstra, Gries and Hoare, it supports proof and refutation games with automated theorem provers, model search tools and computer algebra systems combined with "hidden" domain-specific algebraic theories that have been designed and optimised for automation. The feasibility of this approach is demonstrated through fully automated correctness proofs of some classical algorithms: Warshall's transitive closure algorithm, reachability algorithms for digraphs, and Szpilrajn's algorithm for linear extensions of partial orders. Sophisticated mathematical methods that have been developed over decades could thus be integrated into push-button engineering technology.

1 Introduction

Programs without bugs is one of the great ideals of computing. It motivated decades of research on program construction and verification. A commonality of most approaches to program correctness is the combination of mathematical models of programs with mechanised program analysis tools. It requires integrating the science of programming into the engineering of programs.

Provably correct programs can be obtained in different ways: *Program construction* or *synthesis* means deriving executable programs from mathematical models or specifications, so that a program is correct if its derivation is sound. *Program verification* means proving that a given program satisfies a set of assertions provided by the programmer, usually by demonstating that whenever it initially meets certain constraints (a precondition) it will satisfy a certain property (a postcondition) upon termination. In the presence of loops, invariants must be maintained during their execution.

Models for programs and their properties have been based on different mathematical formalisms. Relational calculi are among the most ubiquitous ones. They form the basis of established methods like Alloy [16], *B* [1] or *Z* [22]. Tool support has usually been integrated through interactive theorem provers or finitist methods such as model checkers or SAT-solvers. To be useful in practice, modelling languages must be simple but expressive, and tool support should be as invisible and automatic as possible. These requirements are incompatible and need to be balanced to yield methods that are lightweight yet powerful.

Our main contribution is an approach to the construction and verification of imperative programs which aims at this balance in a novel way through computer enhanced mathematics. Its backbone is a combination of off-the-shelf automated theorem proving systems (ATP systems), model generators and computer algebra systems with domain-specific algebras that are designed and optimised for automation. This combination allows automatic program correctness proofs, but it also supports program development at a more fundamental level through the inference of specification statements and algorithmic properties in a game of proof and refutation. While algebraic theories and automation technology can largely be hidden behind an interface, developers can focus on the conceptual level and use simple intuitive relational languages for modelling and reasoning.

A second contribution is the demonstration of the potential and feasibility of this approach through partial correctness proofs of some classical algorithms:

- Warshall's algorithm for the transitive closure of a relation,
- two reachability algorithms for directed graphs,
- Szpilrajn's algorithm for linear extensions of partial orders.

While Warshall's and Szpilrajn's algorithm are constructed from specifications, the reachability algorithms are verified. In our current scenario, assertions must be provided by the developer for verification. Similarly, for program construction, we do not automatically synthesise invariants or abduce preconditions, but we provide automated tool support for developing and formally justifying these tasks. In particular, in both cases, all proof obligations are discharged fully automatically. We use different domain-specific algebras at the dark side of the interface, and also discover a new refinement law for reflexive transitive closures in our case studies. All ATP proofs in this paper can be found at a web site [13].

We build on decades of work on formal methods, logics and algebras for programs, ATP technology, and program construction. At the engineering side, our use of relational calculi is inspired by formal methods like Alloy, B or Z; our approach to program correctness is essentially that of Dijkstra and Gries [11] and closely related to Hoare logic [12]. We use the ATP system Prover9 and the model generator Mace4 [19] in our proof experiments, but most other state-of-the-art ATP systems and model generators would yield a similar performance. For testing and visualising relational programs we use the RelView computer algebra system [5]. At the mathematical side, we use Tarski's relation algebras [24] and various reducts, such as idempotent semirings, Kleene algebras or domain semirings [8,17]. Our case studies build on previous manual calculational correctness proofs in relation algebra [3,6], but most of our new proofs are different, sometimes simpler, and valid in a larger class of models.

The power and main novelty of our approach lies in the balance of all these theories, methods and tools. However, presently, we have not much to show beyond the proof of concept. Further work will be needed to transform our ideas into program construction and verification tools that will be useful for teaching and software development practice.

2 A Simple Relational Modelling Language

For constructing and verifying programs we do not assume much more than an intuitive conceptual and operational understanding of binary relations and sets with their fundamental operations and properties, as needed for basic modelling with Alloy or RelView. To obtain a uniform readable syntax, and to make it easy to replay our automation experiments, we use a notation that can be processed by Prover9 and Mace4 although it deviates from most textbooks.

As usual, a *binary relation* x on a set A is a subset of the Cartesian product $A{\times}A$, a set of ordered pairs on A. Our approach extends to heterogeneous relations of type $A{\times}B$, but this would only overload the presentation. As sets, relations form Boolean algebras and inherit the Boolean operations $+$ of union or join, $*$ of intersection or meet and $'$ of complementation; there is an empty relation 0 and a universal relation $U = A{\times}A$. The *relative product* $x;y$ of two binary relations x and y is formed by the ordered pairs (a,b) with $(a,c) \in x$ and $(c,b) \in y$ for some $c \in A$. The *converse* $x^\wedge$ of a binary relation x is formed by the ordered pairs (b,a) with $(a,b) \in x$. The *identity relation* 1 consists of all ordered pairs (a,a) with $a \in A$.

Apart from these basics we assume that readers are familiar with the algorithmics of finite binary relations as presented in undergraduate textbooks [7]. Briefly, finite binary relations can be represented as directed graphs (digraphs) — ordered pairs corresponding to vertices linked by arrows — and implemented via the algebra of Boolean adjacency matrices. The relational operations are reflected in the matrix algebra: join by matrix sum, relative product by matrix product, conversion by matrix transposition; the identity relation by the diagonal matrix, the empty relation by the zero matrix, and the universal relation by the matrix in which each element is 1. Sets can either be implemented as vectors, as row-constant matrices or as subidentity matrices with ones at most along the diagonal. The multiplication of a matrix with a vector corresponds to computing the preimage (a set) of a set with respect to a relation. The matrix representation provides an important intuition for our case studies; all our programs can be implemented directly as matrix algorithms in the RelView tool.

Only a few additional concepts are needed for our case studies. The *reflexive transitive closure* and the *transitive closure* of a relation x are

$$\mathsf{rtc}(x) = \sum_{i \geq 0} x^i \qquad \text{and} \qquad \mathsf{tc}(x) = \sum_{i \geq 1} x^i,$$

with powers x^i defined inductively. The *domain* $d(x)$ and *range* $r(x)$ of x are defined as the sets

$$d(x) = \{a \in A : \exists b \in A.(a,b) \in x\},$$
$$r(x) = \{b \in A : \exists a \in A.(a,b) \in x\}.$$

Throughout this paper, we will call singleton sets *points* and single ordered pairs *atoms*. The technicalities of reasoning with relations are delegated as far as possible to tools. They are hidden from developers behind an interface.

3 The Dark Side of the Interface

For automated program construction and verification, the relational concepts discussed in the previous section must be implemented in domain-specific theories that are suitable and optimised for automated proof search. Developers are not supposed to see these theories in detail, they are hidden behind an interface. We use variants and reducts of relation algebras (in the sense of Tarski [18,21,24]) as domain-specific theories in our case studies.

A *relation algebra* is a structure $(R, +, *, ', 0, U, ;, 1, ^\wedge)$ that satisfies the following axioms taken from Maddux's textbook [18].

```
x+y=y+x  &  x+(y+z)=(x+y)+z  &  x=(x'+y')'+(x'+y)'.
x;(y;z)=(x;y);z  &  x;1=x  &  (x+y);z=(x;z)+(y;z).
(x^)^=x  &  (x+y)^=x^+y^  &  (x;y)^=y^;x^  &  x^;(x;y)'+y'=y'.
```

These axioms are effectively executable by Prover9 and Mace4 [15]; complete input files can be found in a proof data base [13]. The first line contains Huntington's axioms for Boolean algebras, the second line those for relative products, and the third line those for conversion. The following standard definitions are always included in our input files.

```
x*y=(x'+y')'  &  x<=y <-> x+y=y  &  0=x*x'  &  U=x+x'.
% x!=0 -> U;(x;U)=U.
```

The first equation defines meet via De Morgan's law. The second formula defines the standard order of the algebra. The next two equations define the least and the greatest element of the algebra. Tarski's axiom in the second line is needed for proving one single auxiliary lemma in this paper. Apart from this, it has not been used and is therefore commented out.

Following Ng [20], we axiomatise the reflexive transitive closure $\mathsf{rtc}(x)$ of a binary relation x as a least fixed point:

```
1+x;rtc(x)=rtc(x)  &  z+x;y<=y -> rtc(x);z<=y.
1+rtc(x);x=rtc(x)  &  z+y;x<=y -> z;rtc(x)<=y.
```

The transitive closure $\mathsf{tc}(x)$ of x is defined as `tc(x)=x;rtc(x)`. The expressions `rtc(x)` and `tc(x)` can now be used at the developer's side of the interface for modelling and reasoning about programs, whereas the implementation of these concepts in relation algebra at the dark side of the interface is used for automated reasoning with Prover9 and Mace4, but need not concern the developer.

It is standard to model sets in relation algebras either as *vectors* or as *subidentities* (elements below 1). We present the first alternative for relation algebras and the second one for Kleene algebras below. Both allow us to implement points and atoms. In fact, we only need *weak points*, which are points or zero, and *weak atoms*, which are atoms or zero.

```
inj(x) <-> x;x^<=1.                            % def injection
vec(x) <-> x=x;U.                              % def vector
wpoint(x) <-> vec(x) & inj(x).                 % def weak point
watom(x) <-> wpoint(x;U) & wpoint(x^;U).       % def weak atom
```

At the developer's side of the interface, `set(x) <-> vec(x)` can be used for typing sets, and the predicates `wpoint` and `watom` type (weak) points and (weak) atoms. The right-hand sides of these definitions implement these concepts. They can again be hidden. Implementation details are not needed to understand our case studies; a discussion can be found in the literature [21].

Finally, the domain of a relation x is implemented as `d(x)=1*x;U`, and the range `r(x)=d(x^)` as the domain of the converse of x.

The calculus of relations can effectively be automated [15]. But experiments show that ATP systems still have difficulties with proving correctness of complex programs from the axioms of relation algebras and auxiliary concepts alone. Domain-specific and problem-specific theories and assumption sets must be engineered for applications. Adding assumptions requires libraries of verified relational properties [13]. Enhancing proof search requires reducts of relation algebras. Variants of idempotent semirings and Kleene algebras are known to be very suitable in this respect [8,14]. All verified facts about binary relations can, of course, safely be used as independent assumptions with these reducts.

An *idempotent semiring* is a structure $(S, +, ;, 0, 1)$ that satisfies the axioms

```
x+y=y+x   &   x+(y+z)=(x+y)+z   &   x+0=x   &   x+x=x.
x;(y;z)=(x;y);z   &   x;1=x   &   1;x=x   &   x;0=0   &   0;x=0.
x;(y+z)=x;y+x;z   &   (x+y);z=x;z+y;z.
x<=y <-> x+y=y.
```

An idempotent semiring expanded by the reflexive transitive closure operation axiomatised above is a *Kleene algebra*. Conversion can now be axiomatised as

```
(x^)^=x   &   (x+y)^=x^+y^   &   (x;y)^=y^;x^   &   x<=x;(x^;x).
```

The universal relation can be axiomatised as `x<=U`.

Sets can again be modelled as vectors or as subidentities, but, in contrast to relation algebras, the subalgebras of all subidentities in idempotent semirings or Kleene algebras are not necessarily Boolean algebras.

The simplest approach — and best suited for ATP — uses *domain semirings* and *Kleene algebras with domain* [8]. Domain is axiomatised via an *antidomain* operation:

```
a(x);x=0   &   a(x;a(a(y)))=a(x;y)   &   a(a(x))+a(x)=1.
```

Intuitively, the antidomain $a(x)$ of a relation x is the set of all elements which are not in the domain $d(x)$ of x, hence `d(x)=a(a(x))`. Axiomatic details are again not important, but the following two properties are essential for understanding our implementation of sets across the interface: First, the set $d(S)$ of all domain elements of S is precisely the set of all $x \in S$ that satisfy $x = d(x)$. Second $d(S)$ forms a Boolean algebra among the subidentities of S and the meet operation is multiplication. In this setting, we can therefore type `set(x) <-> d(x)=x`.

The notion of range is dual to that of domain. Its axiomatisation only requires swapping the order of multiplication.

In the context of domain semirings, points can be implemented via *rectangular* relations. The corresponding axioms are

```
rctangle(x) <-> x;(U;x)=x.
wpoint(x) <-> set(x) & rctangle(x).
```

Intuitively, a relation is rectangular if it is equal to the cartesian product of its domain and range, and this explains why points are rectangular sets[1]. In some sense, rectangles can be understood as generalised points.

4 Automation Technology Review

In this section we briefly sketch the Dijkstra-Gries approach to program development and discuss the three tools used for its automation behind the interface.

The Dijkstra-Gries Approach. As mentioned in the introduction, program construction means that a program is derived from a specification, and program verification means that a given program is proved to be correct with respect to a given specification. For imperative programs, specifications usually consist of preconditions, postconditions and invariants which model the inductive properties implemented in the body of a loop. The correctness of a simple while loop is implied by the following proof obligations:

1. The invariant is established by the initialisation (before the loop starts)
2. Each execution of the loop's body preserves the invariant, as long as the guard of the loop is true.
3. The invariant establishes the postcondition if the guard of the loop is false.
4. The loop terminates.

We will automatically analyse the first three proof obligations in our case studies, hence prove *partial program correctness*. We do not formally consider termination (*total correctness*) since this requires a different kind of analysis.

For program verification, the precondition, the postcondition and the invariant are added as assertions to the given program code, and the proof obligations can then be generated and analysed automatically. This approach is based on the seminal work of Floyd and Hoare [10,12].

For program construction, the invariant is hypothesised as a modification of the postcondition and the proof obligations are then established step by step together with the synthesis of the program: the guard of its loop, the initialisation of the variables to establish the invariant and the assignments in the loop's body to maintain it. Based on the fundamental principle that "a program and its correctness proof should be developed hand-in-hand with the proof usually leading the way" ([11], p. 164), the approach has been pioneered by Dijkstra [9], and elaborated by Gries [11] and others into a variety of techniques.

[1] The usual definition of rectangles in relation algebras uses only $x;U;x \leq x$, and we could automatically verify that this is equivalent to $x;U;x = x$. In Kleene algebra, in contrast, the equational definition is strictly stronger than the original one; they are separated by a three element counterexample found by Mace4.

Tools for Proofs and Refutations. To automate all synthesis and verification proofs, we use the ATP system Prover9 [19]. The main reason is that its input syntax is very readable and that the model generator Mace4 [19] is based on the same syntax. This makes it particularly easy to replay our proof experiments. The interplay of ATP systems and model generators is essential for our games of proof and refutation in program construction.

Prover9 is complete for first-order logic with equality; it allows reasoning with relation algebra and all reducts we consider. It takes a set of assumptions and a proof goal and uses sophisticated proof search and redundancy elimination strategies combined with complex heuristics to deduce the goal from the assumptions. We use the tool entirely as a black box and as push-button technology. Prover9 is only a semi-decision procedure. In theory it can prove all first-oder theorems, but may run forever on non-valid proof goals. In practice, however, it often runs out of steam without proving a theorem. On success, Prover9 outputs a proof (which is usually not revealing for humans). Mace4 searches for finite models. It accepts essentially the same input as Prover9, and tries to construct a model of the assumptions that fails the proof goal, that is, a counterexample.

Prover9 and Mace4 support infix and postfix notation for algebraic operators and precedence declarations. For relation algebra, we use the following code:

```
op(500, infix,   "+").  % join
op(480, infix,   "*").  % meet
op(300, postfix, "'").  % complementation
op(450, infix,   ";").  % composition
op(300, postfix, "^").  % conversion
```

Operations with a lower number bind more strongly than those with a higher number. For Kleene algebra, we use the following declaration:

```
op(500, infix,   "+").
op(490, infix,   ";").
```

Assumptions and goals must be put into the following environment:

```
formulas(assumptions).     ...     end_of_list.
formulas(goal).            ...     end_of_list.
```

Examples for input and output files, and the complete set of proofs for our case studies, can be found in a proof database [13]. We have so far verified more than 500 theorems of relation algebras and Kleene algebras, including more or less all "textbook theorems".

In our experiments, to demonstrate the robustness of the approach, we always use the weakest possible assumption set, ideally the theory axioms and basic definitions alone, at the expense of long running times. Adding the right lemmas would usually bring proof search down to a few seconds.

Besides Prover9 and Mace4 we also use the RelView system [5]. This is an interactive and graphics-oriented special purpose computer algebra system for the visualisation and manipulation of binary relations, and for relational prototyping and programming. RelView is optimised for very large objects, for instance, membership relations, which is especially important for prototyping. It uses an efficient internal implementation of relations via reduced ordered binary decision diagrams [4]. RelView provides predefined operations and tests for modelling and analysing relations in the lightweight style sketched in Section 2. (We do not use the RelView modelling language since it clashes with Prover9 syntax.) Relational functions and programs can be built from these operations. Relational functions are introduced as usual in mathematics, and relational programs are essentially while-programs based on relational datatypes.

Within our approach, the main applications of RelView are specification testing and support for reasoning with concrete finite binary relations. Specification testing includes mainly the evaluation of relational specifications and the comparison of the results obtained with original specifications in another formalism, or even the intuitive notion of the problem, in order to find weaknesses or inconsistencies. Support for relational reasoning is very important in program construction for finding and analysing loop invariants, that is, whether a candidate for an invariant satisfies or violates a proof obligation in some special cases. RelView allows developers to experiment with programs and assertions and to visualise this information via graphical representations of relations. Particularly useful for program construction and verification is the fact that the tool allows testing validity of arbitrary Boolean combinations of relational inclusions via an `ASSERT` command and that the relations needed can be randomly generated.

5 Synthesis of Warshall's Algorithm

As a first case study, we construct Warshall's classical algorithm [25] for computing the transitive closure of a finite binary relation — a digraph — from its specification. We carefully separate the developer's view from the domain-specific theory — in this case Kleene algebra with domain — and the automation technology at the dark side of the interface. We also aim at illustrating how a proof and refutation game with Prover9, Mace4 and RelView is essential in this construction.

We use the Dijkstra-Gries framework to derive a simple while-program from a given precondition and postcondition, inferring the loop invariant, the guard of the loop and the variable assignments along the way. This development is inspired by a previous manual correctness proof in relation algebra [3].

Initial Specification. Consider the following program construction task:

> *Given a finite binary relation x, find a program with a relational variable y that stores the transitive closure of x after its execution.*

We aim at a while-program of the following schematic form:

```
... y:=x ...
while ... do
  ... y:=? ... od
```

The precondition and postcondition are evident from the above specification:

```
pre(x) <-> x=x.
post(x,y) <-> y=tc(x).
```

The formula $x = x$ expresses for Prover9 that the precondition is always true. This is the case because the input relation x can be arbitrary. The proof obligations from Section 4 guide us through the synthesis of the initialisation, the guard and the body of the loop.

Developing the Invariant. The most important ingredient of our construction is still missing: Warshall's insight that makes the algorithm work.

> *Initially, compute only those paths contributing to the transitive closure of x that traverse no inner vertices. Then iteratively add inner vertices and compute the local transitive closure restricted to each new inner-vertex set incrementally from that of its predecessor set. Terminate when all possible new inner vertices have been added.*

How the incremental computation can be achieved will concern us later, but the invariant of the algorithm should by now be evident:

> *The variable y must maintain the transitive closure of x restricted to each set v of inner vertices that is constructed along the way.*

Formally, for Prover9, we obtain:

```
inv(x,y,v) <-> (set(v) -> y=rtc(x;v);x).
```

Note that $(x; v)^k; x$ yields the first and last points of x-paths with k inner vertices from the set v, and that sets are implemented as subidentities via domain.

Initialisation, and Guard. According to the specification, the set variable v should be initialised as `v:=0` and the loop should terminate when $v = 1$, or, even better, when $v = d(x)$, that is, when all vertices from which x is enabled have been visited. The guard of the loop should therefore be `v!= d(x)` (or `v!= 1`). We can justify these assumptions by verifying the following proof obligations.

Theorem 1. *The invariant is established by the initialisation (if the precondition holds); it establishes the postcondition when the guard of the loop is false.*

Proof. Using Kleene algebra with domain (behind the interface) we proved

```
pre(x) -> inv(x,x,0).
inv(x,y,v) & v=1 -> post(x,y).
inv(x,y,v) & v=d(x) -> post(x,y).
```

Since Prover9 can only handle one non-equational proof goal at a time, two goals always need to be commented out. The assumption file contains the axioms for Kleene algebras with domain, and the definition of set, transitive closure, precondition, postcondition and invariant, as listed above. The proofs of the first and the third goal were instantaneous and very short. The proof of the second goal was slightly harder. It required about 18s and has 129 steps. □

Termination and Development of the Loop. We now consider termination of the algorithm, and synthesise the body of the loop by considering the proof obligation that the invariant be preserved when executing the loop. This means synthesising assignments to the set variable v and the relational variable y:

- The assignment to v is obvious from the above discussion. We add a single new point p to the set v; `v:=v+p`.
- The assignment of y should, if possible, increment y, which stores the transitive closure of x restricted to v, by the transitive closure computed incrementally with respect to y and p. So we postulate `y:=y+f(y,p)`.

Our program should then have the following form:

```
y,v:=x,0
while v!=d(x) do
  p:=point(v')
  y,v:=y+f(y,p),v+p od
```

Here, `point` is a choice function that returns some point from the complement set of v, which is taken at the set level. It satisfies the axiom

```
wpoint(point(x)) & point(x)!=0.
```

but this is not needed in our development apart from termination. At the dark side of the interface, v' is translated to $a(v)$. Now, the assignment `v:=v+p` with v and p disjoint enforces termination of the loop. This is the only part of the proof which we do not automate. It remains to determine `y+f(y,p)`.

We compare the value y before and after the assignment to v. Before the assignment, $y = \mathsf{rtc}(x; v); x$. After the assignment `v:=v+p` we have the value

$$y = \mathsf{rtc}(x; (v + p)); x = \mathsf{rtc}(x; v + x; p).$$

So we could try to refine the reflexive transitive closure of this sum into a sum of reflexive transitive closures.

Consider $\mathsf{rtc}(a + b)$ for arbitrary relations a and b. First, the most straightforward refinement into $\mathsf{rtc}(a) + \mathsf{rtc}(b)$ can be refuted by a six-element counterexample. Hence we need to consider this expression in more detail.

Obviously, $\mathsf{rtc}(a + b)$ represents arbitrary alternating sequences of a and b, hence should be equal to $\mathsf{rtc}(\mathsf{rtc}(a); \mathsf{rtc}(b))$. This has already been verified by ATP [13]. Such sequences could form either one single a-block (possibly empty), or alternating blocks of a and b. But $b = x; p$ is not an arbitrary relation. In the matrix model, the point p projects x onto a matrix in which only one single non-zero row. This can be visualised in experiments with RelView. In other words, b is a rectangle and we conjecture the following more general fact.

Lemma 2. *Let x, y be elements of a relation algebra in which Tarski's axiom holds. If y is rectangular, then $x; y$ is rectangular.*

Proof. Using the axioms of relation algebras, Tarski's axiom and the above definition of rectangles, we proved `rctangle(y) -> rctangle(x;y)` by ATP. □

Interestingly, Lemma 2 does not hold in all Kleene algebras (Mace4 found a three-element counterexample) or for relation algebras without Tarski's axiom. But we can safely add it as an independent assumption.

The fact that $b = x;p$ is a rectangle has some impact on the alternating blocks of a and b. Intuitively, rectangles can be seen as generalised points. Many of their properties can be conjectured by thinking about points in the first place and then proved or refuted by Prover9 and Mace4. First, $b;b = b * b = b$, hence all b-blocks must have length one. Second, $b;\mathsf{rtc}(a);b \leq b$, since in the left-hand side of this inequality, all inputs and outputs are projected onto the rectangle b. Hence, if b is rectangular, the developer might conjecture that $\mathsf{rtc}(a+b)$ can be refined to $\mathsf{rtc}(a) + \mathsf{rtc}(a);b;\mathsf{rtc}(a)$. And indeed we can prove the following new refinement law for reflexive transitive closures that is of general interest.

Proposition 3. *Let x, y be elements of a Kleene algebra with greatest element, and let y be rectangular. Then*

$$\mathsf{rtc}(x+y) = \mathsf{rtc}(x) + \mathsf{rtc}(x);y;\mathsf{rtc}(x).$$

Proof. By ATP, using the Kleene algebra axioms and the definitions of universal relation and rectangle. The $\geq$-proof took less than 10s. The $\leq$-proof took about 30s; it has 56 steps. □

Proposition 3 allows us to refine the term $(x;v + x;p)$ in the assignment of y since $x;p$ is rectangular by Lemma 2. We can therefore incrementally compute the transitive closure of x restricted to inner vertices in $v+p$ from that restricted to inner vertices in v by updating the relational variable y to $y + y;p;y$.

Lemma 4. *Let x, v, p be elements of a relation algebra in which Tarski's axiom holds, let p be rectangular, and let $y = \mathsf{rtc}(x;v);x$. Then*

$$\mathsf{rtc}(x;(v+p));x = y + y;p;y.$$

Proof. Using the idempotent semiring axioms, the refinement law from Proposition 3, and the equation from Lemma 2, we could automatically prove

```
rctangle(z) -> rtc(x;(v+z));x=rtc(x;v);x+(rtc(x;v);((x;z);rtc(x;v)));x.
```

in less than 15s. Note that Prover9 would not accept p as an implicitly universally quantified variable. Setting $y = \mathsf{rtc}(x;v);x$ then yields the result. □

We have thus formally justified the assignment `y:=y+y;p;y`, which is another key insight in Warshall's algorithm. We derived it from a general refinement law for reflexive transitive closures. Based on this formal development we can now prove explicitly and in declarative style the remaining proof obligation.

Theorem 5. *Executing the loop preserves the invariant if the guard of the loop is true.*

Proof. We attempted to prove the formula

```
wpoint(w) & inv(x,y,v) & y!=d(x) -> inv(x,y+y;(w;y),v+w).
```

from the axioms of Kleene algebras with domain, the definition of weak point, and the decomposition law from the proof of Lemma 4. However, Mace4 immediately found a three-element counterexample, so we needed to strengthen the assumptions. Adding the independent assumption `x;U=d(x);U`, which we automatically verified in relation algebras, yielded a short proof in less than 15s. □

A proof of Theorem 5 from the axioms of Kleene algebra and $x;U = d(x);U$ within reasonable time bounds did not succeed. Alternatively, we have found a less elegant and generic proof of Theorem 5 based on a decomposition of points instead of rectangles.

Partial Correctness. The result of the construction is summed up in the main theorem of this section.

Theorem 6. *The following variant of Warshall's transitive closure algorithm is (partially) correct:*

```
y,v:=x,0
while v!=d(x) do
  p:=point(v')
  y,v:=y+y;p;y,v+p od
```

The proof of this theorem has been fully automated in every detail, and the algorithm has been shown to be correct by construction. Kleene algebra alone did not suffice for this analysis. We used two additional independent assumptions that hold in relation algebras. One even required Tarski's axiom. Mace4 is instrumental for indicating when such assumptions are needed. Finding the right assumptions of course requires some background knowledge, but could be automated. A tool could blindly try combinations of previously verified relational lemmas from a given library.

To appreciate the complexity of proof search involved, it should be noted that most of the proofs involving transitive closures in this section are essentially inductive. With our algebraic axiomatisation of (reflexive) transitive closures of binary relations, these inductions could be captured calculationally in a first-order setting and therefore be automated.

In conclusion, the construction of Warshall's algorithm required some general familiarity with relations and some operational understanding of reflexive transitive closures. For the operational understanding, testing the loop invariant and developing the refinement law for reflexive transitive closures, experimenting with Mace4 and RelView was very helpful. But for the synthesis of the algorithm, no particular knowledge of the calculus of relation algebras or Kleene algebras was needed. All that could be hidden in the darkness of the interface.

6 Verification of Reachability Algorithms

The task of determining the set of states that are reachable from some given set of states in a digraph can also be reduced to relational reasoning. As a second case study, we show how two matrix-based algorithms can be automatically *verified.* In this scenario, the programmer annotates code with assertions for the precondition, postcondition and loop invariant. The code and the assertions are then tranformed into proof obligations from which program correctness is proved automatically in one full sweep behind the interface.

The RelView system provides a relational modelling language and an imperative programming language in which the programs and all assertions can be implemented and executed. We assume that RelView, Prover9 and Mace4 have been integrated into an imaginary tool that does all the translations between programs and tools, and runs the tools in the background.

A Naive Algorithm. The following relational algorithm computes the set w of states that are reachable in some digraph y from a set v of initial states; a previous manual construction can be found in [3]:

```
{pre(y,v) <-> x=x}
w:=v
while -(y^;w<=w) do
  {inv(y,v,w) <-> v<=w & w<=rtc(y^);v}
  w:=w+y^;w od
{post(y,v,w) <-> w=rtc(y^);v}
```

In this program, y is an adjacency matrix; v and w are vectors or other implementations of sets. But our imaginary verification tool ignores all types and selects Kleene algebra as the domain-specific theory after a syntactic analysis. It also ignores the operation of converse in $y^\wedge$ because y itself does not occur in the code. It therefore uniformly replaces $y^\wedge$ by x. Note that this step has little impact on ATP performance. The tool then passes the following postcondition, guard of the loop and invariant to Prover9; it ignores the trivial precondition:

```
guard(x,v,w) <-> -(x;w<=w).
post(x,v,w) <-> w=rtc(x);v.
inv(x,v,w) <-> v<=w & w<=rtc(x);v.
```

The postcondition says that upon termination the vector w stores all those vertices that are linked by an arrow in the reflexive transitive closure of x to a vertex in v. The idea of the program, to compute intermediate states w iteratively with respect to x such that after each iteration w is a superset of v and a subset of the set of reachable states, is captured by the invariant.

Proving partial correctness means discharging the following proof obligations, which our imaginary verification tool could automatically extract from the code and the assertions.

```
inv(x,v,w) & -guard(x,v,w) -> post(x,v,w).
inv(x,v,v).
inv(x,v,w) & guard(x,v,w) -> inv(x,v,w+x;w).
```

Using the axioms of Kleene algebra, Prover9 could instantaneously verify the first and the second proof obligation; the third one needed about ten seconds. In fact, the third proof obligation did not require the assumption that the guard of the loop is true. Termination has again been neglected. But obviously, the set w is enlarged in each iteration of the loop and finitely bounded by the guard.

A Refined Algorithm. The main drawback of the naive algorithm is that the guard of the loop is recomputed in each turn of the loop. Finite differencing yields a refined algorithm which uses a new vector or set variable u to store the intermediate values of $x; w \cdot w'$, where the complement w' of w is again taken at the set level (For a manual development, see again [3]). In this example, we do not use the precondition, postcondition and invariant in declarative style, but encode assertions directly in the relational modelling language.

```
{pre: true}
w,u:= v,v'*y^;v
while u!=0 do
  {inv: v<=w & w<=rtc(y^);v & u=w'*y^;w}
  w:=w+u
  u:=w'*y^;u od
{post: w=rtc(y^);v}
```

We now assume that our imaginary tool translates these assertions into a Kleene algebra with range (with dual domain axioms). The range operation is used behind the scene for typing sets and for computing sets of successor states: The set of states that are reachable (in one step) from a set v with respect to a relation x is given by the range of v;x. Now $a(x)$ denotes the antirange of x and $r(x)$ the range of x. If x is a set, then $x = r(x)$, $r(x) = a(a(x))$, and $a(r(x)) = a(x)$. Thus the verification tool can use $r(x)$ for "typing" that x is a set and generate the following formulas:

```
r(u)!=0.                                                    % guard
r(w)=r(r(v);rtc(x)).                                        % postcond.
r(v)<=r(w) & r(w)<=r(r(v);rtc(x)) & r(u)=a(w);r(r(w);x).  % invariant
```

Again, $y^\wedge$ is uniformly replaced by x, the trivial precondition is omitted. The tool would then generate the following proof obligations:

(1) The invariant is established by the initialisation.

```
r(v)<=r(v) & r(v)<=r(r(v);rtc(x)) & a(v);r(r(v);x)=a(v);r(r(v);x).
```

(2) The invariant establishes the postcondition if the guard of the loop is false.

```
r(v)<=r(w) & r(w)<=r(r(v);rtc(x)) & a(w);r(r(w);x)=0
                  -> r(r(v);rtc(x))<=r(w) & r(w)<=r(r(v);rtc(x)).
```

(3) Executing the loop preserves the invariant if the guard of the loop is true.

```
r(v)<=r(w) & r(w)<=r(r(v);rtc(x)) & r(u)=a(w);r(r(w);x)
                  -> r(v)<=r(w)+r(u) & r(w)+r(u)<=r(r(v);rtc(x)).
```

A proof of (1) took about 35s from the axioms of Kleene algebras with range and the definition of invariant. It yielded a proof with 42 steps. For the proof of (2), the postconditions has been split into two inequalities, and the (negated) guard of the loop has been ignored. A proof from the axioms alone failed within reasonable time; the following additional assumptions were required:

```
r(x+y)=r(x)+r(y)  &  r(v)+r(r(w);x)<=r(w) -> r(r(v);rtc(x))<=r(w).
```

Both formulas have already been verified [8]. The first assumption is additivity of range, which seems fundamental enough to be added to any domain-specific theory that uses range. The second additional assumption is based on the observation that the proof of the formula $r(w) \leq r(r(v); \mathsf{rtc}(x))$ from the third assumption, $r(w) \leq r(r(v); x)$, is essentially induction with respect to x. With these additional assumptions, Prover9 took about 150s; the proof has 102 steps. Proofs with fewer axioms failed within reasonable time. Whether a tool could automatically learn such additional assumptions is an interesting research question. The proof of (3) used the axioms of Kleene algebras with range and additivity of range. Prover9 took less than 130s and provided a proof with 149 steps.

The results of this section further demonstrate the flexibility of our approach. Here, two reachability algorithms, to which preconditions, postconditions and invariants have been added as assertions, have been automatically verified. Such correctness proofs could run in the background and complement existing techniques for extended static checking. Again, reasoning about reachability required induction, which could be fully automated in Kleene algebra.

7 Synthesis of Szpilrajn's Algorithm

Our final case study is again on program construction. To further demonstrate the versatility of our approach, we now use relation algebra with sets modelled as vectors as the domain-specific theory behind the interface. We synthesise Szpilrajn's algorithm [23] that computes the linear extension of a given partial order. This synthesis is inspired by a previous manual correctness proof [6].

Initial Specification. Consider the following program construction task:

> *Given a finite partial order x, find a program with variable y that stores the linear extension of x after execution.*

Again, we conjecture that our program is a simple while loop:

```
... y:=x ...
while ... do
  ... y:=? ... od
```

Obviously, the precondition is that the input relation x is a partial order, a reflexive, antisymmetric and transitive relation. In our relational modelling language we can write:

```
1<=x  &  x*x^<=1  &  x;x<=x.
```

But we could also provide more declarative concepts such as `ref(x)`, `antisym(x)` and `trans(x)`. The postcondition is that the relational variable y stores a total order relation that extends x:

```
1<=y & y*y^<=1 & y;y<=y & x<=y & y+y^=U.
```

The fourth inequality states that the partial order relation y extends x; the last one expresses totality of y.

Developing the Invariant. The basic idea of Szpilrajn's algorithm is to build a chain of partial extensions of the partial order x by iteratively adding atoms (single ordered pairs) z that are incomparable by x, and incrementally computing the partial order for these extensions. Algorithmic details will again be part of the development, but we can now state the invariant:

The relation y is a partial order that contains x.

In our relational modelling language, this can be formalised as

```
1<=y & y*y^<=1 & y;y<=y & x<=y
```

Initialisation and Guard. We assume that our algorithm has only one global relational variable, namely y, which is initialised as x. Moreover, the while loop should terminate when no further extension of x can be computed, that is, when $y + y^\wedge = U$. The guard of the loop should therefore be `y+y^!=U`. We can justify these choices by verifying the following proof obligations:

Theorem 7. *The invariant is established by the initialisation (if the precondition holds); it establishes the postcondition when the guard of the loop is false.*

Proof. Using the axioms for relation algebras, these trivial ATP exercises needed no time. The invariant after initialisation *is* the precondition; the conjunction of the invariant and the negated guard of the loop *is* the postcondition. □

Termination and Development of the Loop. Termination of the algorithm is obvious, since only finitely many atoms can be added. As before, this is not further formalised.

To synthesise the loop body, we must determine the assignments to atoms z and the relational variable y. Obviously, z can be an arbitrary atom from the complement of $y + y^\wedge$. The variable y should be incremented by a function in y and z, that is, `y:=y+f(y,z)`. So we postulate that our program be of the following form, where `atom` is a choice function that picks some atom from $(y + y^\wedge)'$:

```
y:=x
while y+y^!=U do
  z:=atom((y+y^)')
  y:=y+f(y,z) od
```

The choice function can be axiomatised by `watom(atom(x)) & atom(x)!=0`, analogously to `point`. It remains to synthesise f and to show that the resulting assignment preserves the invariant whenever the guard of the loop is true. This can again be based on experiments with RelView, Prover9 and Mace4. But, since atoms are rectangles, Proposition 3 can again be used:

$$\mathsf{rtc}(y + z) = \mathsf{rtc}(y) + \mathsf{rtc}(y); z; \mathsf{rtc}(y).$$

Now $\mathsf{rtc}(y) = y$ since y is reflexive and transitive, which can be checked by ATP. Hence one of Szpilrajn's key insights can again be derived from our refinement law: `y:=y+y;z;y`.

Theorem 8. *Executing the loop preserves the invariant (if the guard of the loop is true).*

Proof. We assume that before executing the loop y is a partial order that extends x. We must prove that after execution the same properties hold of the new value of y. We use Prover9 with the axioms of relation algebras and the definition of weak atoms, but do not need the guard condition.

- The new value of y is a reflexive extension of x (this proof required no time):

  ```
  1<=y & y*y^<=1 & y;y<=y & x<=y -> x+1<=y+y;(z;y).
  ```

- The new value of y is transitive.

  ```
  y;y<=y & watom(z) & z<=(y+y')^
           -> (y+y;(z;y));(y+y;(z;y))<=(y+y;(z;y)).
  ```

  We used the axioms for relation algebra without the relation-algebraic definition of $U = x + x'$, but needed four additional (verified) assumptions,

  ```
  x;(y+z)=x;y+x;z  &  x<=y -> z;x<=z;y  &  x<=y -> x;z<=y;z  &  x<=U.
  ```

  and the definition of a weak atom from Section 3. These assumptions are very natural and should perhaps be included in any assumption set for relation algebras. The proof then took about 200s and has 158 steps.
- The new value of y is antisymmetric.

  ```
  y;y<=y & y*y^<=1 & watom(z) & z<=(y+y^)'
                     -> (y+y;(z;y))*(y+y;(z;y))^<=1.
  ```

  We could prove this goal from the axioms of relation algebras, and the definition of weak atom alone. The proof needed about 410s and has 274 steps, which is very long compared to similar experiments. □

The fact that additional assumptions were needed for proving transitivity may seem disappointing, but we can do better: Using idempotent semirings with converse (cf. Section 3) yielded a fully automated proof with 111 steps in less than 10s. Injections, vectors, weak points and weak atoms were used as before.

Of course we cannot use this simpler domain-specific theory for proving antisymmetry since this property cannot be expressed in Kleene algebra. To appreciate the complexity of proof search involved it should be mentioned that the manual relation-algebraic proof of antisymmetry [6] is quite involved: It requires several lemmas and covers almost an entire page. Proving the lemmas themselves is rather tedious and heavily involves the Schröder and Dedekind rules. Since these rules are difficult to prove by ATP, it is rather surprising that our automated correctness proof succeeds, possibly via a different route.

Partial Correctness. The result of this construction can be summed up in the main theorem of this section.

Theorem 9. *The following variant of Szpilrajn's algorithm for computing linear extensions of partial orders is (partially) correct:*

```
y:=x;
while y+y^!=U do
  z:=atom((y+y^)')
  y:=y+y;z;y od
```

Choosing the right domain-specific theories, the entire program construction could be automated from the theory axioms alone, that is, without any additional assumptions. The granularity of proof would again allow a fully automatic post-hoc verification with ATP systems in the background.

8 Discussion

Our main technical contribution is the demonstration that an integration of ATP systems and domain-specific algebras supports automatic correctness proofs for imperative programs through verification or program construction. We believe that these results motivate a new approach, to program construction in particular, which could combine simple high level program development techniques with computer enhanced mathematics based on domain-specific algebras and powerful proof automation. This section sketches some research questions and speculates about tools in which this approach could be implemented.

Theory Engineering. Relational calculi are not only useful for verifying and constructing algorithms for graphs, ordered sets, or other structures like games, Petri nets and lattices, they also form the basis for popular software development methods such as Alloy, B or Z. In preliminary experiments we have shown that the basic calculus of binary relations, as presented in the textbooks by Maddux [18] and Schmidt and Ströhlein [21] or Abrial's B-Book [1], can be automated [13]. Similar experiment in computer enhanced mathematics with reducts of relation algebras, in particular variants of idempotent semirings and Kleene algebras, are equally positive. While fully automated proofs of simpler theorems are possible from the theory axioms alone, more difficult goals require selecting appropriate lemmas (or even deleting "prolific" but unnecessary axioms). This suggests the following research questions: How can we organise and manage theory-specific and problem-specific knowledge to obtain useful assumption sets for ATP systems? How can we learn or abduce specific assumptions that are needed for particular proofs?

Relation algebras and variants of Kleene algebras capture the control flow in imperative programs and provide semantics for various computing applications, but they are less appropriate for modelling data structure or data types, or quantitative aspects of computations. Correctness proofs involving numbers, lists or arrays require different background theories or decision procedures. While their integration into our approach seems straightforward, their combination with relation algebras or Kleene algebras need further investigation.

Program Construction Technology. The most significant future task is to build program construction and verification tools that support the development of programs from specifications.

First, this requires the design of suitable modelling languages similar to those of Alloy or RelView. Second, existing libraries must be linked into a coherent data base. Third, to reason about data structures and data types, our present tool set should be complemented by SMT solvers and other decision procedures. Fourth, to manage the program construction process, automated tools need to be combined with interactive theorem provers which can handle the proof obligations of the Dijkstra-Gries approach or Hoare logic through tactics. These tools can also be used for residual inductive proofs that cannot be automated or for splitting complex proof goals into subgoals. Fifth, mechanisms for selecting appropriate theories behind the interface must be developed. We believe that our approach can largely be based on existing technology that only needs to be balanced in suitable ways. While for relation-based algorithms, development tools could be built around the RelView system, more general tools could support any other relation-based software development method.

The ultimate goal of our approach is to turn program construction into an activity in which the intellectually demanding and creative engineering tasks are separated — at an appropriate level of granularity — from routine calculations, such that developers can focus on the conceptual side of the construction while delegating technicalities to automated tools. The resulting approach would not only support teaching formal program development in more lightweight ways; it might also substantially increase the automation of existing formal methods for software development.

9 Conclusion

We have introduced a new approach to program construction and verification that is based on computer enhanced mathematics through a combination of domain-specific algebras with ATP systems, model generators and computer algebra tools. Using this combination we could prove the correctness of some standard algorithms fully automatically within the Dijkstra-Gries approach. While these results seem an interesting contribution per se, we see them predominantly as first steps within a larger programme aiming at lightweight formal methods with heavyweight automation. Traditional program construction techniques could thereby be lifted to a new level of simplicity and applicability. It seems feasible to realise this programme through a collective activity within the Mathematics of Program Construction community in the near future.

Acknowledgement. We want to thank Walter Guttmann and the anonymous referees for valuable remarks. We are grateful to Demetris Kennes for his help with preparing the proofs at our web site.

References

1. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
2. Back, R.-J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, Heidelberg (1998)

3. Berghammer, R.: Combining Relational Calculus and the Dijkstra-Gries Method for Deriving Relational Programs. Information Sciences 119, 155–171 (1999)
4. Berghammer, R., Leoniuk, B., Milanese, U.: Implementation of Relation Algebra using Binary Decision Diagrams. In: de Swart, H. (ed.) RelMiCS 2001. LNCS, vol. 2561, pp. 241–257. Springer, Heidelberg (2002)
5. Berghammer, R., Neumann, F.: RelView – an OBDD-based computer algebra system for relations. In: Ganzha, V.G., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2005. LNCS, vol. 3718, pp. 40–51. Springer, Heidelberg (2005)
6. Berghammer, R.: Applying Relation Algebra and RelView to Solve Problems on Orders and Lattices. Acta Informatica 45, 211–236 (2008)
7. Cormen, T.H., Leiserson, C.E., Rivest, D.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge (2009)
8. Desharnais, J., Struth, G.: Modal Semirings Revisited. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 360–387. Springer, Heidelberg (2008)
9. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
10. Floyd, R.W.: Assigning Meanings to Programs. In: Proc. AMS Symposia on Applied Mathematics, vol. 19, pp. 19–31 (1967)
11. Gries, D.: The Science of Computer Programming. Springer, Heidelberg (1981)
12. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM 12(10), 576–580 (1969)
13. Höfner, P., Struth, G.: Algebraic Reasoning with Prover9 (2009), `www.dcs.shef.ac.uk/~georg/ka/`
14. Höfner, P., Struth, G.: Automated Reasoning in Kleene Algebra. In: Pfenning, P. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 279–294. Springer, Heidelberg (2007)
15. Höfner, P., Struth, G.: On Automating the Calculus of Relations. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 50–66. Springer, Heidelberg (2008)
16. Jackson, D.: Software Abstractions. The MIT Press, Cambridge (2006)
17. Kozen, D.: Kleene Algebra with Tests. ACM Trans. Program. Lang. Syst. 19(3), 427–443 (1997)
18. Maddux, R.D.: Relation Algebras. Elsevier, Amsterdam (2006)
19. McCune, W.: Prover9 and Mace4 (2007), `www.prover9.org`
20. Ng, J.: Relation Algebras with Transitive Closure. Ph.D. thesis, University of California, Berkeley (1984)
21. Schmidt, G., Ströhlein, T.: Relations and Graphs. Springer, Heidelberg (1993)
22. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall, Englewood Cliffs (2006)
23. Szpilrajn, E.: Sur l'extension de l'ordre partiel. Fundamenta Math. 16, 386–389 (1930)
24. Tarski, A.: On the Calculus of Relations. J. Symbolic Logic 6, 73–89 (1941)
25. Warshall, S.: A Theorem on Boolean Matrices. Journal of the ACM 9, 11–12 (1962)

The Logic of Large Enough

Eerke Boiten and Dan Grundy

Computing Laboratory, University of Kent
e.a.boiten@kent.ac.uk, daniel.c.grundy@gmail.com

Abstract. In this paper we explore the "for large enough" quantifier, also known as "all but finitely many", which plays a central role in asymptotic reasoning, as used for example in complexity theory and cryptography. We investigate calculational properties of this quantifier, and show their application in reasoning about limits of functions.

Keywords: Calculational methods; asymptotics; generalised quantifiers.

1 Introduction

In what follows we explore a variant of universal quantification, namely that a particular predicate holds for "large enough" natural numbers. This quantifier occurs naturally in many areas of mathematics that employ asymptotic reasoning, in particular in complexity theory and its applications. Unfortunately, it often occurs in an encoded form (requiring two quantifiers, and worse: two dummy variables), or is left implicit in the context, thereby obscuring which manipulations are permissible. Most striking perhaps, is its negated occurrence, "for infinitely many ...", which is often seen in proofs by contradiction.

In the next section we define the "new" quantifier (in terms of existential and universal quantification) and explore its calculational properties. We then show how the quantifier can be applied in the theory of limits of sequences, where, in particular, it allows us to avoid reference to sequence indices in the resulting theorems and proofs. Finally, we indicate how this work leads to a calculational theory of asymptotics, with applications to complexity theory and beyond.

2 Large Enough Quantifiers

The property that P holds for large enough values of x can be described using an existential-universal quantifier combination:

$$\langle \exists\, X :: \langle \forall\, x : x > X : P \rangle \rangle$$

Throughout this paper we assume that x and X are natural numbers. In that case, the above is sometimes known as the "almost-all" quantifier, as it requires P to hold for all but finitely many numbers. This quantifier has been studied in logic since at least the 1970s [1,2], and belongs to the class of "generalised", or "modal quantifiers", defined by Mostowski [3] in 1957, and studied in the 1990s by Alechina, Van Lambalgen, and Van Benthem [4,5]. However, their work concentrated on properties of the class in general, in particular expressiveness and decidability, rather than on practical calculation.

Taking the natural numbers as a timeline, a property holding for large enough numbers means it will hold continuously from a certain point onwards; in other words, eventually it will always hold. The notation used in the following definition has been designed to emphasise the modal view of this quantifier, and that it binds a particular variable:

$$\langle \Diamond\!\Box\, x :: P \rangle \quad \equiv \quad \langle \exists\, X :: \langle \forall\, x : x > X : P \rangle\rangle \tag{1}$$

where X does not occur free in P . Properties of this quantifier follow. The first collection of properties concerns situations where the quantifier can be eliminated.

First, we show that:

$$\langle \Diamond\!\Box\, x :: \mathbf{true} \rangle \quad \equiv \quad \mathbf{true} \tag{2}$$

Proof:

$\langle \Diamond\!\Box\, x :: \mathbf{true} \rangle$

$\equiv$ { definition of $\Diamond\!\Box$; i.e., (1) }

$\langle \exists\, X :: \langle \forall\, x : x > X : \mathbf{true} \rangle\rangle$

$\equiv$ { property of $\forall$ }

$\langle \exists\, X :: \mathbf{true} \rangle$

$\equiv$ { the range of quantification (natural numbers) is non-empty }

true

Similarly, we have:

$$\langle \Diamond\!\Box\, x :: \mathbf{false} \rangle \quad \equiv \quad \mathbf{false} \tag{3}$$

Proof:

$\langle \Diamond\!\Box\, x :: \mathbf{false} \rangle$

$\equiv$ { definition of $\Diamond\!\Box$ }

$\langle \exists\, X :: \langle \forall\, x : x > X : \mathbf{false} \rangle\rangle$

$\equiv$ { the range of $\forall$ quantification is non-empty:
there is no largest natural number }

$\langle \exists\, X :: \mathbf{false} \rangle$

$\equiv$ { property of $\exists$ }

false

We can generalise (2) and (3) as follows: if x does not occur free in P we have

$$\langle \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu x :: P\rangle \;\equiv\; P \tag{4}$$

The proof uses the two properties of natural numbers used for proofs of (2) and (3) :

$$\begin{array}{ll}
 & \langle \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu x :: P\rangle \\
\equiv & \{ \text{ definition of } \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu \} \\
 & \langle \exists\, X :: \langle \forall\, x : x > X : P\rangle\rangle \\
\equiv & \{ \text{ eliminate redundant quantifiers; non-empty ranges } \} \\
 & P
\end{array}$$

Provided x does not occur free in a real-valued expression E we have:

$$\langle \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu x :: x > E\rangle \;\equiv\; \mathbf{true} \tag{5}$$

Proof:

$$\begin{array}{ll}
 & \langle \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu x :: x > E\rangle \\
\equiv & \{ \text{ definition of } \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu \} \\
 & \langle \exists\, X :: \langle \forall\, x : x > X : x > E\rangle\rangle \\
\Leftarrow & \{ \text{ one-point rule, } \lceil E\rceil \text{ is the least integer } \geq E \} \\
 & \langle \forall\, x : x > \lceil E\rceil : x > E\rangle \\
\equiv & \{ \text{ predicate calculus } \} \\
 & \mathbf{true}
\end{array}$$

Next, we investigate properties of the quantifier in combination with standard operators and quantifiers. The following useful monotonicity property follows immediately from monotonicity of the standard quantifiers (with respect to the $\Rightarrow$ ordering):

$$\langle \forall\, x :: P \Rightarrow Q\rangle \quad\Rightarrow\quad (\langle \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu x :: P\rangle \;\Rightarrow\; \langle \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu x :: Q\rangle) \tag{6}$$

We can use (6) to prove various weakening and strengthening rules; for example:

$$\langle \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu x :: P\rangle \quad\Rightarrow\quad \langle \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu x :: P \,\vee\, Q\rangle \tag{7}$$

Similarly, we have:

$$\langle \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu x :: P \,\wedge\, Q\rangle \quad\Rightarrow\quad \langle \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu x :: P\rangle \tag{8}$$

Clearly other variations are possible.

We can use (7) to prove the following "almost" distributivity property:

$$\langle \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu x :: P\rangle \,\vee\, \langle \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu x :: Q\rangle \quad\Rightarrow\quad \langle \Diamond\mkern-12.5mu{\scriptscriptstyle\square}\mkern4mu x :: P \,\vee\, Q\rangle \tag{9}$$

Proof:

$$
\begin{array}{ll}
 & \langle \Diamond\!\!\!\!\!\square\, x :: P\rangle \;\vee\; \langle \Diamond\!\!\!\!\!\square\, x :: Q\rangle \\
\Rightarrow & \{\ (7)\ ,\ \text{twice}\ \} \\
 & \langle \Diamond\!\!\!\!\!\square\, x :: P \vee Q\rangle \;\vee\; \langle \Diamond\!\!\!\!\!\square\, x :: P \vee Q\rangle \\
\equiv & \{\ \text{idempotence of}\ \vee\ \} \\
 & \langle \Diamond\!\!\!\!\!\square\, x :: P \vee Q\rangle
\end{array}
$$

The opposite direction does not hold: replace P with $even.x$ and Q with $odd.x$, for example.

Intuitively we have

$$\langle \forall\, x :: P\rangle \quad \Rightarrow \quad \langle \Diamond\!\!\!\!\!\square\, x :: P\rangle \qquad , \tag{10}$$

but we can prove it without unfolding $\Diamond\!\!\!\!\!\square$ by virtue of (6) :

$$
\begin{array}{ll}
 & \langle \forall\, x :: P\rangle \\
\equiv & \{\ \text{left identify of}\ \Rightarrow\ ,\ \text{heading for an appeal to}\ (6)\ \} \\
 & \langle \forall\, x :: \mathbf{true} \Rightarrow P\rangle \\
\Rightarrow & \{\ (6)\ \text{with}\ P, Q := \mathbf{true}, P\ \} \\
 & \langle \Diamond\!\!\!\!\!\square\, x :: \mathbf{true}\rangle \;\Rightarrow\; \langle \Diamond\!\!\!\!\!\square\, x :: P\rangle \\
\equiv & \{\ (2)\ \} \\
 & \mathbf{true} \;\Rightarrow\; \langle \Diamond\!\!\!\!\!\square\, x :: P\rangle \\
\equiv & \{\ \text{left identity of}\ \Rightarrow\ \} \\
 & \langle \Diamond\!\!\!\!\!\square\, x :: P\rangle
\end{array}
$$

Next, we have the useful property that conjunction distributes over $\Diamond\!\!\!\!\!\square$:

$$\langle \Diamond\!\!\!\!\!\square\, x :: P\rangle \;\wedge\; \langle \Diamond\!\!\!\!\!\square\, x :: Q\rangle \quad \equiv \quad \langle \Diamond\!\!\!\!\!\square\, x :: P \;\wedge\; Q\rangle \tag{11}$$

The following proof is by mutual implication; first we prove that

$$\langle \Diamond\!\!\!\!\!\square\, x :: P\rangle \;\wedge\; \langle \Diamond\!\!\!\!\!\square\, x :: Q\rangle \quad \Rightarrow \quad \langle \Diamond\!\!\!\!\!\square\, x :: P \;\wedge\; Q\rangle \qquad .$$

If we assume the antecedent, then there exist witnesses X_0 and X_1 such that:

$$
\begin{array}{ll}
 & \langle \forall\, x : x > X_0 : P\rangle \;\wedge\; \langle \forall\, x : x > X_1 : Q\rangle \\
\Rightarrow & \{\ \text{arithmetic}\ \} \\
 & \langle \forall\, x : x > X_0 \uparrow X_1 :: P\rangle \;\wedge\; \langle \forall\, x : x > X_0 \uparrow X_1 : Q\rangle \\
\equiv & \{\ \text{distributivity}\ \}
\end{array}
$$

$$
\begin{array}{ll}
& \langle \forall\, x : x > X_0 \uparrow X_1 :: P \ \wedge\ Q \rangle \\
\Rightarrow & \{ \ \exists \ \text{ introduction, with } \ X := \ X_0 \uparrow X_1 \ \} \\
& \langle \exists\, X :: \langle \forall\, x : x > X : P \ \wedge\ Q \rangle \rangle \\
\equiv & \{ \text{ definition of } \ \Diamond\!\!\!\!\!\;\square \ \} \\
& \langle \Diamond\!\!\!\!\!\;\square\, x :: P \ \wedge\ Q \rangle
\end{array}
$$

The opposite direction, viz

$$\langle \Diamond\!\!\!\!\!\;\square\, x :: P \rangle \ \wedge\ \langle \Diamond\!\!\!\!\!\;\square\, x :: Q \rangle \quad \Leftarrow \quad \langle \Diamond\!\!\!\!\!\;\square\, x :: P \ \wedge\ Q \rangle \qquad ,$$

is easily proved by appealing to the idempotence of conjunction, and then weakening via (8) .

Remark. Properties (2) , (6) , and (11) , along with "dummy renaming", correspond to the "minimal logic" of generalised quantifiers described in [4]. **End of Remark.**

It should be clear that we can generalise (11) to an arbitrary, but finite number of conjuncts; that is, for any fixed, finite set F , we have:

$$\langle \forall\, i : i \in F : \langle \Diamond\!\!\!\!\!\;\square\, x :: P_i \rangle \rangle \quad \equiv \quad \langle \Diamond\!\!\!\!\!\;\square\, x :: \langle \forall\, i : i \in F : P_i \rangle \rangle \tag{12}$$

and as a consequence, for fixed, finite set F , we have:

$$\langle \forall\, y : y \in F : \langle \Diamond\!\!\!\!\!\;\square\, x :: P \rangle \rangle \quad \equiv \quad \langle \Diamond\!\!\!\!\!\;\square\, x :: \langle \forall\, y : y \in F : P \rangle \rangle \tag{13}$$

It is clear from the first part of the proof of (11) , that in the general case, finiteness is required to take the maximum over the X bounds of each conjunct. Since finiteness is only necessary in one direction, if we drop this requirement we retain the following, weaker form of (13) :

$$\langle \forall\, y :: \langle \Diamond\!\!\!\!\!\;\square\, x :: P \rangle \rangle \quad \Leftarrow \quad \langle \Diamond\!\!\!\!\!\;\square\, x :: \langle \forall\, y :: P \rangle \rangle \tag{14}$$

Proof:

$$
\begin{array}{ll}
& \langle \Diamond\!\!\!\!\!\;\square\, x :: \langle \forall\, y :: P \rangle \rangle \\
\equiv & \{ \text{ definition of } \ \Diamond\!\!\!\!\!\;\square \ \} \\
& \langle \exists\, X :: \langle \forall\, x : x > X : \langle \forall\, y :: P \rangle \rangle \rangle \\
\equiv & \{ \text{ nesting } \} \\
& \langle \exists\, X :: \langle \forall\, y :: \langle \forall\, x : x > X : P \rangle \rangle \rangle \\
\Rightarrow & \{ \ \exists \forall \Rightarrow \forall \exists \ \} \\
& \langle \forall\, y :: \langle \exists\, X :: \langle \forall\, x : x > X : P \rangle \rangle \rangle \\
\equiv & \{ \text{ definition of } \ \Diamond\!\!\!\!\!\;\square \ \} \\
& \langle \forall\, y :: \langle \Diamond\!\!\!\!\!\;\square\, x :: P \rangle \rangle
\end{array}
$$

As a counterexample for the reverse implication, consider $x \geq y$ for P .

There are several ways of generalising the definition of $\mathop{\Diamond\!\!\!\!\!\square}$ to vectors; we choose the following as it is insensitive to the ordering of dummy variables:

$$\langle \mathop{\Diamond\!\!\!\!\!\square} x, y :: P \rangle \quad \equiv \quad \langle \exists\, X, Y :: \langle \forall\, x, y : x > X \;\wedge\; y > Y : P \rangle \rangle \tag{15}$$

Equivalently, we have what we refer to as the "diagonal" property:

$$\langle \mathop{\Diamond\!\!\!\!\!\square} x, y :: P \rangle \quad \equiv \quad \langle \exists\, Z :: \langle \forall\, x, y : x > Z \;\wedge\; y > Z : P \rangle \rangle \tag{16}$$

We prove (16) by mutual implication. Assume X , Y , and Z are not free in P , then:

$$
\begin{array}{ll}
& \langle \mathop{\Diamond\!\!\!\!\!\square} x, y :: P \rangle \\
\equiv & \{ \text{ (15) } \} \\
& \langle \exists\, X, Y :: \langle \forall\, x, y :: x > X \;\wedge\; y > Y : P \rangle \rangle \\
\Rightarrow & \{ \; \exists \text{ introduction, with } Z := \; X \uparrow Y \; \} \\
& \langle \exists\, X, Y :: \langle \exists\, Z :: \langle \forall\, x, y : x > Z \;\wedge\; y > Z : P \rangle \rangle \rangle \\
\equiv & \{ \text{ eliminate redundant outer quantifiers } \} \\
& \langle \exists\, Z :: \langle \forall\, x, y : x > Z \;\wedge\; y > Z : P \rangle \rangle \\
\Rightarrow & \{ \; \exists \text{ introduction, with } X, Y := \; Z, Z \; \} \\
& \langle \exists\, Z :: \langle \exists\, X, Y :: \langle \forall\, x, y : x > X \;\wedge\; y > Y : P \rangle \rangle \rangle \\
\equiv & \{ \text{ eliminate redundant outer quantifier } \} \\
& \langle \exists\, X, Y :: \langle \forall\, x, y : x > X \;\wedge\; y > Y : P \rangle \rangle \\
\equiv & \{ \text{ (15) } \} \\
& \langle \mathop{\Diamond\!\!\!\!\!\square} x, y :: P \rangle
\end{array}
$$

We refer to the following distributivity property as "unvectoring". If x does not occur free in Q , and y does not occur free in P , then:

$$\langle \mathop{\Diamond\!\!\!\!\!\square} x, y :: P \;\wedge\; Q \rangle \quad \equiv \quad \langle \mathop{\Diamond\!\!\!\!\!\square} x :: P \rangle \;\wedge\; \langle \mathop{\Diamond\!\!\!\!\!\square} y :: Q \rangle \tag{17}$$

Proof:

$$
\begin{array}{ll}
& \langle \mathop{\Diamond\!\!\!\!\!\square} x, y :: P \;\wedge\; Q \rangle \\
\equiv & \{ \; \wedge \text{ over } \mathop{\Diamond\!\!\!\!\!\square} \text{ ; i.e., (11) } \} \\
& \langle \mathop{\Diamond\!\!\!\!\!\square} x, y :: P \rangle \;\wedge\; \langle \mathop{\Diamond\!\!\!\!\!\square} x, y :: Q \rangle \\
\equiv & \{ \text{ eliminate redundant quantifiers } \} \\
& \langle \mathop{\Diamond\!\!\!\!\!\square} x :: P \rangle \;\wedge\; \langle \mathop{\Diamond\!\!\!\!\!\square} y :: Q \rangle
\end{array}
$$

The generalised definition of $\Diamond\Box$ in (15) allows us to nest quantifications as follows:

$$\langle \Diamond\Box\, x, y :: P \rangle \quad \Rightarrow \quad \langle \Diamond\Box\, x :: \langle \Diamond\Box\, y :: P \rangle\rangle \tag{18}$$

Proof:

$$\begin{array}{ll}
 & \langle \Diamond\Box\, x, y :: P \rangle \\
\equiv & \{\ (15)\ \} \\
 & \langle \exists\, X, Y :: \langle \forall\, x, y : x > X \ \wedge\ y > Y : P \rangle\rangle \\
\equiv & \{\ \text{nesting, twice}\ \} \\
 & \langle \exists\, X :: \langle \exists\, Y :: \langle \forall\, x : x > X : \langle \forall\, y : y > Y : P \rangle\rangle\rangle\rangle \\
\Rightarrow & \{\ \exists\forall \Rightarrow \forall\exists\ \} \\
 & \langle \exists\, X :: \langle \forall\, x : x > X : \langle \exists\, Y : \langle \forall\, y : y > Y : P \rangle\rangle\rangle\rangle \\
\equiv & \{\ (1)\ ,\ \text{twice}\ \} \\
 & \langle \Diamond\Box\, x :: \langle \Diamond\Box\, y :: P \rangle\rangle
\end{array}$$

The reverse implication does not hold as $\exists$ and $\forall$ do not generally commute; for example, consider $y > x$ for P .

Next, we investigate circumstances where we can replace x by $f.x$ inside $\Diamond\Box$-expressions, for an "eventually increasing" function f . Specifically, let f be a function from natural numbers to reals that satisfies the following property:

$$\langle \forall\, y :: \langle \Diamond\Box\, x :: f.x > y \rangle\rangle \tag{19}$$

Informally, this states that $f.x$ will eventually remain above any bound. Most of the functions considered in computational complexity theory have this property, including, for example, positive polynomials, and their quotients where the numerator has a higher degree than the denominator (but excluding the constant 0); the identity function also satisfies it. For functions that satisfy (19) , by skolemising the existential quantification inside $\Diamond\Box$ we can introduce a function *bound* that satisfies the property:

$$\langle \forall\, x : x > \mathit{bound}.y : f.x > y \rangle$$

For functions that satisfy (19) , for Boolean function P we have:

$$\langle \Diamond\Box\, x :: P.x \rangle \quad \Rightarrow \quad \langle \Diamond\Box\, x :: P.(f.x) \rangle \tag{20}$$

If we assume the antecedent, then according to the definition of $\Diamond\Box$ we have $\langle \forall\, x : x > X : P.x \rangle$ for some X . Since f satisfies (19) we have:

$$\begin{array}{ll}
 & \langle \forall\, x : x > \mathit{bound}.X : f.x > X \rangle \\
\Rightarrow & \{\ \exists\ \text{introduction, with}\ \ Y :=\ \mathit{bound}.X\ \}
\end{array}$$

$$
\begin{array}{ll}
 & \langle \exists\, Y :: \langle \forall\, x : x > Y : f.x > X \rangle\rangle \\
\Rightarrow & \{ \text{ antecedent: } x > X \Rightarrow P.x \ \} \\
 & \langle \exists\, Y :: \langle \forall\, x : x > Y : P.(f.x) \rangle\rangle \\
\equiv & \{ \text{ definition of } {\Diamond\mkern-12mu{\scriptstyle\square}} \ \} \\
 & \langle {\Diamond\mkern-12mu{\scriptstyle\square}}\, x :: P.(f.x) \rangle
\end{array}
$$

For functions that satisfy (19) , "for large enough x" is equivalent to "for large enough $f.x$":

$$
\langle {\Diamond\mkern-12mu{\scriptstyle\square}}\, x :: P \rangle \quad \equiv \quad \langle \exists\, Y :: \langle \forall\, x :: f.x > Y : P \rangle\rangle \tag{21}
$$

We prove this by mutual implication. From left to right, (19) is not necessary. We observe that f has a maximal value on every prefix of $\mathbb{N}$; we denote this maximum $\nabla.X$, where

$$
\nabla.X \quad = \quad \langle \uparrow x : x \leq X : f.x \rangle
$$

It follows that if $f.x > \nabla.X$ then $x > X$. Now we calculate as follows:

$$
\begin{array}{ll}
 & \langle {\Diamond\mkern-12mu{\scriptstyle\square}}\, x :: P \rangle \\
\equiv & \{ \text{ definition of } {\Diamond\mkern-12mu{\scriptstyle\square}} \ \} \\
 & \langle \exists\, X :: \langle \forall\, x : x > X : P \rangle\rangle \\
\Rightarrow & \{ \text{ range strengthening: } f.x > \nabla.X \Rightarrow x > X \ \} \\
 & \langle \exists\, X :: \langle \forall\, x : f.x > \nabla.X : P \rangle\rangle \\
\Rightarrow & \{ \ \exists \text{ introduction, with } Y := \nabla.X \text{ ; eliminate redundant quantifier } \} \\
 & \langle \exists\, Y :: \langle \forall\, x : f.x > Y : P \rangle\rangle
\end{array}
$$

To prove the opposite direction we observe that as a consequence of (19) , for every bound Y , the set $\langle x : f.x \leq Y : x \rangle$ is finite, and has a maximum, which we denote $\nabla.Y$; that is:

$$
\nabla.Y \quad = \quad \langle \uparrow x : f.x \leq Y : x \rangle
$$

If follows that if $x > \nabla.Y$ then $f.x > Y$. Now we calculate as follows:

$$
\begin{array}{ll}
 & \langle \exists\, Y :: \langle \forall\, x : f.x > Y : P \rangle\rangle \\
\Rightarrow & \{ \text{ range strengthening: } x > \nabla.Y \Rightarrow f.x > Y \ \} \\
 & \langle \exists\, Y :: \langle \forall\, x : x > \nabla.Y : P \rangle\rangle \\
\Rightarrow & \{ \ \exists \text{ introduction, with } X := \nabla.Y \text{ ; eliminate redundant quantifier } \} \\
 & \langle \exists\, X :: \langle \forall\, x : x > X : P \rangle\rangle \\
\equiv & \{ \text{ definition of } {\Diamond\mkern-12mu{\scriptstyle\square}} \ \} \\
 & \langle {\Diamond\mkern-12mu{\scriptstyle\square}}\, x :: P \rangle
\end{array}
$$

The corresponding "existential" operator, denoted by $\boxed{\diamond}$, is defined as the dual of $\Diamond\!\!\!\!\!\;{\scriptstyle\Box}$:

$$\langle \boxed{\diamond}\, x :: P\rangle \quad\equiv\quad \neg\langle \Diamond\!\!\!\!\!\;{\scriptstyle\Box}\, x :: \neg P\rangle \tag{22}$$

Consequently,

$$\langle \boxed{\diamond}\, x :: P\rangle \quad\equiv\quad \langle\forall\, X :: \langle\exists\, x : x > X : P\rangle\rangle \qquad , \tag{23}$$

which can be paraphrased as "(by increasing x) always P eventually holds"; equivalently: "there are infinitely many values of x for which P holds", i.e.:

$$\langle \boxed{\diamond}\, x :: P\rangle \quad\equiv\quad \langle x : P : x\rangle \text{ is infinite} \tag{24}$$

The above definition of $\boxed{\diamond}$ rather naturally implies that the set $\langle x : P : x\rangle$ is infinite; in the other direction we have:

$$\begin{array}{ll}
 & \langle x : P : x\rangle \text{ is infinite} \\
\Rightarrow & \{ \text{ prefixes of } \mathbb{N} \text{ are finite } \} \\
 & \langle\forall\, X :: \langle x : P : x\rangle \not\subseteq \langle x : x \le X : x\rangle\rangle \\
\equiv & \{ \text{ definition of } \not\subseteq \} \\
 & \langle\forall\, X :: \langle\exists\, x :: P \ \wedge\ \neg(x \le X)\rangle\rangle \\
\equiv & \{ \text{ trading } \} \\
 & \langle\forall\, X :: \langle\exists\, x : x > X : P\rangle\rangle \\
\equiv & \{ \text{ (23) } \} \\
 & \langle \boxed{\diamond}\, x :: P\rangle
\end{array}$$

As a corollary, we have the property mentioned above, namely that $\Diamond\!\!\!\!\!\;{\scriptstyle\Box}$ denotes "all but finitely many":

$$\langle \Diamond\!\!\!\!\!\;{\scriptstyle\Box}\, x :: P\rangle \quad\equiv\quad \langle x : \neg P : x\rangle \text{ is finite} \tag{25}$$

3 Limits of Sequences

As a simple application of the $\Diamond\!\!\!\!\!\;{\scriptstyle\Box}$ quantifier, we reason about limits of sequences, which we define as follows:

$$\lim_{x\to\infty} f.x = a \quad\equiv\quad \langle\forall\,\epsilon : \epsilon > 0 : \langle \Diamond\!\!\!\!\!\;{\scriptstyle\Box}\, x :: |f.x - a| < \epsilon\rangle\rangle \tag{26}$$

where f is a function from naturals to reals, and ϵ and a are reals.

As a first example, we show that multiplication by a positive constant commutes with taking a limit. For $c > 0$, we have:

$$\begin{array}{ll}
& \lim\limits_{x\to\infty} f.x \;=\; a \\
\equiv & \{\ (26)\ \} \\
& \langle \forall\, \epsilon : \epsilon > 0 : \langle \Diamond\!\!\!\!\!\;\square\, x :: |f.x - a| < \epsilon \rangle\rangle \\
\equiv & \{\ \text{arithmetic}\ \} \\
& \langle \forall\, \epsilon : \epsilon > 0 : \langle \Diamond\!\!\!\!\!\;\square\, x :: |c \cdot f.x \;-\; c \cdot a| < c \cdot \epsilon \rangle\rangle \\
\equiv & \{\ \text{dummy translation:}\ \ \epsilon' :=\ c \cdot \epsilon\ \} \\
& \langle \forall\, \epsilon' : \epsilon' > 0 : \langle \Diamond\!\!\!\!\!\;\square\, x :: |c \cdot f.x \;-\; c \cdot a| < \epsilon' \rangle\rangle \\
\equiv & \{\ (26)\ \} \\
& \lim\limits_{x\to\infty} c \cdot f.x \;=\; c \cdot a
\end{array}$$

In the following proof that limits distribute over addition, we avoid reference to particular values of the function's arguments by appealing to the distributivity of $\Diamond\!\!\!\!\!\;\square$ over conjunction:

$$\begin{array}{ll}
& \lim\limits_{x\to\infty} f.x = a \ \wedge\ \lim\limits_{x\to\infty} g.x = b \\
\equiv & \{\ (26)\ ,\ \ \text{twice}\ \} \\
& \langle \forall\, \epsilon : \epsilon > 0 : \langle \Diamond\!\!\!\!\!\;\square\, x :: |f.x - a| < \epsilon \rangle\rangle \ \wedge\ \langle \forall\, \epsilon : \epsilon > 0 : \langle \Diamond\!\!\!\!\!\;\square\, x :: |g.x - b| < \epsilon \rangle\rangle \\
\equiv & \{\ \text{distributivity}\ \} \\
& \langle \forall\, \epsilon : \epsilon > 0 : \langle \Diamond\!\!\!\!\!\;\square\, x :: |f.x - a| < \epsilon \rangle \ \wedge\ \langle \Diamond\!\!\!\!\!\;\square\, x :: |g.x - b| < \epsilon \rangle\rangle \\
\equiv & \{\ \Diamond\!\!\!\!\!\;\square\ \ \text{over}\ \ \wedge\ \} \\
& \langle \forall\, \epsilon : \epsilon > 0 : \langle \Diamond\!\!\!\!\!\;\square\, x :: |f.x - a| < \epsilon \ \wedge\ |g.x - b| < \epsilon \rangle\rangle \\
\Rightarrow & \{\ \text{arithmetic}\ \} \\
& \langle \forall\, \epsilon : \epsilon > 0 : \langle \Diamond\!\!\!\!\!\;\square\, x :: |(f.x + g.x) - (a + b)| \;<\; 2 \cdot \epsilon \rangle\rangle \\
\equiv & \{\ \text{dummy translation:}\ \ \epsilon' :=\ \epsilon/2\ \} \\
& \langle \forall\, \epsilon' : \epsilon' > 0 : \langle \Diamond\!\!\!\!\!\;\square\, x :: |f.x - a| < \epsilon' \ \wedge\ |g.x - b| < \epsilon' \rangle\rangle \\
\equiv & \{\ (26)\ \} \\
& \lim\limits_{x\to\infty} (f.x + g.x) \;=\; a + b
\end{array}$$

Next, we prove that every converging sequence is bounded. First, we establish:

$$\begin{array}{ll}
& \lim\limits_{x\to\infty} f.x = a \\
\equiv & \{\ (26)\ \} \\
& \langle \forall\, \epsilon : \epsilon > 0 : \langle \Diamond\!\!\!\!\!\;\square\, x :: |f.x - a| < \epsilon \rangle\rangle \\
\Rightarrow & \{\ \text{instantiation, with}\ \ \epsilon :=\ 1\ \}
\end{array}$$

$$\langle \Diamond\!\Box\, x :: |f.x - a| < 1\rangle$$
$$\Rightarrow \quad \{\ \text{arithmetic}\ \}$$
$$\langle \Diamond\!\Box\, x :: f.x \ < \ a+1\rangle$$

Now we can prove boundedness of f :

$$\langle \exists\, b :: \langle \forall\, x :: f.x < b\rangle\rangle$$
$$\equiv \quad \{\ \text{range splitting on } \ f.x \ < \ a+1\ \}$$
$$\langle \exists\, b :: \langle \forall\, x : f.x \ < \ a+1 \ \vee \ f.x \ \geq \ a+1 : f.x < b\rangle\rangle$$
$$\Leftarrow \quad \{\ \text{predicate calculus}\ \}$$
$$\langle \exists\, b :: b \ \geq \ a+1 \ \wedge \ \langle \forall\, x : f.x \ \geq \ a+1 : f.x < b\rangle\rangle$$
$$\equiv \quad \{\ \text{above: } \langle \Diamond\!\Box\, x :: f.x \ < \ a+1\rangle \ \text{ so by } \ (25) \ \text{ the maximum of the complement exists}\ \}$$
$$\langle \exists\, b :: b \ \geq \ a+1 \ \wedge \ b \ > \ \langle \uparrow\, x : f.x \ \geq \ a+1 : f.x\rangle\rangle$$
$$\equiv \quad \{\ \text{one point rule: } \ b \ = \ 1 + (a \uparrow \lceil \langle \uparrow\, x : f.x \ \geq \ a+1 : f.x\rangle \rceil)\ \}$$
$$\mathbf{true}$$

Convergence of functions can also be characterised through the Cauchy criterion:

$$\langle \forall\, \epsilon : \epsilon > 0 : \langle \Diamond\!\Box\, x, y :: |f.x - f.y| < \epsilon\rangle\rangle \tag{27}$$

This follows from the existence of a limit, as proved below. (Note that the reverse implication relies on the function's codomain being a complete metric space.)

$$\langle \forall\, \epsilon : \epsilon > 0 : \langle \Diamond\!\Box\, x, y :: |f.x - f.y| < \epsilon\rangle\rangle$$
$$\Leftarrow \quad \{\ \text{arithmetic: } \ |x - c| < \epsilon \ \wedge \ |y - c| < \epsilon \quad \Rightarrow \quad |x - y| \ < \ 2 \cdot \epsilon\ \}$$
$$\langle \exists\, a :: \langle \forall\, \epsilon : \epsilon > 0 : \langle \Diamond\!\Box\, x, y :: |f.x - a| \ < \ \epsilon/2 \ \wedge \ |f.y - a| \ < \ \epsilon/2\rangle\rangle\rangle$$
$$\equiv \quad \{\ \text{dummy translation: } \ \epsilon' := \ 2 \cdot \epsilon\ \}$$
$$\langle \exists\, a :: \langle \forall\, \epsilon' : \epsilon' > 0 : \langle \Diamond\!\Box\, x, y :: |f.x - a| \ < \ \epsilon' \ \wedge \ |f.y - a| \ < \ \epsilon'\rangle\rangle\rangle$$
$$\equiv \quad \{\ \text{unvector, i.e., } \ (17)\ \}$$
$$\langle \exists\, a :: \langle \forall\, \epsilon' : \epsilon' > 0 : \langle \Diamond\!\Box\, x :: |f.x - a| \ < \ \epsilon'\rangle \ \wedge \ \langle \Diamond\!\Box\, y :: |f.y - a| \ < \ \epsilon'\rangle\rangle\rangle$$
$$\equiv \quad \{\ \text{dummy renaming, with } \ y := \ x\ ; \ \text{idempotence of } \ \wedge\ \}$$
$$\langle \exists\, a :: \langle \forall\, \epsilon' : \epsilon' > 0 : \langle \Diamond\!\Box\, x :: |f.x - a| \ < \ \epsilon'\rangle\rangle\rangle$$
$$\equiv \quad \{\ (26)\ \}$$
$$\langle \exists\, a :: \lim_{x \to \infty} f.x \ = \ a\rangle$$

Remark. The above proofs still require extensive reasoning about the dummy variable ϵ in the definition of limits. A reviewer pointed out that one way of avoiding this may be by defining limits in terms of "limit superior" and "limit inferior", viz.

$$\begin{aligned}
&\text{liminf}\, f \;=\; \langle \uparrow n :: \langle \downarrow m : m \geq n : f.n \rangle\rangle \\
&\text{limsup}\, f \;=\; \langle \downarrow n :: \langle \uparrow m : m \geq n : f.n \rangle\rangle \\
&\text{lim}_{x\rightarrow\infty} f.x \;=\; a \quad \equiv \quad (\text{liminf}\, f = a \;\wedge\; \text{limsup}\, f = a)
\end{aligned}$$

and that it may be useful to explore the connection between the $\Diamond\!\!\!\!\square$ quantifier and infima and suprema over tails of sequences as used above.
End of Remark.

4 Towards Calculational Asymptotics

Our exploration of the "for large enough" quantifier was originally motivated by its application in proofs in asymptotics, which occur commonly in complexity theory and cryptography. Typically, as in the definition of limits, two quantities occur as dummies in asymptotic characterisations: the point where the function value is "close enough", and how close it is. The $\Diamond\!\!\!\!\square$ quantifier eliminates the former, but not yet the latter (the ϵ in the limit definition). In this section we define relations between functions that address this issue.

In the rest of this section we overload constants to denote constant functions, where, in particular, variable x denotes the identity function, and we lift operators on numbers pointwise to operators on functions. Thus, $x+1$ in a position where a function is required denotes $\langle\lambda\, x :: x\rangle + \langle\lambda\, y :: 1\rangle \;=\; \langle\lambda\, x :: x+1\rangle$ as expected.

Two types of asymptotic comparisons between functions exist: comparing asymptotic behaviour (based on absolute differences), and comparing asymptotic growth (based on relative differences). In the former, we define a number of operators between functions as follows:

$$f \leftrightarrow g \quad\equiv\quad \langle\forall\, \epsilon : \epsilon > 0 : \langle \Diamond\!\!\!\!\square\, x :: |f.x - g.x| < \epsilon\rangle\rangle \tag{28}$$

$$f \lhd g \quad\equiv\quad \langle\forall\, \epsilon : \epsilon > 0 : \langle \Diamond\!\!\!\!\square\, x :: 0 \;\leq\; g.x - f.x \;<\; \epsilon\rangle\rangle \tag{29}$$

$$f \rhd g \quad\equiv\quad g \lhd f \tag{30}$$

Oberserve that the above remove the dummy ϵ , with the first generalising the constant in the definition of a limit to a function.

The relation $\leftrightarrow$ is reflexive, symmetric and transitive; the proof of transitivity has the same shape as that for the addition of limits in the previous section. Indeed, we have that

$$\lim_{x\rightarrow\infty} f.x = a \quad\equiv\quad f \leftrightarrow a \tag{31}$$

Using this notation we can encode a number of "asymptotic" relations from the well-known textbook "Concrete Mathematics" [7]. The strict ordering of functions by asymptotic growth is captured by the following definition:

$$f \prec g \quad \equiv \quad f/g \leftrightarrow 0 \tag{32}$$

It is easy to prove from this that $\prec$ is transitive and irreflexive. We also write $g \succ f$ for $f \prec g$.

Useful relations in the world of asymptotic growth can be defined as follows:

$$f \ll g \quad \equiv \quad \langle \exists\, C :: \langle \Diamond\!\!\!\!\!\;\square\; x :: |f.x| \leq C \cdot |g.x| \rangle\rangle \tag{33}$$

$$f \asymp g \quad \equiv \quad f \ll g \;\wedge\; g \ll f \tag{34}$$

$$f \sim g \quad \equiv \quad f/g \leftrightarrow 1 \tag{35}$$

It is easy to prove that the first is a preorder and that the other two are equivalence relations. Our definition of $\asymp$ differs from the one in [7] in that the latter uses a single quantification over C for both instances of $\ll$. However, the two definitions are equivalent, as the inner predicate in the definition of $\ll$ is upward closed in C , so in both cases we can choose the maximum of the two instances of C .

Some further properties of these relations are stated below:

$$f \leftrightarrow g \;\wedge\; \neg(f \leftrightarrow 0) \quad \Rightarrow \quad f \sim g \tag{36}$$

$$\sim \quad \subseteq \quad \asymp \tag{37}$$

$$\asymp \quad \subseteq \quad \ll \tag{38}$$

$$\asymp \fatsemi \prec \quad \subseteq \quad \prec \tag{39}$$

$$\prec \fatsemi \asymp \quad \subseteq \quad \prec \tag{40}$$

where $\fatsemi$ denotes forward function composition. The reverse of (36) does not hold, e.g. $x \sim x+1$ but not $x \leftrightarrow x+1$.

As an example, in [8], for the full verification of a proof in [6], a proof obligation was to show that, for a polynomial a of degree at least 2,

$$\langle \Diamond\!\!\!\!\!\;\square\; x :: (1 - \tfrac{x}{a.x})^{a.x} < 2^{-x} \rangle$$

Starting from a standard result, we derive for constant C :

$$(1 + \tfrac{C}{x})^{x} \leftrightarrow e^{C}$$

$\Rightarrow$ { (21) , $f.x :=\; a.x/x$ satisfies (19) }

$$(1 + \tfrac{C}{a.x/x})^{a.x/x} \leftrightarrow e^{C}$$

$\Rightarrow$ { (36) }

$$(1 + \tfrac{C}{a.x/x})^{a.x/x} \sim e^{C}$$

Also, we need a result based on continuity, which we state without further proof: if f is a curried two-argument function such that $f.x$ is continuous for large enough x , then

$$g \leftrightarrow h \quad \Rightarrow \quad f.x.(g.x) \leftrightarrow f.x.(h.x) \tag{41}$$

Then we calculate:

$$
\begin{array}{ll}
 & (1 - \frac{x}{a.x})^{-a.x} \\
= & \{ \text{ fractions } \} \\
 & (1 + \frac{(-1)}{a.x/x})^{-a.x} \\
= & \{ \text{ exponents } \} \\
 & ((1 + \frac{(-1)}{a.x/x})^{a.x/x})^{-x} \\
\sim & \{ \text{ above with } \ C := \ -1 \ , \ \text{ and } \ (41) \ \} \\
 & (e^{-1})^{-x} \\
= & \{ \text{ exponents } \} \\
 & e^{x} \\
\succ & \{ \ c^x \prec d^x \ \text{ for } \ 1 < c < d \ \} \\
 & 2^x
\end{array}
$$

Using properties (37) to (40) we conclude from this calculation that

$$(1 - \tfrac{x}{a.x})^{-a.x} \ \succ \ 2^x$$

and using the definition of $\prec$ thus also

$$\langle \Diamond\!\!\!\!\!\;\square\ x :: (1 - \tfrac{x}{a.x})^{a.x} < 2^{-x} \rangle$$

as required.

We can also express so-called "Big Oh" notation using these relations. As stated in [7], this notation is usually defined in a particular context, e.g., for all arguments to the function, or near a fixed argument value, or for "large enough" arguments. In keeping with our application area, we assume the last case here. Thus, for functions on natural numbers, we have[1]

$$f \in O(g) \quad \equiv \quad f \ll g \tag{42}$$

Because $\ll$ is used but not given a separate notation in [7], this observation is missing there. The consequence that Θ (the intersection of "Big Oh" with its converse) corresponds to $\asymp$ is included, and so is the link between $\prec$ and Landau's "little oh". An alternative characterisation we may use and explore further is finiteness of $\limsup \ |f/g|$.

[1] Despite all the good reasons cited in [7] for writing "$f(x) = O(g(x))$" etc ("tradition [...] tradition [...] tradition [...] for our purposes it's natural."), we can't bring ourselves to do so. We do stick with a more traditional way of denoting the application of the O function though.

We now consider a number of the well-known properties of O, and how they may be proved in this set-up. Since $\ll$ is a preorder we have:

$$f \in O(f) \tag{43}$$

$$f \in O(g) \ \wedge\ g \in O(h) \quad \Rightarrow \quad f \in O(h) \tag{44}$$

Properties relating O to arithmetic operators would generally require unfolding the definition of $\ll$, for example:

$$f \in O(g) \ \Rightarrow\ C \cdot f \in O(g) \quad \text{for constant } C \tag{45}$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \ \Rightarrow\ f_1 + f_2 \in O(|g_1| + |g_2|) \tag{46}$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \ \Rightarrow\ f_1 + f_2 \in O(g_1 \uparrow g_2) \quad \text{for positive } g_1, g_2 \tag{47}$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \ \Rightarrow\ f_1 \cdot f_2 \in O(g_1 \cdot g_2) \tag{48}$$

Finally, properties (36) to (38) allow all three kinds of asymptotic equivalence between f and g to be transformed into $f \in O(g)$.

5 Further Applications

The original motivation for this work was found in cryptography. The second author's PhD thesis [8] explores calculational approaches to proofs in cryptography, which, in addition to traditional correctness notions and logic, contain elements of probabilism, number theory, complexity theory, and —through the latter— asymptotics.

Algebraic and symbolic reasoning has always been common in number theory, and typical proofs in this area are calculational and elegant. However, typical proofs in modern cryptography contain quantifications over algorithms and polynomials, with some of the quantifiers left implicit, and all of them changing between existential and universal in every (possibly nested) proof by contradiction. For the particular proof explored in detail in [8] (a demonstration proof from [6]), the "large enough" quantifier helped in the housekeeping of quantifications "in the context" and their acceptable manipulations. Notations explored in the previous section helped to structure and clarify a lemma based on asymptotics.

In general, this paper makes a small contribution to making modern cryptographic proofs more structured and manageable, with the ultimate goal of correctness by construction in modern cryptography. Our work in this area continues in the context of the UK EPSRC-funded CryptoForma network of excellence [9].

Acknowledgements

We would like to thank our colleagues in the TCS group, especially Stefan Kahrs and Simon Thompson, for fruitful discussions, and the MPC reviewers for many interesting and relevant suggestions.

References

1. Adams, E.: The logic of 'almost all'. Journal of Philosophical Logic 3, 3–17 (1974)
2. Marker, D., Slaman, T.: Decidability of the natural numbers with the almost-all quantifier (2006), `http://arxiv.org/abs/math/0602415v1`
3. Mostowski, A.: On a generalization of quantifiers. Fundamenta Mathematicae 44, 12–36 (1957)
4. Alechina, N., van Lambalgen, M.: Correspondence and completeness for generalized quantifiers. Bulletin of the Interest Group in Pure and Applied Logic 3, 167–190 (1995)
5. Alechina, N., van Benthem, J.: Modal quantification over structured domains. In: de Rijke, M. (ed.) Advances in Intensional Logic, pp. 1–27. Kluwer, Dordrecht (1997)
6. Goldreich, O.: Foundations of Cryptography, Basic Tools, vol. 1. Cambridge University Press, Cambridge (2001)
7. Graham, R., Knuth, D., Patashnik, O.: Concrete Mathematics, 2nd edn. Addison-Wesley, Reading (1994)
8. Grundy, D.: Concepts and Calculation in Cryptography. PhD thesis, Computing Laboratory, University of Kent (2008), `http://www.cs.kent.ac.uk/~eab2/crypto/thesis.web.pdf`
9. EPSRC CryptoForma network, `http://www.cryptoforma.org.uk`

Dependently Typed Grammars

Kasper Brink[1], Stefan Holdermans[2], and Andres Löh[1]

[1] Department of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB, Utrecht, The Netherlands
{kjbrink,andres}@cs.uu.nl
[2] Vector Fabrics
Paradijslaan 28, 5611 KN Eindhoven, The Netherlands
stefan@vectorfabrics.com

Abstract. Parser combinators are a popular tool for designing parsers in functional programming languages. If such combinators generate an abstract representation of the grammar as an intermediate step, it becomes easier to perform analyses and transformations that can improve the behaviour of the resulting parser. Grammar transformations must satisfy a number of invariants. In particular, they have to preserve the semantics associated with the grammar. Using conventional type systems, these constraints cannot be expressed satisfactorily, but as we show in this article, dependent types are a natural fit. We present a framework for grammars and grammar transformations using Agda. We implement the left-corner transformation for left-recursion removal and prove a language-inclusion property as use cases.

Keywords: context-free grammars, grammar transformation, dependently typed programming.

1 Introduction

Parser combinators are a popular tool for designing parsers in functional programming languages. Classic combinator libraries [1–4] directly encode the semantics of the parsing process. The user of such a library builds a function that – when run – attempts to parse some input and produces a result on a successful parse.

Many of today's parser combinator libraries, however, try to compute much more than merely the result. The reasons are manifold, but most prominently efficiency and error reporting. The technique is to choose an abstract representation for the parser that enables computing additional information, such as lists of errors and their positions or lookahead tables, or to perform optimizations such as left-factoring automatically.

If one takes this approach to the extreme, one ends up with a combinator library that builds an abstract representation of the entire grammar first, coupled with the desired semantics. This representation still contains the complete information the user has specified, and is therefore most suitable for performing analyses and transformations. After the grammar has been transformed to satisfaction, the library can then interpret the grammar as a parser, again with a choice on which parsing algorithm to employ.

However, operating on the abstract representation is somewhat tricky. There are several invariants that such operations must satisfy. In particular, they have to preserve the semantics associated with the grammar. Using conventional type systems, writing transformations in even a type-correct way can therefore be difficult, and even if it succeeds, the underlying constraints can often not be expressed satisfactorily.

In this article, we present a framework for grammars and grammar transformations using the dependently typed language Agda [5, 6]. We show that dependent types are a natural fit for the kinds of constraints and invariants one wants to express when dealing with grammars. In our framework, grammars are explicitly parameterized over the sets of terminals and nonterminals. We can talk about the left- and right-hand sides of productions in the types of values, and express properties such as that a certain production does not have an empty right-hand side. Each production carries its semantics, and the shape of the production determines the type of the associated semantic function.

As a case study, we present the left-corner transform for removal of left recursion from a grammar. Contrary to most other presentations of this grammar transformation, we also show how the semantic functions are transformed. Furthermore, we explain how properties about the transformation can be proved if desired, and give a language-inclusion property as an example.

The paper concludes with a discussion of what we have achieved and what we envision for the future, and a treatment of related work.

The complete Agda code on which this paper is based is available for download [7].

2 Grammar Framework

An important aspect of our approach is that we do not encode a grammar merely as a set of production rules that can be applied to sequences of symbols. Instead, each grammar has an associated semantics, and the grammar and semantics are encoded together. For every production of the grammar there is a corresponding semantic function, which is applied during parsing when that production is recognized in order to compute a parse result. Naturally, the types of the semantic functions must be consistent with the way in which they are applied. Below, we describe how this is enforced in our framework.

2.1 Representing Grammars

We begin by describing how grammars and the associated semantic functions are represented. A number of parameters (such as the type of nonterminals and terminals) are fixed for the whole development. We therefore assume that all subsequent definitions in this section are part of an Agda parameterized module:

$$
\begin{aligned}
&\textbf{module}\ \mathit{Grammar}\ (\mathit{Terminal} : \mathit{Set})\ (\mathit{Nonterminal} : \mathit{Set})\\
&\qquad (\_\overset{?}{=}t\_ : \mathit{Decidable}\ \{\mathit{Terminal}\}\ \_{\equiv}\_)\\
&\qquad (\_\overset{?}{=}n\_ : \mathit{Decidable}\ \{\mathit{Nonterminal}\}\ \_{\equiv}\_)\\
&\qquad (\llbracket\_\rrbracket : \mathit{Nonterminal} \rightarrow \mathit{Set})\ \textbf{where}
\end{aligned}
$$

We require that both terminals and nonterminals come with decision procedures for equality.

We define a *Symbol* type, which is the union of the *Terminal* and *Nonterminal* types:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Symbol} : \mathit{Set}\ \textbf{where}\\
&\quad \mathit{st}\ \ : \mathit{Terminal} \rightarrow \mathit{Symbol}\\
&\quad \mathit{sn} : \mathit{Nonterminal} \rightarrow \mathit{Symbol}
\end{aligned}
$$

In the following, we will often need lists of symbols, terminals and nonterminals, so we define

$$
\begin{aligned}
\mathit{Symbols} &= \mathit{List}\ \mathit{Symbol}\\
\mathit{Terminals} &= \mathit{List}\ \mathit{Terminal}\\
\mathit{Nonterminals} &= \mathit{List}\ \mathit{Nonterminal}
\end{aligned}
$$

as abbreviations.

To ensure that the semantic functions of the grammar are consistently typed with respect to the productions, the module signature introduces the mapping $\llbracket\_\rrbracket$, which assigns a *semantic type* to each nonterminal. This is the type of values that are produced when parsing that nonterminal (i. e., parsing A results in values of type $\llbracket A \rrbracket$).

The types of the semantic functions are determined by the nonterminals in the corresponding productions, and the semantic mapping $\llbracket\_\rrbracket$. A semantic function for a production computes a value for the left-hand side (LHS) from the parse results of the right-hand side (RHS) nonterminals. Thus, its argument types are the RHS nonterminal types and its result type is the LHS nonterminal type. For example, the production $A \rightarrow a\,B\,b\,C\,c$ has a semantic function of type $\llbracket B \rrbracket \rightarrow \llbracket C \rrbracket \rightarrow \llbracket A \rrbracket$. The semantic type of a *production* is the type of its semantic function; we shall write the semantic type of a production $A \rightarrow \beta$ as $\llbracket \beta \parallel A \rrbracket$. This can be defined in Agda as follows:

$$
\begin{aligned}
&\llbracket\_\|\_\rrbracket : \mathit{Symbols} \rightarrow \mathit{Nonterminal} \rightarrow \mathit{Set}\\
&\llbracket\ []\qquad\ \ \parallel A \rrbracket = \llbracket A \rrbracket\\
&\llbracket\ \mathit{st}\ \_\ :: \beta \parallel A \rrbracket = \llbracket \beta \parallel A \rrbracket\\
&\llbracket\ \mathit{sn}\ B :: \beta \parallel A \rrbracket = \llbracket B \rrbracket \rightarrow \llbracket \beta \parallel A \rrbracket
\end{aligned}
$$

Each nonterminal B in the RHS adds an argument of type $\llbracket B \rrbracket$ to the semantic type, whereas terminal symbols in the RHS are ignored.[1]

[1] It would be possible to associate semantics also with terminals (for example, if all identifiers of a language are represented by a common terminal), but in order to keep the presentation simple, we do not consider this variation in this paper.

$$\begin{aligned} E &\rightarrow E\,B\,N \mid N \\ B &\rightarrow + \mid - \\ N &\rightarrow 0 \mid 1 \end{aligned}$$

Fig. 1. Example grammar

We can now define a datatype to represent a production:

$$\begin{array}{l} \textbf{data}\ \mathit{Production} : \mathit{Set}\ \textbf{where} \\ \quad \mathit{prod} : (A : \mathit{Nonterminal}) \rightarrow (\beta : \mathit{Symbols}) \rightarrow [\![\ \beta \parallel A\ ]\!] \rightarrow \mathit{Production} \end{array}$$

A production consists of an LHS nonterminal A, the RHS symbols β, and an associated semantic function of type $[\![\ \beta \parallel A\ ]\!]$. Agda's dependent type systems enables us to concisely specify how the type of the semantic function depends on the shape of the production.

As an example of the representation of grammars in our framework we consider the grammar shown in Figure 1. It is a small grammar fragment that derives (from the start symbol E) a language of arithmetic expressions involving the numbers 0 and 1, and left-associative binary operators + and −. We shall refer to this example grammar throughout the article.

For this grammar, we define a semantics that evaluates an arithmetic expression to the number it represents. The nonterminals E and N each evaluate to a natural number, and B evaluates to a binary operator on naturals. Given a suitable definition for the type *Nonterminal*, we can define the semantic mapping as:

$$\begin{array}{l} [\![\_]\!] : \mathit{Nonterminal} \rightarrow \mathit{Set} \\ [\![\ E\ ]\!] \ = \ \mathbb{N} \\ [\![\ N\ ]\!] \ = \ \mathbb{N} \\ [\![\ B\ ]\!] \ = \ \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \end{array}$$

By assigning these types to the nonterminals, we also fix the semantic types of the productions. Below, we show the semantic types for four productions from the example grammar:

$$\begin{array}{lll} E \rightarrow E\,B\,N & [\![\ E\ B\ N \parallel E\ ]\!] & = \ \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ E \rightarrow N & [\![\ N \parallel E\ ]\!] & = \ \mathbb{N} \rightarrow \mathbb{N} \\ B \rightarrow + & [\![\ + \parallel B\ ]\!] & = \ \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ N \rightarrow 1 & [\![\ \mathit{1} \parallel N\ ]\!] & = \ \mathbb{N} \end{array}$$

We can encode these productions in Agda using the *Production* datatype (with terminal symbols represented by the built-in character type). This encoding includes the desired semantic functions, which match the types shown above.

```
p₁ = prod E (sn E :: sn B :: sn N :: []) (λ x f y → f x y)
p₂ = prod E [sn N]                        id
p₃ = prod B [st '+']                      _+_
p₄ = prod N [st '1']                      1
```

2.2 Constraints on Productions

Except for the consistency of the semantic function with respect to the LHS and RHS nonterminals, the *Production* datatype imposes no constraints on the form of a production. In some cases we wish to specify at the type level that a production has a specific left-hand side. We shall need this information when constructing a parser from a list of productions in Section 2.5, in order to show that the parse results of the constructed parser are consistently typed. To constrain the LHS of a production, we define an indexed datatype *ProductionLHS*:

```
projlhs : Production → Nonterminal
projlhs (prod A _ _) = A

data ProductionLHS : Nonterminal → Set where
  prodlhs : (p : Production) → ProductionLHS (projlhs p)
```

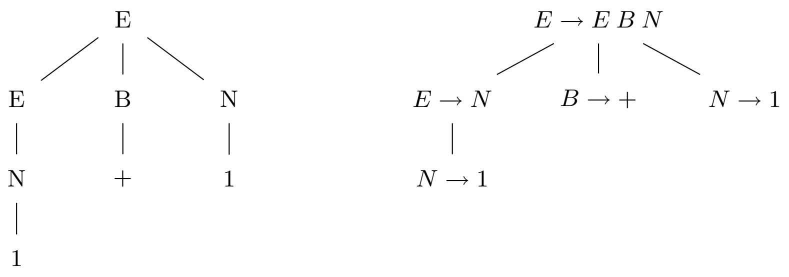
In Agda, a constructor of a datatype can restrict the index of the datatype: in this case, the wrapper constructor *prodlhs* wraps around an "ordinary" production *p* and uses the LHS of *p* as the index. In this way we expose information about the value of the production at the type level. We define a synonym for a list of productions with a specific LHS nonterminal:

```
ProductionsLHS : Nonterminal → Set
ProductionsLHS A = List (ProductionLHS A)
```

Such a list can be obtained by filtering an arbitrary list of productions with the function *filterLHS*:

```
filterLHS : (A : Nonterminal) → Productions → ProductionsLHS A
filterLHS _ [] = []
filterLHS A (prod B β sem :: ps) with A ≟n B
filterLHS A (prod .A β sem :: ps) | yes refl = prodlhs (prod A β sem) ::
                                               filterLHS A ps
filterLHS A (prod B β sem :: ps)  | no _     = filterLHS A ps
```

For a non-empty input list, the LHS of the first production, B, is compared to the specified nonterminal, A, in the **with**-clause. If the equality test $\_\overset{?}{=}n\_$ is successful, it returns a proof *refl* which is the single constructor of the equality type $\_\equiv\_$. Pattern matching on *refl* exposes this equality to the type system. In the *yes*-branch, we can therefore assume that B is equal to A (as expressed in Agda using the dot-pattern $.A$) and add the production to a list of type *ProductionLHS A*. In the *no*-branch, the production is discarded, and we continue with the rest of the list. The result type of *filterLHS* holds evidence of the filtering step that has been performed.



(a) Symbols in nodes (b) Productions in nodes

Fig. 2. Representations of parse trees

2.3 Parse Trees

To enable us to state and prove properties about grammars and their transformations we must define a representation for parse trees. In the conventional depiction of parse trees, internal nodes are labelled with nonterminals, and leaves with terminals. For a parse tree to be consistent with a grammar, an internal node A and its direct descendants $X_1, \ldots, X_n$ must form a production $A \to X_1 \cdots X_n$ of the grammar. In Figure 2a, we show the conventional representation of the parse tree for the sentence "$1 + 1$" in the example grammar.

In our case, it is more convenient to label the nodes of the parse tree with *productions*. This makes it easier to express constraints on the productions (e. g., that they belong to a certain grammar), and enables us to store the semantic functions in the parse tree, so that a parse result can be computed from it. The *root* of a parse tree shall refer to the LHS nonterminal of the production in the root node. Figure 2b shows how we represent the parse tree for the sentence "$1 + 1$" in our framework.

Our representation of parse trees introduces redundant information between a node and its children. To ensure that a parse tree is well-formed, we require that the nonterminals in the RHS of the production in each node correspond to the roots of its subtrees. This correspondence is encoded with the help of the relation $\_\!\sim\!\_$:

$$\begin{aligned} &\_\!\sim\!\_ : \mathit{Symbols} \to \mathit{Nonterminals} \to \mathit{Set} \\ &\beta \sim \mathit{ns} \;=\; \mathit{filterN}\ \beta \equiv \mathit{ns} \end{aligned}$$

where $\mathit{filterN} : \mathit{Symbols} \to \mathit{Nonterminals}$ filters the nonterminals out of a list of symbols.

We wish to enforce that the productions in a parse tree belong to a certain grammar. In some cases, the productions are subject to additional constraints, such as the requirement that the RHS is nonempty. The parse-tree datatype defined below is therefore parametrized over a predicate $Q : \mathit{Production} \to \mathit{Set}$, which represents the combined constraints that are satisfied by each production. The datatype is implemented with two mutually recursive definitions:

```
mutual
  data ParseTree (Q : Production → Set) : Set where
    node : (p : Production) → Q p → (cs : List (ParseTree Q)) →
              (projrhs p ∼ map projroot cs) → ParseTree Q
  projroot : ∀ { Q } → ParseTree Q → Nonterminal
  projroot (node (prod A _ _) _ _ _) = A
```

A *node* of a parse tree contains a *Production* (which includes a semantic function), a proof that it satisfies the predicate Q, a list of children, and a proof that the nonterminals on the RHS of the production match the roots of the child parse trees. The leaves of the tree contain productions with a RHS that consists entirely of terminals (or is empty).[2] In the function *projroot*, the type argument Q remains implicit, as indicated by the curly brackets. Agda will try to infer the argument whenever the function is called.

Given a parse tree, it is straight-forward to compute the sentence it represents:

```
merge : Symbols → List Terminals → Terminals
merge []           []         = []
merge (st b  :: β) us         = b :: merge β us
merge (sn B  :: β) (u :: us)  = u ++ merge β us
merge _            _          = []
sentence : ∀ { Q } → ParseTree Q → Terminals
sentence (node (prod _ β _) _ cs _) = merge β (map sentence cs)
```

The function *sentence* traverses the children of a node recursively. The helper function *merge* then concatenates the recursively computed subsequences with the terminals stored in the production of the node.

Another interesting computation over a parse tree is the semantic value that is represented by such a tree:

```
semantics : ∀ { Q } → (pt : ParseTree Q) → ⟦ projroot pt ⟧
```

The implementation performs a fold on the parse tree, building up the result starting from the leaves by applying all the semantic functions stored in the nodes. We omit the code for brevity.

2.4 Parser Combinators

To construct a parser for a grammar encoded in our representation, we make use of a parser combinator library. Parser combinators form a domain-specific embedded language for the implementation of parsers in functional languages such as Agda. The interface consists of several elementary parsers, and combinators that allow us to construct complex parsers out of simpler ones. This formalism

[2] The definition of *ParseTree* does not pass Agda's positivity checker. This is mainly a problem of the current checker, not a fundamental problem. We can circumvent the problem by rewriting our code slightly, but here, we opt for readability.

offers a convenient way to implement parsers using a notation that stays close to the underlying grammar. Here, we review one possible interface for parser combinators; their implementation is described elsewhere [2, 8].

The basic type *Parser A* denotes a parser that returns values of type A. We define the following elementary parsers:

$$\begin{aligned}
&\mathit{symbol} &&: \mathit{Terminal} \to \mathit{Parser}\ \mathit{Terminal}\\
&\mathit{succeed} &&: \forall\ \{A\} \to A \to \mathit{Parser}\ A\\
&\mathit{fail} &&: \forall\ \{A\} \to \mathit{Parser}\ A
\end{aligned}$$

The parser *symbol* recognizes a single terminal symbol and returns that symbol as a witness of a successful parse. The parser *succeed* recognizes the empty string (which always succeeds) and returns the supplied value of type A as the parse result. The parser *fail* always fails, so it never has to produce a value of the result type A.

The library contains the following elementary combinators:

$$\begin{aligned}
&\_{<}|{>}\_ &&: \forall\ \{A\} \to \mathit{Parser}\ A \to \mathit{Parser}\ A \to \mathit{Parser}\ A\\
&\_{<}{*}{>}\_ &&: \forall\ \{A\ B\} \to \mathit{Parser}\ (A \to B) \to \mathit{Parser}\ A \to \mathit{Parser}\ B
\end{aligned}$$

The combinator $\_{<}|{>}\_$ implements a choice between two parsers: the resulting parser recognizes either a sentence of the left parser or of the right parser. The result types of the parsers must be the same. The combinator $\_{<}{*}{>}\_$ implements sequential composition: the resulting parser recognises a sentence of the left parser followed by a sentence of the right parser. The parse results of the two parsers are combined by function application: the left parser produces a function, the argument type of which must match the result type of the right parser. We also define the derived combinator $\_{<}{*}\_$, which recognizes its arguments in sequence just like $\_{<}{*}{>}\_$, but which only returns the parse result of the left parser, discarding the result of the right.

2.5 Generating Parsers

The function *generateParser* constructs a parser for a grammar by mapping its productions onto the parser combinator interface, and it has type:

$$\mathit{generateParser} : \mathit{Productions} \to (S : \mathit{Nonterminal}) \to \mathit{Parser}\ [\![\ S\ ]\!]$$

It takes a list of productions of the grammar and a start nonterminal S, and constructs a parser for S that returns values of type $[\![\ S\ ]\!]$. The implementation of the function makes use of three mutually recursive subfunctions:

```
generateParser gram = gen where
  mutual
    gen : (A : Nonterminal) → Parser ⟦ A ⟧
    gen A = (foldr _<|>_ fail ∘ map genAlt ∘ filterLHS A) gram
    genAlt : ∀ {A} → ProductionLHS A → Parser ⟦ A ⟧
```

$$
\begin{array}{l}
\textit{genAlt}\ (\textit{prodlhs}\ (\textit{prod}\ A\ \beta\ \textit{sem})) \ = \ \textit{buildParser}\ \beta\ (\textit{succeed}\ \textit{sem}) \\
\textit{buildParser} : \forall\ \{A\}\ \beta \to \textit{Parser}\ [\![\ \beta \parallel A\ ]\!] \to \textit{Parser}\ [\![\ A\ ]\!] \\
\textit{buildParser}\ [\,] \qquad\qquad\quad p \ = \ p \\
\textit{buildParser}\ (\textit{st}\ \ b\ :: \beta)\ p \ = \ \textit{buildParser}\ \beta\ (p\ <\!\ast\ \textit{symbol}\ b) \\
\textit{buildParser}\ (\textit{sn}\ B :: \beta)\ p \ = \ \textit{buildParser}\ \beta\ (p\ <\!\ast\!>\ \textit{gen}\ B)
\end{array}
$$

The function *gen* takes a nonterminal A and generates a parser for A that returns values of type $[\![\ A\ ]\!]$. It first selects all productions with LHS A from the grammar using *filterLHS*. To each of these productions it applies *genAlt*, which generates a parser that corresponds to that particular alternative for A. The alternatives are combined into a single parser by folding with the parallel composition combinator $\_<|>\_$ (which has *fail* as a unit).

The function *genAlt* generates a parser for a single alternative; that is, the derivation always starts with the specified production. Note that the connection between the left-hand side nonterminal of the production and the result type of the parser is made in the type signature of *genAlt*. The semantic function *sem* is lifted to the trivial parser *succeed sem* with type *Parser* $[\![\ \beta \parallel A\ ]\!]$. The actual parser construction is performed by *buildParser*.

The function *buildParser* builds the parser by recursing over the right-hand side symbols β. The argument p is an accumulating parameter that is expanded into a parser that recognizes β. When β is empty, we return the constructed parser p, which has type *Parser* $[\![\ A\ ]\!]$. If β starts with a terminal b, we recognize it with *symbol* b, leaving the semantic types unchanged. If β starts with a nonterminal B, we generate a parser for B by calling *gen* recursively, and append this to p. Note that in this branch, p has type *Parser* $([\![\ B\ ]\!] \to [\![\ \beta \parallel A\ ]\!])$, and *gen* B has type *Parser* $[\![\ B\ ]\!]$, and the sequential composition removes the leftmost argument from the parser's result type.

3 Left-Corner Transform

The function *generateParser* from the preceding section does not pass Agda's termination checker. This is not surprising, since we do not impose any restrictions on the grammar at this point. In particular, a left-recursive grammar will lead to a non-terminating parser.

In this section, we discuss the left-corner transform (LCT), a grammar transformation that removes left recursion from a grammar [9, 10].

3.1 Transformation Rules

The transformation is presented below as a set of transformation rules that are applied to the productions of a grammar. If the grammar satisfies certain preconditions, applying the rules yields a transformed grammar that derives the same language and that does not contain left-recursive nonterminals. A nonterminal A is *left-recursive* if it derives a sequence of symbols beginning with A itself (i. e., $A \overset{*}{\Rightarrow} A\,\beta$). Such nonterminals can lead to non-termination with top-down parsers.

The LCT is based on manipulation of the left corners of the grammar. A symbol X is a *direct left corner* of the nonterminal A if there is a production $A \to X\beta$ in the grammar. The *left-corner relation* is the transitive closure of the direct left-corner relation.

The transformation extends the set of nonterminals with new nonterminals of the form A–X, where A and X are a nonterminal and a symbol of the original grammar, respectively. A new nonterminal A–X represents the part of an A that follows an X. For example, if $A \overset{*}{\Rightarrow} Bcd \overset{*}{\Rightarrow} abcd$, then $A\text{–}B \overset{*}{\Rightarrow} cd$ and $A\text{–}a \overset{*}{\Rightarrow} bcd$.

The three transformation rules for the left-corner transform, as formulated by Johnson [10], are as follows:

$$\begin{array}{lll} \forall A \in N,\ a \in T : & A \to a\, A\text{–}a & \in\ P' \end{array} \tag{1}$$

$$\begin{array}{lll} \forall C \in N,\ A \to X\beta \in P : & C\text{–}X \to \beta\, C\text{–}A & \in\ P' \end{array} \tag{2}$$

$$\begin{array}{lll} \forall A \in N : & A\text{–}A \to \epsilon & \in\ P' \end{array} \tag{3}$$

The rules are universally quantified over the terminals T, nonterminals N and productions P of the original grammar. The set P' contains the productions of the transformed grammar; the start symbol remains the same.

Some of the nonterminals and productions generated by these rules are useless: they can never occur in a complete derivation of a terminal string from the start symbol. There are other formulations of the LCT which avoid generating such useless productions; we have chosen this variant because its simplicity makes it easier to prove properties about the transformation (cf. Section 4).

3.2 Transforming Productions

Because rule (2) refers to the left corner of the input production, it is not defined for ϵ-productions. Therefore, we must encode the precondition that the transformation can only be applied to non-ϵ-productions. We begin by defining a predicate that identifies productions with a nonempty RHS:

```
isNonEpsilon : Production → Set
isNonEpsilon p = T ((not ∘ null ∘ projrhs) p)
```

The standard library function T maps boolean values to the corresponding propositions: the result is either the type $\top$ for truth with one inhabitant, or the type $\bot$ for falsity without inhabitants.

The type of non-ϵ-productions is a dependent pair of a production and a proof that its RHS is nonempty:

```
data NonEpsilonProduction : Set where
  nϵ : (p : Production) → {_ : isNonEpsilon p} →
       NonEpsilonProduction
```

We make the proof implicit,[3] since we typically only want to refer to it when dismissing the "impossible case" (empty RHS) in function definitions with a *NonEpsilonProduction* argument.

[3] In Agda, implicit arguments must always be named, hence the underscore in the type signature.

In the LCT, the transformed grammar uses a different set of nonterminals than the original grammar. To encode this in our implementation we require two separate instantiations of the *Grammar* module. We define modules O and T, and inside these modules we instantiate the *Grammar* module with the appropriate parameters. This enables us to refer to entities from either grammar with the prefixes "O" and "T".[4]

The set of nonterminals of the transformed grammar is derived from that of the original grammar by adding nonterminals of the form A–X. This is represented by the datatype *TNonterminal*:

```
data TNonterminal : Set where
  n     : ONonterminal → TNonterminal
  n_–_  : ONonterminal → OSymbol → TNonterminal
```

With this datatype, the original nonterminal A can be encoded in the transformed grammar as $n\ A$, and the new nonterminal A–B as $n\ A - O.sn\ B$.

The semantic types of the transformed nonterminals are a function of the original semantic types:

```
T⟦_⟧ : TNonterminal → Set
T⟦ n A ⟧             = ⟦ A ⟧
T⟦ n A – O.st b  ⟧   = ⟦ A ⟧
T⟦ n A – O.sn B ⟧    = ⟦ B ⟧ → ⟦ A ⟧
```

The semantic types of the original nonterminals are preserved in the transformed grammar. To explain the semantic type for a nonterminal A–X, consider the situation where we have recognized the left corner X, and continue by recognizing the remainder of A. If X was a terminal b, we must simply produce a value of type $[\![A]\!]$, but if X was a nonterminal B, we have already got a value of type $[\![B]\!]$, so to produce a result of type $[\![A]\!]$ we need a function $[\![B]\!] \to [\![A]\!]$.

With the representation of the two grammars in place, we now turn to the transformation itself. At the top level, we implement the universal quantification over symbols and productions in the transformation rules with a combination of *concat* and *map*.

```
lct : ONonEpsilonProductions → TProductions
lct ps = concatMap (λ A → map (rule1 A) ts) ns ++
         concatMap (λ C → map (rule2 C) ps) ns ++
         map rule3 ns
           where ts = terminals    ps
                 ns = nonterminals ps
```

The type of *lct* specifies the precondition that the productions of the original grammar must have a nonempty RHS. In the interface of the *Grammar* module, we do not require an operation to enumerate all symbols in the sets *Terminal*

[4] For readability, we also define synonyms such as $O[\![\_|\!|\_]\!] = O.[\![\_|\!|\_]\!]$.

and *Nonterminal*. Instead we use the functions *terminals* and *nonterminals* here, which traverse the list of productions, collect all terminal or nonterminal symbols encountered, and remove duplicates. This means that we only quantify over symbols that are actually used in the grammar.

The functions *rule1* and *rule3* directly encode the corresponding transformation rules (1) and (3) from page 67:

$$\begin{aligned}
&\mathit{rule1} : \mathit{ONonterminal} \;\rightarrow\; \mathit{Terminal} \;\rightarrow\; \mathit{TProduction}\\
&\mathit{rule1}\ A\ a \;=\; T.\mathit{prod}\ (n\ A)\ (T.\mathit{st}\ a :: [\,T.\mathit{sn}\ (n\ A - O.\mathit{st}\ a)])\ \mathit{id}\\
&\mathit{rule3} : \mathit{ONonterminal} \;\rightarrow\; \mathit{TProduction}\\
&\mathit{rule3}\ A \;=\; T.\mathit{prod}\ (n\ A - O.\mathit{sn}\ A)\ [\,]\ \mathit{id}
\end{aligned}$$

In both cases, the semantics of the constructed production has type $[\![A]\!] \rightarrow [\![A]\!]$, so the corresponding semantic function is the identity.

The function *rule2* is more interesting, since it is the only rule that actually transforms productions of the original grammar.

$$\begin{aligned}
&\mathit{rule2} : \mathit{ONonterminal} \;\rightarrow\; \mathit{ONonEpsilonProduction} \;\rightarrow\; \mathit{TProduction}\\
&\mathit{rule2}\ C\ (O.n\epsilon\ (O.\mathit{prod}\ A\ (X :: \beta)\ \mathit{sem})) \;=\\
&\qquad T.\mathit{prod}\ (n\ C - X)\ (\mathit{liftSymbols}\ \beta +\!\!+ [\,T.\mathit{sn}\ (n\ C - O.\mathit{sn}\ A)])\\
&\qquad\qquad (\mathit{semtrans}\ C\ A\ X\ \beta\ \mathit{sem})\\
&\mathit{rule2}\ \_\ \ (O.n\epsilon\ (O.\mathit{prod}\ \_\ [\,]\ \_)\ \{\,\})
\end{aligned}$$

Although the notation is slightly cluttered by the various symbol constructors, it is clear that this function performs a straightforward rearrangement of the input symbols. The function *liftSymbols* : *OSymbols* → *TSymbols* maps symbols of the original grammar to the same symbols in the transformed grammar (e. g., $O.\mathit{sn}\ A$ to $T.\mathit{sn}\ (n\ A)$). We will explain *semtrans* below. The second case is required to get the function through Agda's totality checker. We consider the case that the RHS of the production is empty, but can refute it due to the implicit proof of non-emptiness contained in the constructor $O.n\epsilon$, using Agda's notation for an absurd pattern { }.

3.3 Transforming Semantics

One problem still remains to be solved: when transforming a production with *rule2*, how should the associated semantic function be transformed? This task is performed by the function *semtrans*. We compute the semantic transformation incrementally, by folding over the symbols in the RHS of the production, and the type of the transformation depends on the symbols we fold over. For this, we use a dependently typed fold for a list of symbols, which is defined as part of the grammar framework:

$$\begin{aligned}
\mathit{foldSymbols} : \;&\{P : \mathit{Symbols} \rightarrow \mathit{Set}\} \rightarrow\\
&(\forall\ b\ \ \{\beta\} \rightarrow P\ \beta \rightarrow P\ (\mathit{st}\ b\ \ :: \beta)) \rightarrow\\
&(\forall\ B\ \{\beta\} \rightarrow P\ \beta \rightarrow P\ (\mathit{sn}\ B :: \beta)) \rightarrow\\
&P\ [\,] \rightarrow
\end{aligned}$$

$$\begin{aligned}
&\qquad\qquad (\beta : \mathit{Symbols}) \to P\ \beta\\
&\mathit{foldSymbols}\ \mathit{ft}\ \mathit{fn}\ \mathit{fe}\ (\mathit{st}\ b\ \ :: \beta) = \mathit{ft}\ b\ \ (\mathit{foldSymbols}\ \mathit{ft}\ \mathit{fn}\ \mathit{fe}\ \beta)\\
&\mathit{foldSymbols}\ \mathit{ft}\ \mathit{fn}\ \mathit{fe}\ (\mathit{sn}\ B :: \beta) = \mathit{fn}\ B\ (\mathit{foldSymbols}\ \mathit{ft}\ \mathit{fn}\ \mathit{fe}\ \beta)\\
&\mathit{foldSymbols}\ \_\ \_\ \ \mathit{fe}\ [\,] \qquad\qquad\ = \mathit{fe}
\end{aligned}$$

This is a dependently typed generalization of the ordinary *foldr* for lists; in addition, we also make a distinction between terminal and nonterminal symbols at the head of the list. The type P is the result type of the fold, which depends on the symbols folded over.

To define the transformation of the semantic functions, we use the semantic *types* as a guide. A production $A \to B\beta$ is transformed as follows by rule (2):

$$A \to B\beta \quad\longrightarrow\quad C\text{–}B \to \beta\, C\text{–}A$$

The types of the semantic functions must be transformed accordingly:

$$[\![B\beta \| A]\!] \quad\longrightarrow\quad [\![\beta C\text{–}A \| C\text{–}B]\!]$$

This can be viewed as a function type mapping the original semantic function to the transformed semantic function; in other words, it is the type of the semantic transformation. We encode this type in Agda as:

$$\begin{aligned}
\mathit{semtransN} : &\ \forall\ C\ A\ B\ \beta \to\\
&\ O[\![\ O.\mathit{sn}\ B :: \beta \ \|\ A\ ]\!] \to\\
&\ T[\![\ \mathit{liftSymbols}\ \beta +\!\!+ [\,T.\mathit{sn}\ (n\ C - O.\mathit{sn}\ A)] \ \|\ n\ C - O.\mathit{sn}\ B\ ]\!]
\end{aligned}$$

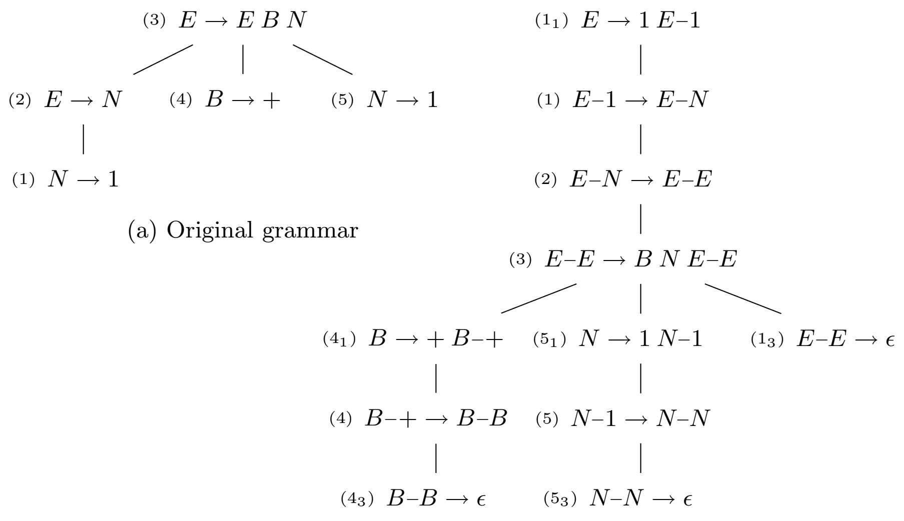
The implementation of *semtransN* is obtained by constructing a function that satisfies the given type:

$$\begin{aligned}
\mathit{semtransN}\ \_\ \_\ \_ = O.\mathit{foldSymbols}\ &(\lambda\ c\ \ f \to f)\\
&(\lambda\ C\ f \to \lambda\ g \to f \circ \mathit{flip}\ g)\\
&(\lambda\, f\ \ g \to g \circ f)
\end{aligned}$$

The suffix N to *semtransN* signifies that this transformation applies to a production with a *nonterminal* left corner; the function *semtransT* for a terminal left corner is analogous, with slightly different arguments to the fold. The function *semtrans* that is used in the definition of *rule2* makes the choice between the two based on the left corner of the input production.

4 Proof of a Language-Inclusion Property

Dependent types can be used to develop correctness proofs for our programs, without resorting to external proof tools. We illustrate this by proving a language-inclusion property for the LCT. This property forms part of a correctness proof for our implementation of the transformation; a full proof of correctness would also establish the converse property (thereby proving language preservation for the LCT), and the absence of left-recursion in the transformed grammar.


(3) $E \to E\,B\,N$
(2) $E \to N$
(4) $B \to +$
(5) $N \to 1$
(1) $N \to 1$
(1_1) $E \to 1\,E\text{–}1$
(1) $E\text{–}1 \to E\text{–}N$
(2) $E\text{–}N \to E\text{–}E$
(3) $E\text{–}E \to B\,N\,E\text{–}E$
(4_1) $B \to +\,B\text{–}+$
(5_1) $N \to 1\,N\text{–}1$
(1_3) $E\text{–}E \to \epsilon$
(4) $B\text{–}+ \to B\text{–}B$
(5) $N\text{–}1 \to N\text{–}N$
(4_3) $B\text{–}B \to \epsilon$
(5_3) $N\text{–}N \to \epsilon$

(a) Original grammar

(b) Transformed grammar

Fig. 3. Original and transformed parse trees

4.1 Language Inclusion

When applying grammar transformations, we usually require that they preserve the language that is generated by the grammar. In this section, we show how to prove that our implementation of the left-corner transform satisfies a *language-inclusion* property,

$$L(G) \subseteq L(G') \;, \tag{4}$$

which states that the language generated by the transformed grammar G' includes at least the language of the original grammar G.

The left-corner transform operates on the productions of a grammar by application of the transformation rules (1)–(3). The transformation of the productions leads to a corresponding transformation of the parse trees of the grammar. A parse tree is essentially a proof that the derived sentence is in the language of the grammar. To prove property (4), we must show that for every sentence w in the language of G (as evidenced by a parse tree using the productions of G), we can construct a parse tree using the productions of G'. By implementing the parse-tree transformation function we give a constructive proof of the language-inclusion property.

4.2 Relating Parse-Tree Transformation to Grammar Transformation

To illustrate the parse-tree transformation, Figure 3a shows the parse tree for the sentence “1 + 1” in the example grammar, and Figure 3b shows the parse tree for the same sentence in the transformed grammar. Johnson [10] notes that

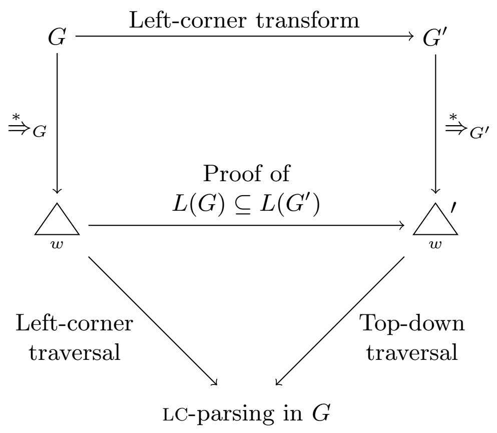


Fig. 4. Relationship between grammar transformation and parse-tree transformation

the LCT emulates a particular parsing strategy called *left-corner* (LC) parsing, in the sense that a top-down parser using the transformed grammar behaves identically to an LC-parser with the original grammar. Left-corner parsing contains aspects of both top-down and bottom-up parsing. We can characterize the parsing strategies as follows: in top-down parsing, productions are recognized before their children and their right siblings; in bottom-up parsing, productions are recognized after their children and their left siblings; and in left-corner parsing, productions are recognized after their left corner, but before their other children, and before their right siblings. Thus, the parse-tree transformation induced by the LCT satisfies the following property: for two parse trees related by the parse-tree transformation, an LC-traversal of the original tree corresponds to a top-down traversal of the transformed tree.

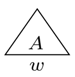
In Figure 3, the nodes of the original tree have been labelled in LC-order, and those of the transformed tree in top-down order. Each node of the transformed tree that is labelled with a plain number (without a subscript) is derived from the node in the original tree with the same label, by application of the LCT transformation rule (2). Note that the *left-hand side nonterminal* of an original node is reflected in the *right corner* of a transformed node. The transformed tree also contains nodes that are generated by LCT transformation rules (1) and (3), indicated by the subscripts on the labels. These nodes occur at the root and the lower right corner of all subtrees that do not correspond to a left corner in the original tree.

The relationships between grammars, parse trees and traversals are depicted schematically in Figure 4. On the top row, we see the original and transformed grammars, related by the left-corner transform. The middle row shows parse trees for the sentence w in the original and transformed grammar. These trees are related by the parse-tree transformation function, which is also the proof of the language-inclusion property (4). On the bottom row, we see that an LC-traversal of the original parse tree, which corresponds to LC-parsing in G, recognizes productions in the same order as a top-down traversal of the transformed parse tree.

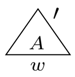
4.3 Parse-Tree Transformation: Specification

The parse-tree transformation function performs an LC-traversal of the original parse tree, transforming each original production with rule (2), and adding productions to the transformed tree in top-down order. Each subtree of the original tree that does not correspond to a left-corner is a new *goal* for the transformation. At the root and the lower right corner of these subtrees, productions are added that are generated by rules (1) and (3), respectively. Finally, we must show that the transformation of a subtree preserves the derived sentence.

We now give a precise definition of the parse-tree transformation. This consists of two mutually recursive functions $\mathcal{G}$ and $\mathcal{T}$. In the description of these functions, we use the following notation to concisely represent parse trees with root A, deriving the sentence w:

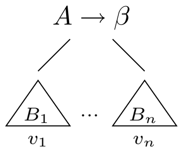


(original parse tree)



(transformed parse tree)

Note that we use this notation both to represent the *set* of parse trees with root A and sentence w (in the types of $\mathcal{T}$ and $\mathcal{G}$), and an *inhabitant* of that set (in the definitions of $\mathcal{T}$ and $\mathcal{G}$). A parse tree with a specific production $A \to \beta$ in the root node is written as:



(where $B_1, \ldots, B_n$ are the nonterminals of β)

The transformation functions are defined in Figure 5. Free variables in the type signatures, such as A and w, are universally quantified. The function $\mathcal{G}$ is the top-level transformation function, which is applied to each new goal. The type of this function specifies that the sentence of the original tree is preserved by the transformation.

The recursive transformation of the subtrees is performed by $\mathcal{T}$, which takes as arguments the current goal nonterminal C, the subtree to be transformed, and an accumulating parameter, which holds the lower right corner of the tree that is being constructed. This function satisfies the invariant that the tail v of the original sentence, concatenated with the sentence w of the tree being constructed, is the sentence of the result.

As can be seen from Figure 5, the transformation functions satisfy certain invariants related to roots of parse trees and the sentence derived by them. This is expressed in the types of $\mathcal{T}$ and $\mathcal{G}$ by referring not to arbitrary sets of parse trees, but to sets of parse trees that depend on a particular nonterminal for the root of the tree and a particular string of terminals for the sentence of the tree. Thus, the transformation functions are naturally dependently typed.

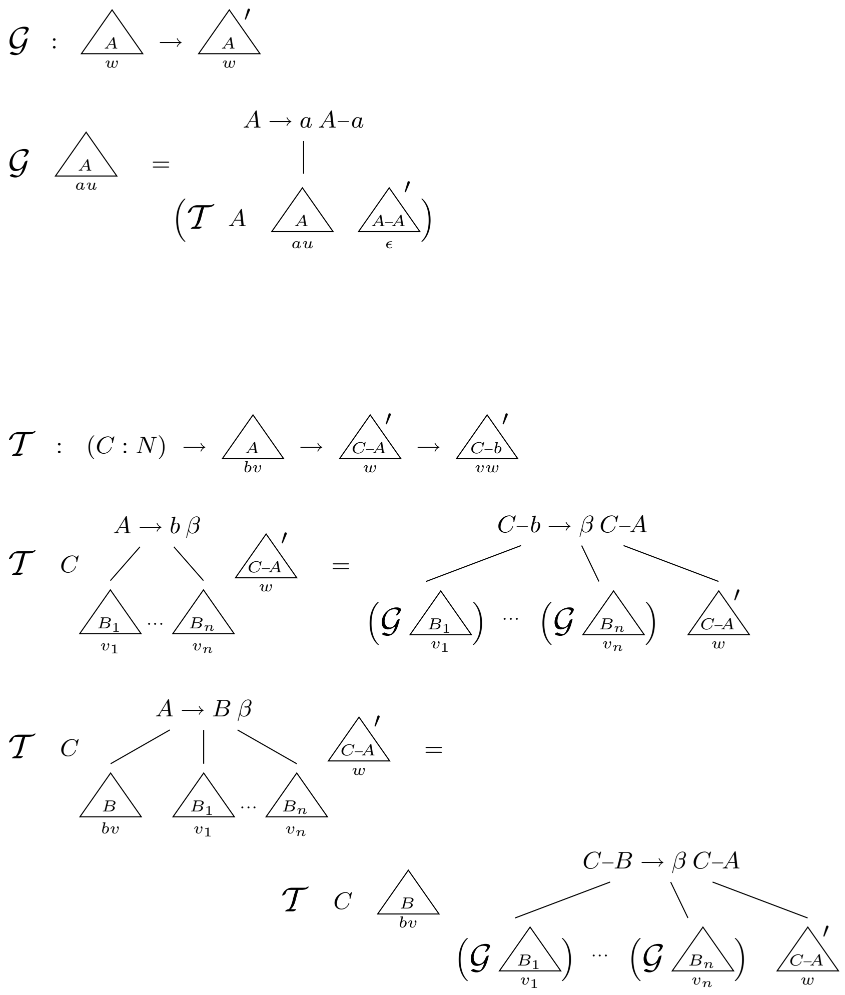

$\mathcal{G}$: A w → A' w
$\mathcal{G}$ A au = $A \to a\, A{-}a$ ($\mathcal{T}$ A A au $A{-}A$' ϵ)
$\mathcal{T}$: $(C : N)$ → A bv → $C{-}A$' w → $C{-}b$' vw
$\mathcal{T}$ C $A \to b\,\beta$ B_1 v_1 ⋯ B_n v_n $C{-}A$' w = $C{-}b \to \beta\, C{-}A$ ($\mathcal{G}$ B_1 v_1) ⋯ ($\mathcal{G}$ B_n v_n) $C{-}A$' w
$\mathcal{T}$ C $A \to B\,\beta$ B bv B_1 v_1 ⋯ B_n v_n $C{-}A$' w =
$\mathcal{T}$ C B bv $C{-}B \to \beta\, C{-}A$ ($\mathcal{G}$ B_1 v_1) ⋯ ($\mathcal{G}$ B_n v_n) $C{-}A$' w

Fig. 5. Parse-tree transformation functions

4.4 Parse-Tree Transformation: Agda Implementation

We now turn to the Agda implementation of the transformation functions that we defined in pseudocode in Figure 5. Our first task is to create a representation of the parse-tree types used in the specification. We begin by defining the synonym *OGrammar*:

$$OGrammar \;=\; ONonEpsilonProductions$$

The original grammar is given as a list of productions, which are guaranteed to be non-ϵ. We now define the general types of parse trees of the original and the transformed grammar, that is, types that do not specify the root and sentence of their inhabitants. To use the parse-tree type of Section 2.3, we must supply it with a predicate that describes the constraints that apply to each production in the parse tree. For original parse trees, we require that the production is a non-ϵ production, and that it is contained in the original grammar:

$$\begin{aligned}
&\mathit{OParseTree} : \mathit{OGrammar} \to \mathit{Set}\\
&\mathit{OParseTree}\ G\ =\ \mathit{O.ParseTree}\ (\lambda\ p \to \Sigma\ (\mathit{O.isNonEpsilon}\ p)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda\ \mathit{pn}\epsilon \to \mathit{O.n}\epsilon\ p\ \{\mathit{pn}\epsilon\} \in G))
\end{aligned}$$

Note that the parse tree itself contains "plain" productions; to express the requirement that they are contained in the grammar, we must combine them with their non-ϵ proofs to construct values of type *NonEpsilonProduction*.

For transformed parse trees, we only require that the productions are contained in the left-corner transform of the original grammar.

$$\begin{aligned}
&\mathit{TParseTree} : \mathit{OGrammar} \to \mathit{Set}\\
&\mathit{TParseTree}\ G\ =\ \mathit{T.ParseTree}\ (\lambda\ p \to p \in \mathit{lct}\ G)
\end{aligned}$$

From the general parse-tree types *OParseTree* and *TParseTree* we can create the specific parse-tree types that are used in the types of the transformation functions. This is done by taking the dependent pair of a parse tree with a pair of proofs about its root and sentence. For original parse trees we define:

$$\begin{aligned}
&\mathit{OPT} : \mathit{OGrammar} \to \mathit{ONonterminal} \to \mathit{Terminals} \to \mathit{Set}\\
&\mathit{OPT}\ G\ A\ w\ =\ \Sigma\ (\mathit{OParseTree}\ G)\ (\lambda\ \mathit{opt} \to \mathit{O.projroot}\ \ \mathit{opt} \equiv A\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \times\ \mathit{O.sentence}\ \mathit{opt} \equiv w)
\end{aligned}$$

The type $\mathit{OPT}\ \ G\ \ A\ \ w$ is the Agda representation of the type $\underset{w}{\triangle\!\!\!\!\!\!\!{}^{A}}$. The type *TPT* is defined in the same way.

Using the types *OPT* and *TPT*, it is straightforward to translate the types of the transformation functions into Agda. For $\mathcal{G}$ we get:

$$\mathit{transG} : \forall\ \{\ G\ A\ w\} \to \mathit{OPT}\ \ G\ \ A\ \ w \to \mathit{TPT}\ \ G\ (n\ \ A)\ \ w$$

And the type of $\mathcal{T}$ becomes:

$$\begin{aligned}
&\mathit{transT} : \forall\ \{\ G\ A\ b\ v\ w\} \to\\
&\qquad\qquad (C : \mathit{ONonterminal}) \to\\
&\qquad\qquad C \in \mathit{nonterminals}\ \ G \to\\
&\qquad\qquad \mathit{OPT}\ \ G\ \ A\ (b :: v) \to\\
&\qquad\qquad \mathit{TPT}\ \ G\ (n\ \ C - \mathit{O.sn}\ \ A)\ \ w \to\\
&\qquad\qquad \mathit{TPT}\ \ G\ (n\ \ C - \mathit{O.st}\ \ b)\ (v \mathbin{+\!\!+} w)
\end{aligned}$$

In the translation to Agda, we have added an additional agument: the condition $C \in \mathit{nonterminals}\ G$, which states that the current goal nonterminal C is actually used in one of the productions of the grammar. In Figure 5, this was left implicit; in Agda, we need this condition to prove that the productions of the transformed tree really exist in the transformed grammar.

The implementations of *transG* and *transT* are also straightforward translations of the pseudocode of Figure 5. The resulting code does, however, require many small helper proofs in order to prove its type correctness. This clutters the structure of the transformation somewhat, compared to the pseudocode.

By implementing *transG*, we have created a machine-checkable proof that our implementation of the LCT satisfies the language-inclusion property (4).

5 Related Work

Baars et al. [11] implement a left-corner transformation of typed grammars in Haskell. To guarantee that the types of associated semantic functions are preserved across the transformation, they make use of various extensions to Haskell's type system, such as generalized algebraic datatypes for maintaining several invariants and nonstrict evaluation at the type level for wrapping the transformation in an arrow [12]. Inspired by Pasălić and Linger [13], their implementation, which is built on top of a general-purpose library for typed transformations of typed syntax [14], uses what are essentially De Bruijn indices for representing nonterminal symbols. At the expense of some additional complexity, this representation allows for a uniform representation of nonterminals across the transformation. Our approach, at the other hand, requires a dedicated representation (*TNonterminal*) for the nonterminals used in the transformed grammar and a corresponding representation for its productions (*TProduction*). In principle, our implementation could be adapted to use a uniform representation of nonterminal symbols across the transformation as well, but doing so would make it considerably more involved to state and prove properties of our transformation. Baars et al., limited by the restrictions of Haskell's type system, do not state or prove any properties of their implementation other than the preservation of semantic types.

Danielsson and Norell present a library [15] of total parser combinators in Agda. Type-correct parsers in this library are guaranteed not to be left-recursive. It would be interesting to investigate if we could generate a parser for our LCT-transformed grammars using these combinators. A more recent version of the library by Danielsson [16] can actually deal with many left-recursive grammars, by controlling the grammar traversal using a mix of induction and co-induction.

6 Conclusions

We have presented a framework for the representation of grammars, together with their semantics, in Agda. Dependent types make it possible to specify precisely how the type of the semantic functions is determined by the shape of the productions. We can generate parsers for the grammars expressed in our framework with the help of a parser-combinator library.

As an example of the use of our framework, we have shown how to implement the left-corner transform, a transformation that removes left recursion from a grammar. This transformation consists not only of relatively simple manipulations of grammar symbols, but also requires a corresponding adaptation of the semantic functions. Fortunately, we can use the types to guide the implementation: by treating the semantic types as a specification of the desired transformation, the problem is reduced to a search for a function of the appropriate type.

Dependent types play an important role in the development of correctness proofs for our programs. We illustrate this by proving a language-inclusion property for our implementation of the LCT, which states that the transformed grammar derives at least the language of the original grammar. From the transformation rules of the LCT, it is not immediately obvious how the parse trees of the original and transformed grammars are related. A key insight in the proof, due to Johnson [10], is the realization that the grammar transformation effectively simulates a left-corner traversal of the original parse tree. This leads us to a specification of the parse-tree transformation in pseudocode, which involves several invariants on the roots and derived sentences of the parse trees. Those invariants are expressed most naturally using dependent types.

The Agda implementation of the proof is a straightforward translation of the pseudocode specification. As we have shown, the language-inclusion property can be represented elegantly in Agda as a type. The *proof* of this property, which is a parse-tree transformation function satisfying the aforementioned type, also follows directly from the pseudocode. However, to show that this function satisfies the specified type, we have to prove many small helper properties, which clutters the main proof considerably. Although Agda's interactive mode proved helpful in getting these details of the proof right, we speculate that the availability of an extensible tactic language, much as has been available in Coq for many years [17], would even further streamline the construction of proof objects.

During the development of the proof of the language-inclusion property we were confronted with inefficiencies in the current implementation of Agda (version 2.2.4). In order for the memory footprint of the typechecker to fit within the physical memory of the machine, we were forced to factor the proof – sometimes unnaturally – into several submodules. Even with this subdivision, the stand-alone typechecker takes about 8 minutes to check the proof on our hardware,[5] limiting the pace of development.

The framework we have presented in this paper can represent any context-free grammar, but when encoding the grammar we are limited to using plain BNF notation. In contrast, parser combinators are far more expressive than plain BNF, offering constructs such as repetition, optional phrases, and more. One of the key advantages of parser combinators is that they allow us to capture recurring patterns in a grammar by defining custom combinators. Our main focus in extending the present work is to develop a library of "grammar combinators". We envisage a combinator library with an interface similar to that of the parser combinators, which constructs an abstract representation of a grammar in our framework. This representation can then be analyzed, transformed, and ultimately turned into a parsing function.

[5] Using a 2 GHz Intel Core 2 Duo CPU (32-bit) and 2 GB RAM.

Another area we are investigating is the development of a library of grammar transformations, such as removal of ϵ-productions, removal of unreachable productions, or left-factoring.

Currently, the generation of a parser for a grammar makes use of top-down, backtracking parser combinators, which leads to very inefficient parsers. However, from our grammar representation, we can generate many kinds of parsers with various parsing strategies. In particular, our grammar representation is well suited to the kind of global grammar analysis normally performed by standalone parser generators, so we intend to explore the possibility of generating efficient, deterministic bottom-up parsers for our grammars.

Finally, we wish to prove more properties about grammar transformations. For instance, we want to expand the proof of the language-inclusion property into a full correctness proof of our implementation of the LCT. We hope that by doing more proofs, recurring proof patterns for proofs over grammars will emerge that we can then include in our general framework.

Acknowledgements. This work was partly supported by the Netherlands Organisation for Scientific Research through its project on "Scriptable Compilers" (612.063.406) and carried out while the second author was employed at Utrecht University. The authors thank the anonymous reviewers for their helpful remarks and constructive suggestions.

References

1. Hutton, G.: Higher-order functions for parsing. Journal of Functional Programming 2, 323–343 (1992)
2. Swierstra, S.D., Duponcheel, L.: Deterministic, error-correcting combinator parsers. In: Launchbury, J., Sheard, T., Meijer, E. (eds.) AFP 1996. LNCS, vol. 1129, pp. 184–207. Springer, Heidelberg (1996)
3. Swierstra, S.D.: Combinator parsing: A short tutorial. In: Bove, A., Barbosa, L.S., Pardo, A., Pinto, J.S. (eds.) ALFA. LNCS, vol. 5520, pp. 252–300. Springer, Heidelberg (2009)
4. Leijen, D., Meijer, E.: Parsec: Direct style monadic parser combinator for the real world. Technical Report UU-CS-2001-035, Utrecht University (2001)
5. Norell, U.: Dependently typed programming in Agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 230–266. Springer, Heidelberg (2009)
6. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology (2007)
7. Brink, K., Holdermans, S., Löh, A.: Dependently typed grammars, Agda code (2010), `http://www.cs.uu.nl/~andres/DTG/`
8. Fokker, J.: Functional parsers. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 1–23. Springer, Heidelberg (1995)
9. Rosenkrantz, D.J., Lewis, P.M.: Deterministic left corner parsing. In: Conference Record of 1970 Eleventh Annual Symposium on Switching and Automata Theory, pp. 139–152. IEEE, Los Alamitos (1970)
10. Johnson, M.: Finite-state approximation of constraint-based grammars using left-corner grammar transforms. In: COLING-ACL, pp. 619–623 (1998)

11. Baars, A.I., Swierstra, S.D., Viera, M.: Typed transformations of typed grammars: The left corner transform. To appear in the proceedings of the 9th Workshop on Language Descriptions, Tools and Applications (LDTA 2009), York, England (March 29, 2009)
12. Hughes, J.: Generalising monads to arrows. Science of Computer Programming 37, 67–111 (2000)
13. Pasălić, E., Linger, N.: Meta-programming with typed object-language representations. In: Karsai, G., Visser, E. (eds.) GPCE 2004. LNCS, vol. 3286, pp. 136–167. Springer, Heidelberg (2004)
14. Baars, A.I., Swierstra, S.D., Viera, M.: Typed transformations of typed abstract syntax. In: Kennedy, A., Ahmed, A. (eds.) Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, pp. 15–26. ACM Press, New York (2009)
15. Danielsson, N.A., Norell, U.: Structurally recursive descent parsing (2008), `http://www.cs.nott.ac.uk/~nad/publications/danielsson-norell-parser-parser-combinators.html`
16. Danielsson, N.A.: Total parser combinators (2009), `http://www.cs.nott.ac.uk/~nad/publications/danielsson-parser-combinators.html`
17. Delahaye, D.: A tactic language for the system Coq. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNCS (LNAI), vol. 1955, pp. 85–95. Springer, Heidelberg (2000)

Abstraction of Object Graphs in Program Verification

Yifeng Chen[1,⋆] and J.W. Sanders[2,⋆⋆]

[1] HCST Key Lab at School of EECS, Peking University, China
[2] UNU-IIST, Macao

Abstract. A set-theoretic formalism, *AOG*, is introduced to support automated verification of pointer programs. *AOG* targets pointer reasoning for source programs before compilation (before removal of field names). Pointer structures are represented as object graphs instead of heaps. Each property in *AOG* is a relation between object graphs and name assignments of program variables, and specifications result from composing properties. *AOG* extends Separation Logic's compositions of address-disjoint separating conjunction to more restrictive compositions with different disjointness conditions; the extension is shown to be strict when fixpoints are present. A composition that is a 'unique decomposition' decomposes any given graph uniquely into two parts. An example is the separation between the non-garbage and garbage parts of memory. Although *AOG* is in general undecidable, it is used to define the semantics of specialised decidable logics that support automated program verification of specific topologies of pointer structure. One logic studied in this paper describes pointer variables located on multiple parallel linked lists. That logic contains quantifiers and fixpoints but is nonetheless decidable. It is applied to the example of in-place list reversal for automated verification, and in outline to the Schorr-Waite marking algorithm. The technique of unique decomposition is found to be particularly useful in establishing laws for such logics.

1 Introduction

In the standard approach of Formal Methods, in order to achieve accountable programs the programmer is expected to specify and verify code. Even aided by automation, for example in the context of the verifying compiler [14], that is a challenging task requiring expertise not normally attributed to programmers. In this paper an approach is considered in which the programmer is expected to know enough about the properties of the program to be able to issue (meta-level) guidance to a smart type checker that consequently uses library routines to check properties of the program. Here we concentrate on the case of programs that act on mutable data structures, and we support the programmer by considering the

⋆ Partially supported by China HGJ Significant Project 2009ZX01036-001-002-4.
⋆⋆ Acknowledges support from the Macao Science and Technology Development Fund, under the PEARL project, grant number 041/2007/A3.

design of useful shape-related properties for which there is a decision procedure that can be incorporated in a smart type checker to perform static analysis.

Suppose the programmer is occupied with a program involving pointers — say that it is to perform in-place list reversal. Some of the most common errors in pointer programs are related to incorrectly ordered pointer assignments; yet common type systems do not detect such errors. However, if the programmer can identify the overall topology of the pointer structures (of which the programmer is normally aware), the compiler will be able to perform in-depth verification to identify or confirm the absence of such errors (an approach known as 'shape analysis'). Let us suppose that the programmer is fully aware that all pointers are located at some arbitrary positions on parallel lists, which should be free of loops and aliasing. That knowledge is not exploited by the standard C or Java type systems; but it could be. The programmer seeks confirmation (that pointer variables are indeed constrained in parallel lists and free of loops and aliasing) and also that no objects are lost as a result of pointer assignments. If those properties are formalised as annotated invariants, the standard two-line program for in-place list reversal becomes

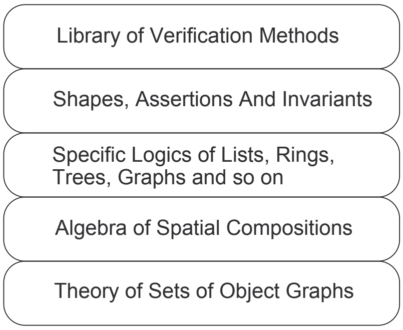
```
inv ParallelLists in
  $y := \text{null}\,;\ z := \text{null}\,;$
  inv NoMemoryLeak in
    while $(x \neq \text{null})$ do  $(z := x.a\,;\ x.a := y\,;\ y := x\,;\ x := z)$.
```

In real applications, pointer structures have diverse topologies such as linked lists, rings, trees and even general graphs. With some assistance from the program to identify the overall topology, a compiler can perform quite precise analysis. Naturally the applications programmer is not expected to write the routines invoked by the compiler. That is the task of the systems programmer who in turn is not expected to establish decision procedures for properties of interest. That is our task.

A shape in 'shape analysis' is usually a syntactic construct that abstractly represents certain pointer-structural feature. The pointer logics mentioned in this paper are logical shape systems that contain propositional logic (with conjunction, disjunction and negation). Each logical formula represents a shape and can be the declared type of a pointer variable, a programmer-annotated assertion or a state invariant. A well-designed pointer logic should be general enough to suit a range of pointer algorithms; and in order to support efficient automated verification, such a logic must be decidable and have a fast decision algorithm. Finally the syntax and semantics of the logic must be succinct and intuitive so that the programmer can understand, in principle, what errors the verifier can detect and what it may miss.

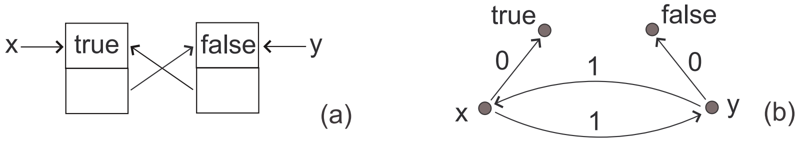
Those criteria obviously conflict. For example, a commonly adopted approach is to start from a general undecidable pointer logic (*e.g.* Separation Logic (SL) [19,24]) and then search for fragments that are expressive but still decidable. That approach addresses the first two criteria but often yields decidable sublogics with convoluted semantics and sophisticated syntactical restrictions on propositional connectives. It is hard for users of the verifier to comprehend what shapes, assertions and invairants can be written and verified in the logic and what kind of false alarms might occur.

As an alternative approach, we start from a set-theoretic framework that allows various spatial compositions and fixpoints. Most of these operators are not expressible in SL (which is already undecidable). Thus no attempt is made to study the large logic including these operators. Instead we develop an algebraic theory that lifts the semantic level of abstraction. Our suggestion is not the design of one single pointer logic, but the definition of a collection of more specialised logics, each (with succinct semantics and syntax) handling a class of pointer applications. The algebraic theory then becomes the common foundation. The programmer chooses the right logic and the corresponding verification algorithm using logical shapes, assertions and invariants in the program. The figure below illustrates the proposed research agenda for verification based on *AOG*. In this paper we concentrate on the design of specialised pointer logics.


Library of Verification Methods
Shapes, Assertions And Invariants
Specific Logics of Lists, Rings, Trees, Graphs and so on
Algebra of Spatial Compositions
Theory of Sets of Object Graphs

In *AOG* (for *Abstract Object Graphs*), pointer structures are represented, like several formalisms studied by the denotational and algebraic-semantic community [11,13,15,17,20,25,26,28], as object graphs (with the presence of precompilation field names). Unlike those formalisms that directly manipulate individual graphs, *AOG* handles *sets* of object graphs. That facilitates *abstract* description of lists, rings, trees and general graphs.

A graph is seen as a finite set of edges, with each edge a tuple of a source name, a label name and a target name. An *object graph* is a graph whose nodes represent objects and whose labelled directed edges represent fields. The label of an edge in every object is unique, reflecting the fact that the object stored at each field is unique.



Graph-based representation reflects the states taken by higher-level OO languages such as Java *before compilation.* Heap representation in SL, on the other hand, reflects the pointer structures *after compilation.* An object is stored in a heap using pointer arithmetic: the first field is stored at an address x and other fields are stored following at $x+1$, $x+2$, $\cdots$. The figures above compare the heap, (a), and object graph, (b), of two mutually referenced objects x and y. Heap representation naturally includes pointer arithmetic, although many decidable SL fragments are free of pointer arithmetic [2,3] (*i.e.* by assuming that each object has exactly two values and only the starting address is accessible). Arithmetic relations between numeric labels can always be added to object graphs. Thus the distinction between the two styles of representation is not essential.

AOG contains 'properties' and 'constructs'. Each property describes a set of object graphs (under certain node-name assignment to variables). Constructs combine properties to form more sophisticated properties (including the fixpoint operator to express transitively-closed compositions) using various spatial compositions whose definitions were motivated by spatial conjunction of SL and the unifying theories approach to parallel composition [5]. A composition corresponds to a relation between graphs. A composition of two graph properties collects all graphs merged from sub-graphs (from the properties) that are related by the designated relation. The set union of two object graphs is not necessarily an object graph, so certain consistency checks are required before a merge. In a graph-based representation, address disjointness corresponds to the condition that requires the object graphs not to contain edges sharing the same source and label. However many useful compositions satisfy that condition other than separating conjunction. In the presence of fixpoints (supplied for SL by Sims [27]), the presence of multiple different compositions becomes strictly more expressive than the presence of just separating conjunction (see Section 2). 'User-defined compositions' [6] (other than separating conjunction) extend separation logic in a way like a 'dynamic separation logic'. This is similar to generalising normal modal logic to dynamic modal logic.

We provide two examples to illustrate application of *AOG*: in-place list reversal (as above) is considered in detail; and the Schorr-Waite marking algorithm is sketched in sufficient detail to indicate our approach. Both are standards and so facilitate comparison of our approach with that of others.

Currently the most popular logical tool for reasoning about pointers is separation logic and its more recent extensions. At the level of explicit memory locations, spatial conjunction provides an expressive power equivalent to a second-order logic, as shown by Kuncak and Rinard [16]; indeed there it is shown that the addition of (unrestricted) spatial conjunction to a decidable logic can lead to an undecidable logic. So automating support for the programmer to reason about pointers using SL must be done carefully. One success, due to Calcagno, Yang and O'Hearn [3], is the decidability of a quantifier-free language, containing separating conjunction and its adjoint, for expressing shape of pointer structures (without properties of data). Another is a custom-crafted logic, due to Rakamarić, Bingham and Hu [23], with a decision procedure for verification of heap-manipulation programs (for further work, see the literature review in [23]); the kinds of property decided include reachability (a node reachable from the root initially is also reachable from the root on termination — referred to above as `NoMemoryLeak`), cyclicity, acyclicity and double-linkedness of a list. We conclude that there is scope for the design of logics that at once express properties of interest concerning shape, yet are decidable.

Graph compositions have been discussed extensively in various graph logics; see for example the work of Courcelle [7]. By comparison, the present paper focuses on user-created compositions. Results related to ours include the decidable fragments of SL studied by Calcagno, Gardner and Hague [2], Calcagno, Yang and O'Hearn [3], and Distefano, O'Hearn and Yang [9] in which quantifiers are not allowed. Berdine, Calcagno and O'Hearn [1] have studied a restrictive decidable fragment with linked lists but without disjunction or quantifiers. Monadic Second-Order Logics (see Dawar, Gardiner and Ghelli's work [8]) use the simple merge operator without consistency checking and allow quantifiers over graphs but do not permit creation of new compositions. Work that is similarly second-order, and also for graph types, is PALE (the pointer assertion logic engine), due to Møller and Schwartzbach [18]. Chang and Rival [4] introduced inductive types for shape analysis. Unfolding (*e.g.* a list) of a structure may occur at different positions (near head or tail) and need different rules to handle different points of breakage in the structure. By comparison, *AOG* does not rely on a specialised inductive formation for lists.

2 Abstraction of Object Graphs

Let $N = \{a, b, c, a_1, \cdots\}$ be a countably infinite set of *names*. An *edge* is a triple (a, b, c) of names where a is called the source, b the label, and c the target. An *object graph*, each denoted as $G, H, G_1, \cdots$, is a set of edges such that for all (a, b, c) and (a, b, d) in the set, we have $c = d$. A *graph property* (denoted $P, Q, P_1, \cdots$) is a set of object graphs. Let $\top$ denote the property containing all object graphs, and $\bot \mathrel{\widehat{=}} \{\,\}$ denote the empty set. The set of all graph properties is denoted Ω. The operator $\overline{P} \mathrel{\widehat{=}} \top \setminus P$ corresponds to set complement. Set union and intersection are $\cup$ and $\cap$ respectively. Let $a \overset{b}{\mapsto} c \mathrel{\widehat{=}} \{\{(a, b, c)\}\}$ denote the edge property, and the empty-graph property be deonted $\varnothing \mathrel{\widehat{=}} \{\{\,\}\}$.

A *graph relation* (denoted $r, r_1, \cdots$) is a relation between object graphs. Let $P \times Q$ denote the Cartesian-product relation. The full relation is $\top \times \top$. We also use $\overline{(\ )}$, $\cup$ and $\cap$ to denote relational complement, union and intersection. An important graph relation $*$ describes the source-label disjointness and corresponds to the spatial conjunction of SL:

$$*(G, H) \mathrel{\widehat{=}} \neg \exists x y z_1 z_2 \cdot (x, y, z_1) \in G \wedge (x, y, z_2) \in H.$$

A *composition* $P\ r\ Q$ is a graph property constructed from the merged graphs of P and Q via the relation r:

$$P\ r\ Q \mathrel{\widehat{=}} \{G \cup H \mid G \in P,\ H \in Q,\ r(G, H)\}.$$

Composition distributes set union in either argument. In this paper, as we focus on object graphs instead of more general graphs; only graph relations smaller than $*$ are considered. Such compositions of graph properties are also graph properties. For example, the property $(a_1 \overset{b_1}{\mapsto} c_1) * (a_2 \overset{b_2}{\mapsto} c_2)$ allows only a graph containing exactly those two edges. However, the property $(a \overset{b}{\mapsto} c) * (a \overset{b}{\mapsto} d)$ is equal to $\perp$, since the two sides have edges sharing the same source and label.

For arbitrary edge extension, we use straight arrows: $a \xrightarrow{b} c \ \widehat{=}\ (a \overset{b}{\mapsto} c) * \top$. The property $(a_1 \xrightarrow{b_1} c_1) \cap (a_2 \xrightarrow{b_2} c_2)$ includes only the graphs containing the edges. The property for edges from source a to target b via an arbitrary label is defined as a universal union: $(a \overset{\bullet}{\mapsto} b)\ \widehat{=}\ \bigcup_c (a \overset{c}{\mapsto} b)$. Similarly, the property for cyclic edges is defined: $cyc\ \widehat{=}\ \bigcup_c (c \overset{\bullet}{\mapsto} c)$, and $acyc\ \widehat{=}\ (\bullet \overset{\bullet}{\mapsto} \bullet) \cap \overline{cyc}$ denotes an acyclic edge (*i.e.* an arbitrary edge that is not a cyclic edge). Bullet $\bullet$ represents union of properties with arbitrary distinct node names. For example, $(\bullet \overset{\bullet}{\mapsto} \bullet)$ denotes $\bigcup_{a,b,c} (a \overset{b}{\mapsto} c)$.

In practice, we often reason about deadends (*i.e.* targets without outgoing edges) and deadheads (*i.e.* sources without incoming edges). For example, to extend a list, we may add an edge to the end (or symmetrically to the head) of the list so that the deadend of the list meets the source of the edge. The property $h(a)\ \widehat{=}\ \varnothing \cup (a \xrightarrow{\bullet} \bullet)$ states that either the node a is a headnode (*i.e.* the source of some edge), or the graph is empty; while $e(a)\ \widehat{=}\ \varnothing \cup (\bullet \xrightarrow{\bullet} a)$ describes an endnode. A node $node(a)\ \widehat{=}\ h(a) \cup e(a)$ within the graph is either a headnode or an endnode. A node a is a deadhead if it is a headnode but not an endnode: $dh(a)\ \widehat{=}\ h(a) \cap \overline{e(a)}$; while $de(a)\ \widehat{=}\ e(a) \cap \overline{h(a)}$ describes a as a deadend. The absence of labels other than a can be defined:

$$label(a)\ \widehat{=}\ \overline{\textstyle\bigcup_{c \neq a} (\bullet \xrightarrow{c} \bullet)}.$$

For example, the property for acyclic edges with label a is $acyc(a)\ \widehat{=}\ acyc \cap label(a)$.

The sequential relation $⨾$ relates two graphs if a deadend of the graph that is the left-hand argument is also a deadhead of the graph that is the right-hand argument, and none of the headnodes of the left is an endnode of the right:

$$⨾\ \ \widehat{=}\ \ * \cap \textstyle\bigcup_x (de(x) \times dh(x))\ \cap\ \overline{\textstyle\bigcup_x (h(x) \times e(x))}.$$

Note that we will use this composition only for properties describing unique deadheads and deadends. The last part of the definition is necessary for general graph properties that describe loops. Sequential composition is associative. For example, a list with three labelled edges is defined:

$$acyc^3(a)\ \widehat{=}\ acyc(a)\ ⨾\ acyc(a)\ ⨾\ acyc(a).$$

Because separating conjunction $*$ does not insist on the RHS not reaching into the LHS, in SL that property would require 'enough inequalities' [1] between variables in conjunction to prevent edges from forming a loop:

$$acyc^3(a) = \textstyle\bigcup_{x_1 x_2 x_3 x_4 :\, \bigwedge_{i=1}^{3} x_i \neq x_4} (x_1 \overset{a}{\mapsto} x_2) * (x_2 \overset{a}{\mapsto} x_3) * (x_3 \overset{a}{\mapsto} x_4).$$

Without the inequalities, the list could form a loop via x_4. The definition using '$\fatsemi$' is arguably more abstract and simpler than the corresponding expression using spatial conjunction. The universal union at the outmost layer reflects that spatial conjunction is not abstract enough for list concatenation.

This issue becomes more significant when fixpoints are present. Let us first define a general property on linked lists comprising recursively $\fatsemi$-concatenated acyclic edges: $ll \mathrel{\widehat{=}} (\varnothing\ (\fatsemi)^*\ acyc)$, which can be comprehended as a (countably) universal union:

$$ll \ = \ \bigcup_k acyc^k \ = \ \varnothing \cup acyc \cup (acyc \fatsemi acyc) \cup \cdots .$$

The definition is so general that it contains no specific names or variables. An obvious law is idempotence: $ll = (ll \fatsemi ll)$. On the other hand, their spatial conjunction $(ll * ll)$ describes two potentially intertwined linked lists and does not satisfy a similar idempotence law. A list segment from some (deadhead) node a to some (deadend) node c through b-labels can be defined as:

$$ls(a, b, c) \mathrel{\widehat{=}} \overline{\varnothing} \cap ll \cap dh(a) \cap label(b) \cap de(c) \qquad \text{(where } a \neq c\text{)},$$

where $ls(a, b, a) \mathrel{\widehat{=}} \varnothing$. For example, any list in Java through attributes next is a list segment: $ls(\bullet, \text{next}, \text{null})$. Two sequentially composed list segments with identical labels still form a longer list segment:

$$ls(a, b, c) \fatsemi ls(c, b, d) \subset ls(a, b, d),$$

where the proper subset indicates that the left-hand segments must go via an intermediate node c, but the right-hand segments may not. On the other hand, their spatial conjunction $ls(a, b, c) * ls(c, b, d)$ allows loops, as d may be identical to some intermediate node of the left-hand segment from a to c. The mechanism to use a graph relation to determine a composition is strictly more expressive than SL's spatial conjunction when fixpoints are present. This parameterised generalisation is analogous to the generalisation from normal modal logic to dynamic modal logic.

This shows that different applications require different compositions: an appropriate composition must be chosen in each specific context (also consider that any doubly-linked list can be decomposed to two singly-linked lists with targets and sources meeting in a reverse order). Although spatial conjunction is not the appropriate concatenation for list segments, it is just right for forming multiple parallel lists in Java. For example, the following property describes two parallel lists following the attributes 'next' and only meeting at the deadend 'null': $ls(a, \text{next}, \text{null}) * ls(b, \text{next}, \text{null})$. The general property for multiple parallel list segments through attributes next are defined: $plists \mathrel{\widehat{=}} \varnothing\ (*)^*\ ls(\bullet, \text{next}, \bullet)$.

3 Manual Reasoning about General Pointer Programs

In this section, we consider object-graph assertions on program variables V. A value assignment $\epsilon : V \to N$ is a mapping from variables to names. An assertion

$p : (V \to N) \to \wp(\Omega)$ is a mapping from value assignments to graph properties. A graph property (e.g. *plists*) can be lifted to (and simply regarded as) a constant assertion that maps every value assignment to the property. Names of a graph property can be substituted with variables to form an assertion. For example,

$$(x \overset{a}{\mapsto} y).\epsilon \;=\; (x_\epsilon \overset{a}{\mapsto} y_\epsilon)$$

where we use x_ϵ to denote the name of the variable x under the value assignment ϵ. The propositional operators on assertions are defined from set theory: conjunction is intersection:

$$(p \wedge q).\epsilon \;\widehat{=}\; p.\epsilon \cap q.\epsilon$$

and negation is complement: $(\neg p).\epsilon \;\widehat{=}\; \overline{p.\epsilon}$. Existential quantification is defined as arbitrary variable substitution:

$$(\exists x \cdot p).\epsilon \;\widehat{=}\; \bigcup_a \, p.(\epsilon \dagger \{x \mapsto a\}),$$

where $f \dagger g$ performs over-riding of relation f by g. Spatial conjunction is pointwise lifting:

$$(p * q).\epsilon \;\widehat{=}\; (p.\epsilon * q.\epsilon).$$

Pointer swing is defined as spatial conjunction after removal of object x's a-attribute:

$$(p[x.a := y]).\epsilon \;\widehat{=}\; \{G \setminus \{(x_\epsilon, a, c) \mid c \in N\} \mid G \in p.\epsilon\} * (x \overset{a}{\mapsto} y).$$

Boolean expressions in pointer programs also correspond to assertions. Note that only pointer variables are considered. Boolean expressions on arithmetic variables do not correspond to any specific pointer assertion; nevertheless we can still capture the upper-limit assertion $\lceil e \rceil$ and lower-limit assertion $\lfloor e \rfloor$ of an expression containing both pointer and arithmetic subexpressions. For pointer equality, we have $\lceil x = y \rceil = \lfloor x = y \rfloor = \top$ if $x_\epsilon = y_\epsilon$; otherwise $\lceil x = y \rceil = \lfloor x = y \rfloor = \bot$. Other equalities related to pointer dereference also have identical upper and lower limits:

$$\lceil x.a = y \rceil \;=\; \lfloor x.a = y \rfloor = (x \xrightarrow{a} y)$$

and

$$\lceil x.a = y.b \rceil \;=\; \lfloor x.a = y.b \rfloor = \bigcup_c \, (x \xrightarrow{a} c \cap y \xrightarrow{b} c).$$

Non-pointer Boolean expressions are not evaluated and hence become either $\top$ as the upper limit or $\bot$ as the lower limit. Boolean `or` and `not` are defined by homomorphism: $\lceil e_1 \texttt{ or } e_2 \rceil \;\widehat{=}\; \lceil e_1 \rceil \vee \lceil e_2 \rceil$, $\lfloor e_1 \texttt{ or } e_2 \rfloor \;\widehat{=}\; \lfloor e_1 \rfloor \vee \lfloor e_2 \rfloor$, $\lceil \texttt{not } e \rceil \;\widehat{=}\; \neg \lfloor e \rfloor$, and $\lfloor \texttt{not } e \rfloor \;\widehat{=}\; \neg \lceil e \rceil$. For example, the Boolean expression $\lceil x.a = y \texttt{ and } w > 5 \rceil$ with both pointer variables x and y and an arithmetic variable w is equal to $(x \xrightarrow{a} y)$, but $\lceil \texttt{not } (x.a = y \texttt{ and } w > 5) \rceil = \top$.

Each pointer program P can be represented as an assertion transformer $\mathbf{sp}.P$ (or operator on assertions) that maps a precondition p to the strongest postcondition $\mathbf{sp}.P.p$ that the program can successfully guarantee; as usual functional composition brackets by default to the left. Let us consider six basic commands:

$$\texttt{skip}, \quad x.a := y, \quad \texttt{endvar}\; x, \quad \texttt{assume}\; e, \quad S \,⨾\, T \quad \text{and} \quad S \sqcap T.$$

Command `skip` does not change the variables. The command $x.a := y$ assigns y to the attribute a of an object x and is defined as a spatial conjunction with the new attribute after removing the existing attribute. Note that if the object x does not have attribute a, then $x.a := \mathrm{null}$ creates a new attribute; and if x is not an existing object, a new object is created. The command `endvar` x renders a variable x arbitrary and existentially quantified. The command `assume` e miraculously forces the Boolean expression to be true and conjoins it with the precondition. This command is useful for defining several advanced commands. Sequential composition corresponds to the composition of transformers. Nondeterministic choice corresponds to disjunction of postconditions.

Definition 1.
$$\begin{aligned}
&\mathbf{sp}.\mathtt{skip}.p \;\widehat{=}\; p\\
&\mathbf{sp}.(x.a := y).p \;\widehat{=}\; p[x.a := y]\\
&\mathbf{sp}.(\mathtt{endvar}\ x).p \;\widehat{=}\; \exists x \cdot p\\
&\mathbf{sp}.(\mathtt{assume}\ e).p \;\widehat{=}\; p \wedge \lceil e \rceil\\
&\mathbf{sp}.(S\,;T).p \;\widehat{=}\; \mathbf{sp}.T.(\mathbf{sp}.S.p)\\
&\mathbf{sp}.(S \sqcap T).p \;\widehat{=}\; \mathbf{sp}.S.p \vee \mathbf{sp}.T.p
\end{aligned}$$

More advanced commands are derivable from the basic commands. Direct assignment $x := y$ between distinct variables forces x to become y after first rendering x arbitrary. Assignment $x := y.b$ requires an unused variable z to record the value of $y.b$, so does assignment $x.a := y.b$. Conditional statement becomes a nondeterministic choice between coerced conditional branches. Loops can be verified using various techniques of symbolic execution and abstract interpretation, though they are not listed as commands here.

Definition 2.
$$\begin{aligned}
&x := y \;\widehat{=}\; \mathtt{endvar}\ x\ ;\ \mathtt{assume}\ x = y\\
&x := y.b \;\widehat{=}\; \mathtt{assume}\ z = y.b\ ;\ x := z\ ;\ \mathtt{endvar}\ z\\
&x.a := y.b \;\widehat{=}\; \mathtt{assume}\ z = y.b\ ;\ x.a := y\ ;\ \mathtt{endvar}\ z\\
&\mathtt{if}\ e\ \mathtt{then}\ S\ \mathtt{else}\ T \;\widehat{=}\; (\mathtt{assume}\ e\ ;\ S) \sqcap (\mathtt{assume}\ \mathtt{not}\ e\ ;\ T)
\end{aligned}$$

Unfortunately the general assertion logic, including the strongest-postcondition operators, is undecidable [1]. This suggests that we use it for only manual reasoning and study its specialised sublogics for automated verification. If every assertion in the logic is automatically reducible to a finite normal form then the compiler, when verifying the preservation of the structure, can automatically generate assertions of the normal form.

4 Automated Verification

Many pointer algorithms, such as in-place list reversal, maintain an overall topological structure consisting of several linked lists meeting only at the deadend null. To check this safety property, a compiler must eliminate all pointer aliases and pointer loops. A logic for assertions of such programs must represent the mutable pointer structures as well as properties such as reachability (requiring fixpoints).

4.1 Assertions for Parallel Lists

The specialised logic of assertions on parallel lists consists of the following primitive assertions and operators:

$$\mathit{true}(X),\quad x \xrightarrow{I} y,\quad x \xrightarrow{I} \text{null},\quad {\sim}p,\quad p \vee q,\quad p * q,\quad \exists x \cdot p \ \text{ and } \ p[x.\text{next} := y].$$

The primitive assertion *true* denotes logical truth and requires variables in X to be located on some parallel lists:

$$\mathit{true}(X) \;\widehat{=}\; \textstyle\bigwedge_{x \in X} \mathit{node}(x) \wedge \mathit{plists}.$$

The primitive $(x \xrightarrow{I} y)$ describes the distance from variable x to variable y on the parallel lists, and $(x \xrightarrow{I} \text{null})$ describes the distance to the end of the list where the integer interval I is $[n, m]$, $[n, \infty)$ or $[n, \infty]$. List segments are the basic building blocks of their definitions:

$$\begin{aligned} x \overset{n}{\mapsto} y &\;\widehat{=}\; \mathit{ls}^n(x, \text{next}, y) \\ x \overset{[0,\infty)}{\mapsto} y &\;\widehat{=}\; \mathit{ls}(x, \text{next}, y) \quad (\text{or simply } x \overset{*}{\mapsto} y) \\ x \overset{n^+}{\mapsto} y &\;\widehat{=}\; (x \overset{n}{\mapsto} \bullet) \,;\, (\bullet \overset{*}{\mapsto} y). \end{aligned}$$

Then the above primitives describe the inclusion of such list segments in the parallel lists:

$$\begin{aligned} (x \xrightarrow{I} y) &\;\widehat{=}\; (x \overset{I}{\mapsto} y * \top) \wedge \mathit{plists} \\ (x \xrightarrow{I} \text{null}) &\;\widehat{=}\; (x \overset{I}{\mapsto} \text{null} * \top) \wedge \mathit{plists}. \end{aligned}$$

Equality is represented as $x \xrightarrow{0} y$ (or $x \xrightarrow{0} \text{null}$). We write $x \xrightarrow{n} y$ for $x \xrightarrow{[n,n]} y$ and $x \xrightarrow{\infty} y$ for $x \xrightarrow{[\infty,\infty]} y$. For example, the property $x \xrightarrow{[1,2]} y$ states that the variables x and y are on the same list and either $x.\text{next} = y$ or $x.\text{next}.\text{next} = y$.

Negation must enforce the restriction on parallel lists: ${\sim}p \;\widehat{=}\; \neg p \wedge \mathit{plists}$. Non-reachability $x \xrightarrow{\infty} y$ is defined as ${\sim}(x \xrightarrow{*} y)$. Conjunction, disjunction, spatial conjunction and quantification are the same as those of the general assertion logic.

4.2 Normal Form

Although the assertions are recursively formed, they are reducible to a semantically equal finite normal form with *signature* X. The idea is to describe each possible overall structure as some parallel lists in which equal pointer variables are grouped, and the distances between adjacent variable groups on the same list are intervals contained by $[1, \infty)$. Writing iterated $*$ as prefix product:

$$\textstyle\bigvee_i \prod_j \left(X_{ij1} \xrightarrow{I_{ij1}} X_{ij2} \xrightarrow{I_{ij2}} \cdots \xrightarrow{I_{ijk-1}} X_{ijk} \xrightarrow{I_{ijk}} X_i \xrightarrow{0} \text{null} \right) \qquad (1)$$

where each X_{ijt} is a non-empty group of equal variables and $I_{ijt} \subseteq [1,\infty)$. Multiple lists are conjoined spatially with $*$, sharing the group X_i of variables that are equal to null. In each disjunct, pairs of different groups are disjoint, and $X = \bigcup_j \bigcup_t X_{ijt} \cup X_i$. The convention $X \xrightarrow{I} Y$ denotes a list segment between variable groups X and Y:

$$\{x_1, \cdots, x_n\} \xrightarrow{I} \{y_1, \cdots, y_m\} \;\;\widehat{=}\;\; x_1 \xrightarrow{0} \cdots \xrightarrow{0} x_n \xrightarrow{I} y_1 \xrightarrow{0} \cdots \xrightarrow{0} y_m .$$

Concatenation $X \xrightarrow{I} Y \xrightarrow{J} Z$ denotes conjunction $X \xrightarrow{I} Y \wedge Y \xrightarrow{J} Z$, and we also use $X \xrightarrow{I} \{\,\} \xrightarrow{0} \text{null}$ as convention for $X \xrightarrow{I} \text{null}$. The underlying structural restriction *plists* prevents the formation of any loop, and hence the normal form (1) is equal to an alternative normal form without spatial conjunction:

$$\bigvee_i \bigwedge_j \left(X_{ij1} \overset{I_{ij1}}{\rightarrow} \cdots X_{ijk} \overset{I_{ijk}}{\rightarrow} X_i \xrightarrow{0} \text{null} \right) \wedge \bigwedge_{j_1 \neq j_2} \left(X_{ij_1 1} \overset{\infty}{\leftrightarrow} X_{ij_2 1} \right) \qquad (2)$$

where the first (non-deadend) groups of variables are mutually unreachable. During later discussions, we will use whichever normal form is convenient.

Assertion $\mathit{true}(\{x\})$ has a normal form $(\{x\} \xrightarrow{1^+} \text{null} \vee \{x\} \xrightarrow{0} \text{null})$ with two disjuncts. The normal form of assertion $\mathit{true}(\{x,y\})$, however, has seven disjuncts:

$$\begin{aligned}
\mathit{true}(\{x,y\}) \;=\;\; & \{x\} \xrightarrow{1^+} \{y\} \xrightarrow{1^+} \text{null} \\
\vee\; & \{y\} \xrightarrow{1^+} \{x\} \xrightarrow{1^+} \text{null} \\
\vee\; & \{x\} \xrightarrow{1^+} \text{null} \;\;*\;\; \{y\} \xrightarrow{1^+} \text{null} \\
\vee\; & \{x\} \xrightarrow{1^+} \{y\} \xrightarrow{0} \text{null} \\
\vee\; & \{y\} \xrightarrow{1^+} \{x\} \xrightarrow{0} \text{null} \\
\vee\; & \{x,y\} \xrightarrow{1^+} \text{null} \\
\vee\; & \{x,y\} \xrightarrow{0} \text{null} .
\end{aligned}$$

Note that the normal form is not unique: for example, each range $[1,\infty)$ can be further decomposed into $\{1\} \cup [2,\infty)$. However, the above normal form is minimum as the relation between every pair of variables must be clarified in every disjunct.

The normal form of $\mathit{true}(X \cup \{x\})$ can be constructed inductively from that of $\mathit{true}(X)$ (namely $x \notin X$). Each disjunct consists of some spatially conjoined lists. The new variable x is added to each disjunct. There are three possibilities: it may be added to one of the variable groups (being equal to some existing variables), it may form a new singleton group between two adjacent groups or before the first group, or it may lead a separate new list. There are $2n+1$ different cases in total where n is the number of existing groups. The same technique can be used to add a new variable to any assertion in normal form where the law

$$(X \xrightarrow{n} Y \;\wedge\; X \xrightarrow{*} Z \xrightarrow{*} Y) \;=\; \bigvee_{l+m=n} X \xrightarrow{l} Z \xrightarrow{m} Y$$

is useful in determining the intervals.

The following laws reduce primitive assertions to normal form.

Law 1

(1) $x \xrightarrow{0} y \; = \{x, y\} \xrightarrow{1^+} \text{null} \vee \{x, y\} \xrightarrow{0} \text{null}$

(2) $x \xrightarrow{I} y \; = \{x\} \xrightarrow{I} \{y\} \xrightarrow{1^+} \text{null} \vee \{x\} \xrightarrow{I} \{y\} \xrightarrow{0} \text{null} \qquad (I \subseteq [1, \infty))$

(3) $x \xrightarrow{\infty} y \; = \{x\} \xrightarrow{1^+} \text{null} * \{y\} \xrightarrow{1^+} \text{null} \vee \{y\} \xrightarrow{1^+} \{x\} \xrightarrow{1^+} \text{null} \vee \{y\} \xrightarrow{1^+} \{x\} \xrightarrow{0} \text{null.}$

Conjunction between two normal-form assertions with the same signature are merged using the law:

$$X \xrightarrow{I_1} Y \wedge X \xrightarrow{I_1} Y \; = \; X \xrightarrow{I_1 \cap I_2} Y.$$

To conjoin assertions with different signatures, we add variables to either assertion until they have the same merged signature. Negation, conjunction and disjunction satisfy standard propositional laws. Negations over primitive assertions are eliminated:

Law 2. (1) $\sim(X \xrightarrow{[m,n]} Y) \; = \; (X \xrightarrow{[0,m-1]} Y) \vee (X \xrightarrow{[n+1,\infty]} Y)$

(2) $\sim(X \xrightarrow{[m,n]} \text{null}) \; = \; (X \xrightarrow{[0,m-1]} \text{null}) \vee (X \xrightarrow{(n+1)^+} \text{null}).$

The above laws are already complete for transforming any assertion without spatial conjunction and quantification to the normal form. Spatial conjunction between two lists can be reduced to a conjunction with additional non-reachability restrictions:

Law 3. $(X_1 \xrightarrow{I_1} \cdots \xrightarrow{I_n} X_{n+1}) * (Y_1 \xrightarrow{I_1} \cdots \xrightarrow{I_m} Y_{m+1}) \; =$

$(X_1 \xrightarrow{I_1} \cdots \xrightarrow{I_n} X_{n+1}) \wedge (Y_1 \xrightarrow{I_1} \cdots \xrightarrow{I_m} Y_{m+1}) \wedge \bigwedge_i \bigwedge_j X_i \overset{\infty}{\leftrightarrow} Y_j \,.$

Existential quantifiers are eliminated with the following laws, where $I_1 + I_2$ denotes $\{n_1 + n_2 \mid n_1 \in I_1, \, n_2 \in I_2\}$:

Law 4. (1) $\exists x \cdot (p \wedge \exists x \cdot q) \; = \; \exists x \cdot p \wedge \exists x \cdot q$

(2) $\exists x \cdot (X \xrightarrow{I} Y) \; = \; X \xrightarrow{I} Y \qquad (x \notin X \cup Y)$

(3) $\exists x \cdot (\{x\} \xrightarrow{I} X) \; = \; \mathit{true}(X)$

(4) $\exists x \cdot (X \xrightarrow{I} \{x\} \xrightarrow{J} Z) \; = \; X \xrightarrow{I+J} Z$

(5) $\exists x \cdot (X \xrightarrow{I} Y \xrightarrow{J} Z) \; = \; X \xrightarrow{I} Y \setminus \{x\} \xrightarrow{J} Z \qquad (x \in Y \neq \{x\})$

(6) $\exists x \cdot (X \xrightarrow{I} Y \xrightarrow{0} \text{null}) \; = \; X \xrightarrow{I} Y \setminus \{x\} \xrightarrow{0} \text{null} \qquad (x \in Y)\,.$

Note that detaching the only variable leading a list will leave a segment not referenced by any variables. Attribute assignment may break a list into spatially conjoined two segments, which are further reducible to normal form. Attribute assignment to a null pointer will lead to compilation error $\perp$.

Law 5

(1) $\mathbf{sp}.(x.\text{next} := y).((X_1 \xrightarrow{I_1} \cdots X_i \xrightarrow{I_i} X_{i+1} \xrightarrow{I_{i+1}} \cdots X_k \xrightarrow{I_k} X \xrightarrow{0} \text{null}) * p)$

$= \; (X_1 \xrightarrow{I_1} \cdots X_i \xrightarrow{1} \{y\}) * (X_{i+1} \xrightarrow{I_{i+1}} \cdots X_k \xrightarrow{I_k} X \xrightarrow{0} \text{null}) * p \qquad (x \in X_i)$

(2) $\mathbf{sp}.(x.\text{next} := y).(X \xrightarrow{0} \text{null} \wedge p) \; = \; \perp \qquad (x \in X).$

Theorem 1 (Normal-Form Completeness). *Every assertion is semantically equal to an assertion in normal form, and there is a finite decision procedure to determine the validity of the normal form.*

Proof. Let $X \mathrel{\widehat{=}} \{x_, \ldots, x_n\}$ be the set of variables. For the property true, we need to enumerate all possible layouts of these variables on parallel lists. This can be achieved by enumerating all partitions of X and, for each part in each partition, identifying all possible (total) orderings among variables, assigning 0 or n^+ to the distances between adjacent variables, creating multiple chains, and finally using disjunction and separating conjunction appropriately to construct a normal form. The number of disjuncts is estimated in the order of $O(2^n)$ (by Rademacher series). Law 2 guarantees that any sublogical property without quantifiers has a negation-free non-constructive normal form:

$$\bigvee_i \bigwedge_j u_{ij} \overset{I_{ij}}{\rightarrow} v_{ij}\ . \tag{3}$$

This transformation takes $O(2^m)$ steps where m is the maximum of the numbers of negations and disjunctions. Each disjunct

$$\bigwedge_j u_{ij} \overset{I_{ij}}{\rightarrow} v_{ij} \tag{4}$$

is logically conjoined with true's normal form. Only those disjuncts of true that are consistent with all conjuncts $u_{ij} \overset{I_{ij}}{\mapsto} v_{ij}$ are collected to form the normal form of (4). The normal form of (3) is the disjunction of the normal forms obtained above. This phase is estimated to have $O(2^{n+m})$ steps.

If there is one outermost existential quantifier $\exists x \cdot P$, then Law 4 can be used to eliminate the quantifier over the normal form of P. Thus the overall reduction takes about $O(2^{n+m})$ steps where m is the maximum of the numbers of negations, disjunctions and existential quantifiers. Pointer swing $p[x.a := y]$ requires Law 5.

An assertion p is valid iff its negation contains no disjuncts and is $\perp$. This can be achieved by reducing $\sim p$ to the normal form in $O(2^{n+m})$ steps. □

4.3 Automated Program Verification of Assertions

We now apply static symbolic execution over assertions as abstract states. If the current abstract state is p, then after the execution of a program S, the generated assertion is $\mathbf{sp}.S.p$.

Along with symbolic execution, various safety conditions should be checked. For example, accessing $x.\mathrm{next}$ requires that the attribute of the object exists. Other undesirable cases include null pointer dereference, nontermination, aliases and loops. So there are different strategies about what should be checked. A conservative strategy —to prevent all possible dynamic errors— is likely to generate false alarms; while a liberal strategy —to report error only if the code always fails— is likely to miss program bugs.

Generalising the underlying assertion logic may improve the precision of verification but it also increases the complexity. Non-pointer assignments for arithmetics are regarded as skip. More-precise analysis that takes arithmetic variables into account is possible but requires extension of the logic (*e.g.* [4]). Here we propose a moderately conservative strategy that detects pointer-related errors. Let $\mathbf{pre}.S$ denote the condition under which an assignment S can proceed safely without generating errors or destroying the overall structure of parallel lists:

$$\begin{aligned}
&\mathbf{pre}.(x := y) \;\widehat{=}\; \top\\
&\mathbf{pre}.(x := y.\mathrm{next}) \;\widehat{=}\; y \xrightarrow{1^+} \mathrm{null}\\
&\mathbf{pre}.(x.\mathrm{next} := y.\mathrm{next}) \;\widehat{=}\; (x \xrightarrow{0} y \,\vee\, y \xrightarrow{1} \mathrm{null}) \wedge x \xrightarrow{1^+} \mathrm{null}\\
&\mathbf{pre}.(x.\mathrm{next} := y) \;\widehat{=}\; x \xrightarrow{1^+} \mathrm{null} \wedge y \overset{\infty}{\nrightarrow} x \wedge \textstyle\bigwedge_z (z \xrightarrow{0} y \vee z \overset{\infty}{\nrightarrow} y \,\vee\, y \xrightarrow{0} \mathrm{null} \,\vee\, z \xrightarrow{*} x)\,.
\end{aligned}$$

Before computing the strongest postcondition of any assignment from a precondition, the verifier must check that the precondition implies the safety condition. As the assertion logic is decidable, such checking can be automated.

The assignment $x.a := y$ requires that $x \neq \mathrm{null}$, y is not path-reachable to x (forming a loop), and a non-null pointer y should be either at the start of another list or reachable from x. The evaluation of expressions may also generate errors. Let $\mathbf{pre}.e$ denote the condition for an expression e to be safely evaluated: $\mathbf{pre}.e \;\widehat{=}\; \bigwedge_{x\in V(e)} x \xrightarrow{1^+} \mathrm{null}$ where $V(e)$ is the set of free variables x such that $x.a$ appears in e.

Program loop ($\texttt{while}\ b\ \texttt{do}\ S$) is handled by static iteration in abstract states and checks the safety conditions in every step. Here we are using a widening operator $p\!\uparrow$ over assertions. The space of abstract states is infinite. Widening forces the static iteration to reach a fixpoint in finitely-many steps. In the normal form, the widening operator lifts every singleton range $[n, n]$ to $[n, \infty)$ (or written as n and n^+ respectively). For example, $(y \xrightarrow{2} x \xrightarrow{0} \mathrm{null}) \uparrow = (y \xrightarrow{2^+} x \xrightarrow{0} \mathrm{null})$.

The verification of a program corresponds to symbolic execution over abstract states, each as a logical formula. When there is loop, the symbolic execution may go through the loop body several times. Since the sizes of the minimum normal forms are unbounded, the widening operator is employed to force termination. We will use // to separate the abstract states of different iterations:

$$\begin{array}{l}
\{p_0 \quad (\text{check } p_0 \Rightarrow \mathbf{pre}.b\,)\}\\
\;//\,\{p_1 \quad (\,\widehat{=}\; p_0' \uparrow \wedge {\sim} p_0,\quad \text{check } p_1 \Rightarrow \mathbf{pre}.b \text{ and } p_1 \not\Rightarrow {\sim}\lceil b\rceil\,)\,\}\\
\cdots\cdots\\
\;//\,\{p_{m+1} \quad (\,\widehat{=}\; p_m' \uparrow \wedge {\sim} p_m \wedge \cdots \wedge {\sim} p_0,\\
\qquad\qquad\quad \text{check } p_{m+1} \Rightarrow \mathbf{pre}.b \text{ and } p_{m+1} \Rightarrow {\sim}\lceil b\rceil\,)\,\}
\end{array}$$

$$\begin{array}{l}
\texttt{while}\ b\ \texttt{do}\quad \{p_0 \wedge \lceil b\rceil\} \,//\, \{p_1 \wedge \lceil b\rceil\} \,//\, \cdots \,//\, \{p_m \wedge \lceil b\rceil\}\\
\qquad\quad S \qquad \{p_0'\} \,//\, \{p_1'\} \,//\, \cdots \,//\, \{p_m'\}\\
\{\,(p_0 \vee \cdots \vee p_m) \wedge \lceil \texttt{not}\ b\rceil)\}
\end{array}$$

where $p_i' \mathrel{\widehat{=}} \mathbf{sp}.S.(p_i \wedge \lceil b \rceil)$. The initial abstract state p_0 must allow b to be evaluated successfully. The abstract state p_1 at the beginning of the second iteration is the widened result of the first iteration negating p_0. Previous checking does not need to be repeated. If p_1 is inconsistent with $\lceil b \rceil$ (*i.e.* $p_1' \Rightarrow \sim\lceil b \rceil$)) then all possible initial states have been covered by p_0, and no further static iteration is needed. If $p_1 \wedge \lceil b \rceil$ is not invalid, the iteration continues until the abstract state p_{m+1} is inconsistent with $\lceil b \rceil$, and the final abstract state is the collection of all p_i conjoined with $\lceil \texttt{not}\ b \rceil$. The following theorem guarantees termination of the verification.

Theorem 2. *The parallel-list verification terminates in finitely-many steps.*

Proof. The number k of variables appearing in a program is finite. That means the monotonic widening reaches a fixpoint in no more than k steps. □

4.4 List Reversal

The method is used to verify that the list reversal program has no errors like dereferencing a null pointer, forming pointer loops or aliases.

```
$\{\,true\,\}$
$y :=$ null; $z :=$ null;
$\Big\{\{y,z\} \xrightarrow{0} \text{null}\Big\} \,/\!/\, \Big\{\,\{y\} \xrightarrow{1^+} \{x,z\} \xrightarrow{0} \text{null} \ \vee\ \{y\} \xrightarrow{1^+} \text{null} * \{x,z\} \xrightarrow{1^+} \text{null}\Big\}$
while $(x \neq \text{null})$ do   $\Big\{\{x\} \xrightarrow{1^+} \{y,z\} \xrightarrow{0} \text{null}\Big\} \,/\!/\, \Big\{\,\{y\} \xrightarrow{1^+} \text{null} * \{x,z\} \xrightarrow{*} \text{null}\Big\}$
    $z := x.\text{next}$;   $\Big\{\{x\} \xrightarrow{1} \{z\} \xrightarrow{*} \{y\} \xrightarrow{0} \text{null}\Big\}$
    $/\!/\Big\{\,\{y\} \xrightarrow{1^+} \text{null} * \{x\} \xrightarrow{1} \{z\} \xrightarrow{0} \text{null} \ \vee\ \{y\} \xrightarrow{1^+} \text{null} * \{x\} \xrightarrow{1} \{z\} \xrightarrow{1^+} \text{null}\ \Big\}$
    $x.\text{next} := y$;   $\Big\{\,\{x\} \xrightarrow{1} \{y,z\} \xrightarrow{0} \text{null} \ \vee\ \{x\} \xrightarrow{1} \{y\} \xrightarrow{0} \text{null} * \{z\} \xrightarrow{1^+} \text{null}\Big\}$
    $/\!/\Big\{\,\{x\} \xrightarrow{1} \{y\} \xrightarrow{1^+} \{z\} \xrightarrow{0} \text{null} \ \vee\ \{x\} \xrightarrow{1} \{y\} \xrightarrow{1^+} \text{null} * \{z\} \xrightarrow{1^+} \text{null}\Big\}$
    $y := x$;   $\Big\{\,\{x,y\} \xrightarrow{1} \{z\} \xrightarrow{0} \text{null} \ \vee\ \{x,y\} \xrightarrow{1} \text{null} * \{z\} \xrightarrow{1^+} \text{null}\Big\}$
    $/\!/\Big\{\,\{x,y\} \xrightarrow{2^+} \{z\} \xrightarrow{0} \text{null} \ \vee\ \{x,y\} \xrightarrow{2^+} \text{null} * \{z\} \xrightarrow{1^+} \text{null}\Big\}$
    $x := z$;   $\Big\{\,\{y\} \xrightarrow{1} \{x,z\} \xrightarrow{0} \text{null} \ \vee\ \{y\} \xrightarrow{1} \text{null} * \{x,z\} \xrightarrow{1^+} \text{null}\Big\}$
    $/\!/\Big\{\,\{y\} \xrightarrow{2^+} \{x,z\} \xrightarrow{0} \text{null} \ \vee\ \{y\} \xrightarrow{2^+} \text{null} * \{x,z\} \xrightarrow{1^+} \text{null}\Big\}$
$\Big\{\,\{x,z\} \xrightarrow{0} \text{null}\Big\}$
```

As observed in the introduction, simple shape analysis can avoid common pointer errors involving incorrect ordering of pointer assignments. For example, if the last two assignments in the loop body are wrongly swapped, the compiler will pick up this error in the second static iteration when the program may form a pointer loop from y to itself.

```
$y := \text{null}\,;\ z := \text{null}\,;$    $\Big\{\{y,z\}\xrightarrow{0}\text{null}\Big\}\ /\!/ \Big\{\ \{x,y,z\}\xrightarrow{1^+}\text{null}\Big\}$
while $(x \neq \text{null})$ do    $\Big\{\{x\}\xrightarrow{1^+}\{y,z\}\xrightarrow{0}\text{null}\Big\}\ /\!/ \Big\{\ \{x,y,z\}\xrightarrow{1^+}\text{null}\Big\}$
    $z := x.\text{next}\,;$    $\Big\{\{x\}\xrightarrow{1}\{z\}\xrightarrow{*}\{y\}\xrightarrow{0}\text{null}\Big\}\ /\!/ \Big\{\ \{x,y\}\xrightarrow{1}\{z\}\xrightarrow{*}\text{null}\Big\}$
    $x.\text{next} := y\,;$    $\Big\{\ \{x\}\xrightarrow{1}\{y,z\}\xrightarrow{0}\text{null}\ \vee\ \{x\}\xrightarrow{1}\{y\}\xrightarrow{0}\text{null} * \{z\}\xrightarrow{1^+}\text{null}\Big\}$
                $/\!/$ error: forming loop
    $x := z\,;$    $\Big\{\ \{x,z\}\xrightarrow{*}\{y\}\xrightarrow{0}\text{null}\Big\}$
    $y := x\,;$    $\Big\{\ \{x,y,z\}\xrightarrow{*}\text{null}\Big\}$
```

Another feature that we can verify using assertions concerns memory leakage. The assignment to a variable x must maintain the pointer structure and check that the object initially referenced by x is still reachable from other variables. Even if a pointer swing $x.\text{next} := y$ maintains the structure of parallel lists, it may still lose objects unless the object $x.\text{next}$ is null or previously referenced by another pointer variable (a condition represented as $\bigvee_{y\,\in\,V}\ x\xrightarrow{1}y$) where V is the set of all variables:

$$\begin{aligned}
\mathbf{pre}'.(x := y) \ &\widehat{=}\ y\xrightarrow{0}x \ \vee\ \textstyle\bigvee_{z\in V\setminus\{x\}}\ z\xrightarrow{*}x\\
\mathbf{pre}'.(x := y.\text{next}) \ &\widehat{=}\ \mathbf{pre}.(x := y.\text{next}) \ \wedge\ \textstyle\bigvee_{z\in V\setminus\{x\}}\ z\xrightarrow{*}x\\
\mathbf{pre}'.(x.\text{next} := y.\text{next}) \ &\widehat{=}\ \mathbf{pre}.(x.\text{next} := y.\text{next}) \ \wedge\ \textstyle\bigvee_{z}\ x\xrightarrow{1}z\\
\mathbf{pre}'.(x.\text{next} := v) \ &\widehat{=}\ \mathbf{pre}.(x.\text{next} := v) \ \wedge\ \textstyle\bigvee_{z}\ x\xrightarrow{1}z\,.
\end{aligned}$$

For in-place list reversal, if the first two assignments in the loop body are wrongly swapped, then the assignment $x.\text{next} := y$ will shortcut the linked list from x and set the object $x.\text{next}$ to be null immediately. If $x.\text{next}$ is initially a non-null object, then the shortcut assignment will lose the reference to that object. This error is detectable in the first static iteration:

```
$y := \text{null}\,;\ z := \text{null}\,;$    $\Big\{\{y,z\}\xrightarrow{0}\text{null}\Big\}$
while $(x \neq \text{null})$ do    $\Big\{\{x\}\xrightarrow{1^+}\{y,z\}\xrightarrow{0}\text{null}\Big\}$
    $x.\text{next} := y\,;$    error: memory leakage
    $z := x.\text{next}\,;$
    $y := x\,;$
    $x := z\,.$
```

4.5 Schorr-Waite Graph Marking

We now indicate how *AOG* may be used to discuss safety verification for a different algorithm, the Schorr-Waite algorithm, chosen to exemplify different topological constraints. Abstraction of general object graphs can be achieved with different levels of precision. Here, we consider the abstraction that describes the rough distances between objects (e.g. $x, y, \cdots$) and their immediate fields (e.g. $x.l, y.r, \cdots$).

There are three possibilities: $x \overset{0}{\to} y$ for pointer equality, $x \overset{+}{\to} y$ for unequal reachability and $x \overset{\infty}{\to} y$ for non-reachability in assertions. For example, in this particular analysis, equality between $x.l.r$ and y is represented as $x.l \overset{+}{\to} y$, which means that the latter is not equal to but reachable from the former. It turns out that such abstraction is already precise enough to verify the basic safety properties of Gries's Schorr-Waite code [10], such as the absence of memory leakage.

Assume that at the beginning of the algorithm, the l-field of z is x, and $y = z$. In the first iteration, the predicate-abstract state reaches X_1 after the pointer assignment and the conditional test for the counter $x.m$ to be 3 or 0. Note that the logic does not handle arithmetic variables. The conditional essentially becomes a nondeterministic choice under such analysis. The other conditional branch reaches the abstract state X_1' instead. The disjunctively accumulated predicate-abstract states reach a fixpoint in four rounds (separated by double backslashes). Interestingly, although the two branches of the conditional are very different, the result assertions are entirely symmetric (with positions of x and y swapped):

```
{$z.l = x \wedge y = z \quad (X_0)$}
{$X_1 \vee X_1'$} // {$X_2 \vee X_2'$} // {$X_3 \vee X_3' \vee X_4 \vee X_4'$} // {$X_5 \vee X_5'$}
while ($x \neq z$) and ($x \neq$ null) do
    {$y = z \neq z.l = x \neq \text{null}$} //{$z.l = y \neq y.r = z \neq x \neq \text{null} \vee z.l = x \neq x.r = z$}
    $x.m := x.m + 1$;
    if ($x.m = 3$ or $x.m = 0$)
    then $x, x.l, x.r, y := x.l, x.r, y, x$    {$z.l = y \neq y.r = z \quad (X_1)$}
        //{$z \neq y.r = z.l \overset{+}{\to} z \neq y \ \vee\ z.l = y \neq y.l = z \quad (X_2)$}
        //{$y.r \overset{+}{\to} z.l \overset{+}{\to} z \neq z.l \wedge y \neq z \ \vee\ z \neq y.l = z.l \overset{+}{\to} z \neq y \quad (X_3)$
           $\vee\ x = z \neq z.l = y \neq \text{null} \quad (X_4)$}
        //{$X_3 \vee X_1 \vee y.l \overset{+}{\to} z.l \overset{+}{\to} z \neq z.l \wedge y \neq z \quad (X_5)$}
    else $x.l, x.r, y := x.r, y, x.l$    {$z.l = x \neq x.r = z \quad (X_1')$}
        //{$z \neq x.r = z.l \overset{+}{\to} z \neq x \ \vee\ z.l = x \neq x.l = z \quad (X_2')$}
        //{$x.r \overset{+}{\to} z.l \overset{+}{\to} z \neq z.l \wedge x \neq z \ \vee\ z \neq x.l = z.l \overset{+}{\to} z \neq x \quad (X_3')$
           $\vee\ x = z \neq z.l = x \neq \text{null} \quad (X_4')$}
        //{$X_3' \vee X_1' \vee x.l \overset{+}{\to} z.l \overset{+}{\to} z \neq z.l \wedge x \neq z \quad (X_5')$}
{$(x = z \vee x = \text{null}) \wedge (X_0 \vee \bigvee_{i=1}^{5} X_i \vee X_i')$}.
```

The compiler does not verify the functional correctness of the code. It checks the safety property that at least the algorithm does not lose useful content. This still can significantly enhance the program's trustability.

5 Conclusions

Logic-based program verification often faces a dilemma. If we rely on a general and expressive logic (*e.g.* to combine pointer and arithmetic reasoning in one logic) we may be able to improve the overall precision of the analysis. But unfortunately such a logic is likely to be undecidable and suitable for only manual reasoning. To automate such reasoning, we may study the general logic's logical fragments in the hope of discovering an expressive but still-decidable sublogic. However, such effort often results in ill-shaped logics whose restrictions and limitations are hard for users to comprehend. The software engineers using such a verification tool are unlikely to be able to understand what the tool really does and when there will be false alarms and missed errors.

This paper has proposed a different style of solution to program verification. Instead of making the effort to discover decidable fragments of undecidable general logics, we propose to introduce a library of well-shaped specialised logics, each handling the verification of a certain aspect of programs and having a fast decision procedure with comprehensible power and limitations. Users are then able to choose the required verification tool(s) according to their understanding of the functionality of the underlying logic(s).

Shape analysis [9,4] has undergone a similar development, in which certain formal structures are introduced and manipulated using inference rules. Such systems often encounter restrictions related to the construction of the structures. For example, if the underlying structure of linked lists is recursively defined by breaking the first element [4], then the shape abstraction can be unfolded from only that point; unfolding from some other point requires a 'specific', manually-proved, lemma.

Since a 'best' verification does not exist, we have chosen to support multiple methods with different tradeoffs. The minimum requirement is that software engineers understand not only their programs but also the verification methods they choose to apply. One advantage of designing smaller logics is that they can incorporate negation and proposition calculus. Negation together with disjunction/conjunction provides a more complete conceptual framework for its understanding. Negation is also often used in the more abstract, specification-related, stages of development.

A decidable logical system is applicable to not only program verification but also consistency checking of specifications and refinement between specification and program. All such logics share the same semantic foundation in set theory and there is no limit to the operators that can be introduced. Compositions like spatial conjunction and sequential composition are useful for raising the abstraction level of manual reasoning. The laws of specialised logics may also rely on the properties of these high-level operators. The design of the logics should ensure that there exists a finite normal form and a fast decision procedure that reduces any assertion to the normal form.

Automated verification of legacy pointer codes is restricted by the lack of information about the pointer variables' roles in a program. In a standard C/Java program, the type of a pointer variable determines only its object type and does not indicate whether it points to a tree or some position in a loop; but that information can be extremely useful for a compiler in conducting the most appropriate static analysis. On the other hand, manual reasoning can establish the entire correctness of a program with respect to some formal assertion, but the general formalism used is often undecidable. We have adopted an alternative tradeoff by requiring the source programmer to provide a small amount of information about how the variables are used in the program, by identifying the overall topology of the pointer structures. With such information, the compiler can then perform in-depth analysis.

Assertions in our examples are generated, but they can be inserted by programmers too. An interesting future direction is the study of interactions between various verification methods based on *AOG* as well as their interaction with other logical methods of arithmetics. One obvious advantage of running two analysis methods at the same time is to improve the precision of static evaluation of Boolean expressions in conditional statements.

Acknowledgements

It is a pleasure to acknowledge the comprehensive and careful comments of the four referees. Also, the authors are grateful to encouragement and advice from Professor C. Calcagno.

References

1. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 97–109. Springer, Heidelberg (2004)
2. Calcagno, C., Gardner, P., Hague, M.: From separation logic to first-order logic. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 395–409. Springer, Heidelberg (2005)
3. Calcagno, C., Yang, H., O'Hearn, P.W.: Computability and complexity results for a spatial assertion language for data structures. In: APLAS, pp. 289–300 (2001)
4. Chang, B.Y.E., Rival, X.: Relational inductive shape analysis. In: POPL '08, pp. 247–260. ACM, New York (2008)
5. Chen, Y., Sanders, J.W.: Logic of global synchrony. ACM TOPLAS 26(2), 221–262 (2004)
6. Chen, Y., Sanders, J.W.: Compositional reasoning for pointer structures. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 115–139. Springer, Heidelberg (2006)
7. Courcelle, B.: Graph decompositions definable in monadic second-order logic. In: 7th International Colloquium on Graph Theory. Electronic Notes in Discrete Mathematics, vol. 22(15), pp. 13–19 (2005)
8. Dawar, A., Gardiner, P., Ghelli, G.: Expressiveness and complexity of graph logic. Information and Computation 205, 263–310 (2006)
9. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)

10. Gries, D.: The Schorr-Waite graph marking algorithm. Acta Inf. 11, 223–232 (1979)
11. Harwood, W., Cavalcanti, A., Woodcock, J.: A theory of pointers for the UTP. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 141–155. Springer, Heidelberg (2008)
12. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall, Englewood Cliffs (1998)
13. Hoare, C.A.R., He, J.: A trace model for pointers and objects. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 1–17. Springer, Heidelberg (1999)
14. Hoare, C.A.R.: The verifying compiler: A grand challenge for computing research. Journal of the ACM 50(1), 63–69 (2003)
15. Ke, W., Liu, Z., Wang, S., Zhao, L.: A graph-based operational semantics of OO programs (May 2009)
16. Kuncak, V., Rinard, M.C.: On spatial conjunction as second-order logic. MIT CSAIL Technical Report 970 (October 2004)
17. Liu, X., Liu, Z., Zhao, L.: Object-oriented structure renement A graph transformational approach. UNU-IIST Technical Report 340 (July 2006)
18. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: PLDI'01, pp. 221–231 (2001)
19. O'Hearn, P.W., Reynolds, J., Yang, H.: Separation and information hiding. In: POPL'04, vol. 2142, pp. 268–280. ACM, New York (2004)
20. Paige, R.F., Ostroff, J.S.: Erc: an object-oriented renement calculus for Eiffel. Formal Aspects of Computing 16(1), 51–79 (2004)
21. Parkinson, M.J.: Local Reasoning for Java. PhD thesis, University of Cambridge, Computer Laboratory (November 2005)
22. Preoteasa, V.: Frame rule for mutually recursive procedures manipulating pointers. Theoretical Computer Science 410(42) (2009)
23. Rakamarić, Z., Bingham, J., Hu, A.J.: An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 106–121. Springer, Heidelberg (2007)
24. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: LICS'02, pp. 55–74. IEEE Computer Society, Los Alamitos (2002)
25. Schieder, B.: Pointer theory and weakest preconditions without addresses and heap. In: Kozen, D., Shankland, C. (eds.) MPC 2004. LNCS, vol. 3125, pp. 357–380. Springer, Heidelberg (2004)
26. Silva, L., Sampaio, A., Liu, Z.: Laws of object-orientation with reference semantics. In: SEFM, pp. 217–226 (2008)
27. Sims, E.J.: Extending Separation Logic with fixpoints and postponed substitution. Theoretical Computer Science 351(2), 258–275 (2006)
28. Smith, M.A., Gibbons, J.: Unifying theories of locations. In: Butterfield, A. (ed.) Unifying Theories of Programming, Dublin (September 2008)

Subtyping, Declaratively
An Exercise in Mixed Induction and Coinduction

Nils Anders Danielsson and Thorsten Altenkirch

University of Nottingham

Abstract. It is natural to present subtyping for recursive types coinductively. However, Gapeyev, Levin and Pierce have noted that there is a problem with coinductive definitions of non-trivial transitive inference systems: they cannot be "declarative"—as opposed to "algorithmic" or syntax-directed—because coinductive inference systems with an explicit rule of transitivity are trivial.

We propose a solution to this problem. By using mixed induction and coinduction we define an inference system for subtyping which combines the advantages of coinduction with the convenience of an explicit rule of transitivity. The definition uses coinduction for the structural rules, and induction for the rule of transitivity. We also discuss under what conditions this technique can be used when defining other inference systems.

The developments presented in the paper have been mechanised using Agda, a dependently typed programming language and proof assistant.

1 Introduction

Coinduction and corecursion are useful techniques for defining and reasoning about things which are potentially infinite, including streams and other (potentially) infinite data types (Coquand 1994; Giménez 1996; Turner 2004), process congruences (Milner 1990), congruences for functional programs (Gordon 1999), closures (Milner and Tofte 1991), semantics for divergence of programs (Cousot and Cousot 1992; Hughes and Moran 1995; Leroy and Grall 2009; Nakata and Uustalu 2009), and subtyping relations for recursive types (Brandt and Henglein 1998; Gapeyev et al. 2002).

However, the use of coinduction can lead to values which are "too infinite". For instance, a non-trivial binary relation defined as a coinductive inference system cannot include the rule of transitivity, because a coinductive reading of transitivity would imply that every element is related to every other (to see this, build an infinite derivation consisting solely of uses of transitivity). As pointed out by Gapeyev et al. (2002) this is unfortunate, because without transitivity, conceptually unrelated rules may have to be merged or otherwise modified in order to ensure that transitivity can be proved as a derived property. Gapeyev et al. give the example of subtyping for records, where a dedicated rule of transitivity ensures that one can give separate rules for depth subtyping (which states that a record field type can be replaced by a subtype), width subtyping (which states that new fields can be added to a record), and permutation of record fields.

We propose a solution to this problem. The problem stems from a *coinductive* reading of transitivity, and it can be solved by reading the rule of transitivity *inductively*, and only using coinduction where it is necessary. We illustrate this idea by using mixed induction and coinduction to define a subtyping relation for recursive types; such relations have been studied repeatedly in the past (Amadio and Cardelli 1993; Kozen et al. 1995; Brandt and Henglein 1998, and others). The rule which defines when a function type is a subtype of another is defined coinductively, following Brandt and Henglein (1998) and Gapeyev et al. (2002), while the rule of transitivity is defined inductively.

The technique of mixing induction and coinduction has been known for a long time (Park 1980; Barwise 1989; Raffalli 1994; Giménez 1996; Hensel and Jacobs 1997; Müller et al. 1999; Barthe et al. 2004; Levy 2006; Bradfield and Stirling 2007; Abel 2007; Hancock et al. 2009), but we feel that it deserves to be more well-known in the programming language community. We also believe that the approach to coinduction used in the paper, due to Coquand (1994), deserves more attention: following the Curry-Howard correspondence the coinductive definition and proof principles both take the form of guarded corecursion for (potentially indexed) lazy data types.

The main developments in the paper have been formalised using the dependently typed, total[1] functional programming language Agda (Norell 2007; Agda Team 2010), which provides good support for mixed induction and coinduction in the style mentioned above. The source code is at the time of writing available to download (Danielsson 2010a).

The rest of the paper is structured as follows: Section 2 gives an introduction to induction and coinduction in the context of Agda. Section 3 defines a small language of recursive types, and Section 4 defines a subtyping relation for this language by viewing the types as potentially infinite trees. Section 5 defines an equivalent, declarative subtyping relation using mixed induction and coinduction, and Section 6 compares this definition to another equivalent definition, given by Brandt and Henglein (1998). Finally Section 7 discusses a potential pitfall associated with the technique we propose, and Section 8 concludes.

2 Induction and Coinduction

This section gives a brief introduction to induction and coinduction, with an emphasis on how these concepts are realised in Agda. For more formal accounts of induction and coinduction see, for instance, the theses of Hagino (1987) and Mendler (1988).

2.1 Induction

Let us start with a simple inductive definition. In Agda the type of *finite* lists can be defined as follows:

[1] Agda is an experimental system. The meta-theory has not been formalised, and the type checker has not been proved bug-free, so take phrases such as "total" with a grain of salt.

$$
\begin{array}{l}
\textbf{data}\ \mathit{List}\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where} \\
\quad [\,]\quad\ :\ \mathit{List}\ A \\
\quad \_{::}\_\ :\ A\ \rightarrow\ \mathit{List}\ A\ \rightarrow\ \mathit{List}\ A
\end{array}
$$

This states that $\mathit{List}\ A$ is a type (or Set) with two constructors, $[\,]$ of type $\mathit{List}\ A$ and $\_{::}\_$ of type $A \rightarrow \mathit{List}\ A \rightarrow \mathit{List}\ A$. The constructor $\_{::}\_$ is an infix operator; the underscores mark the argument positions. The type $\mathit{List}\ A$ is isomorphic to the least fixpoint $\mu X.\ 1 + A \times X$ in the category of types and total functions.[2]

Agda has a termination checker which ensures that all code is terminating (or productive, see below). It is assisted by other checkers which ensure that data types are strictly positive, and not too large. The termination checker allows lists to be destructed using structural recursion:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{map}\ :\ \{A\ B\ :\ \mathit{Set}\}\ \rightarrow\ (A\ \rightarrow\ B)\ \rightarrow\ \mathit{List}\ A\ \rightarrow\ \mathit{List}\ B} \\
\mathit{map}\ f\ [\,] & = & [\,] \\
\mathit{map}\ f\ (x :: \mathit{xs}) & = & f\ x :: \mathit{map}\ f\ \mathit{xs}
\end{array}
$$

The use of braces in $\{A\ B\ :\ \mathit{Set}\} \rightarrow \ldots$ means that the two type arguments A and B are *implicit*; they do not need to be given explicitly if Agda can infer them. Note that in this context $A\ B$ is not an application, it is a sequence of variables.

2.2 Coinduction

If we want to have infinite lists, or streams, we can use the following coinductive definition instead (note that constructors, such as $\_{::}\_$, can be overloaded in Agda):

$$
\begin{array}{l}
\textbf{data}\ \mathit{Stream}\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where} \\
\quad \_{::}\_\ :\ A\ \rightarrow\ \infty\ (\mathit{Stream}\ A)\ \rightarrow\ \mathit{Stream}\ A
\end{array}
$$

The type $\mathit{Stream}\ A$ is isomorphic to the greatest fixpoint $\nu X.\ A \times X$. The type function $\infty\ :\ \mathit{Set} \rightarrow \mathit{Set}$ marks its argument as being coinductive. It is analogous to the suspension type constructors which are sometimes used to implement non-strictness in strict languages (Wadler et al. 1998), and comes with a force function and a delay constructor:

$$
\begin{array}{lllllll}
\flat & : & \{A\ :\ \mathit{Set}\} & \rightarrow & \infty\ A & \rightarrow & A \\
\sharp\_ & : & \{A\ :\ \mathit{Set}\} & \rightarrow & A & \rightarrow & \infty\ A
\end{array}
$$

The constructor $\sharp\_$ is a tightly binding prefix operator. Ordinary function application binds tighter, though.

Values of coinductive types can be constructed using *guarded* corecursion (Coquand 1994):

[2] At the time of writing this is not exactly true in Agda (Danielsson and Altenkirch 2009), but the difference between $\mathit{List}\ A$ and the fixpoint is irrelevant for the purposes of this paper. Similar considerations apply to greatest fixpoints.

$$\begin{aligned} &map_{\mathrm{S}} \;:\; \{A\ B \;:\; Set\} \to (A \to B) \to Stream\ A \to Stream\ B \\ &map_{\mathrm{S}}\ f\ (x :: xs) \;=\; f\ x :: {}^{\sharp}\ map_{\mathrm{S}}\ f\ ({}^{\flat}\ xs) \end{aligned}$$

The definition of map_{S} is accepted by Agda's termination checker because the corecursive call is guarded by ${}^{\sharp}\_$, without any non-constructor function between the left-hand side and the corecursive call. This syntactic notion of guardedness ensures that corecursive definitions are *productive*: even if the value being constructed is infinite, the next constructor can always be computed in a finite number of steps.

It may also be instructive to see (attempted) definitions which are not accepted:

$$\begin{aligned} &bad \;:\; Stream\ \mathbb{N} \\ &bad \;=\; \mathsf{zero} :: {}^{\sharp}\ tail\ bad \end{aligned} \qquad\qquad \begin{aligned} &nats \;:\; Stream\ \mathbb{N} \\ &nats \;=\; \mathsf{zero} :: {}^{\sharp}\ map_{\mathrm{S}}\ \mathsf{suc}\ nats \end{aligned}$$

Both definitions are rejected because they are not guarded, but only the first one is non-productive; *nats* uniquely specifies the stream of natural numbers, but is rejected by the termination checker because it does not satisfy the syntactic criterion imposed by Agda.

2.3 Coinductive Relations

Let us now consider a coinductively defined *relation*: stream equality, also known as bisimilarity. Two streams are equal if they have identical heads and their tails are equal (coinductively):

$$\frac{{}^{\flat}\ xs \approx {}^{\flat}\ ys}{x :: xs \approx x :: ys}\ \text{(coinductive)}$$

This inference system can be represented using an indexed data type:

$$\begin{aligned} &\mathbf{data}\ \_\!\approx\!\_\ \{A \;:\; Set\} \;:\; Stream\ A \to Stream\ A \to Set\ \mathbf{where} \\ &\quad \_\!::\!\_ \;:\; (x \;:\; A)\ \{xs\ ys \;:\; \infty\ (Stream\ A)\} \to \infty\ ({}^{\flat}\ xs \approx {}^{\flat}\ ys) \to \\ &\qquad\qquad x :: xs \approx x :: ys \end{aligned}$$

Some remarks on this definition may be useful:

– The elements of the type $xs \approx ys$ are *proofs* witnessing the equality of xs and ys. Agda does not make a distinction between proofs and programs, and the termination checker ensures productivity of both kinds of definition.
– *Dependent* function spaces ($(x \;:\; A) \to B$ where x can occur in B) are used to set up dependencies of types on values.
– The first occurrence of the type constructor ∞ just reflects the fact that the second argument to the stream constructor $\_\!::\!\_$ is delayed. The second occurrence is necessary to be able to construct infinite equality proofs; if we had omitted it the relation would have been empty.

– We overload the constructor $\_{::}\_$ so that it stands both for the "cons" function for streams, and for the proof that cons preserves equality. The constructors can be disambiguated based on type information.

Elements of coinductively defined relations can be constructed using corecursion. As an example, let us prove the map-iterate property (Gibbons and Hutton 2005):

$$\mathit{map}_{\mathrm{S}}\ f\ (\mathit{iterate}\ f\ x)\ \approx\ \mathit{iterate}\ f\ (f\ x).$$

The function *iterate* repeatedly applies a function to a seed element and collects the results in a stream:

$$\mathit{iterate}\ f\ x\ =\ x :: {}^{\sharp}\,(f\ x :: {}^{\sharp}\,(f\ (f\ x) :: \ldots)).$$

The function is defined corecursively:

$$\begin{aligned}&\mathit{iterate}\ :\ \{A\ :\ \mathit{Set}\}\ \to\ (A\ \to\ A)\ \to\ A\ \to\ \mathit{Stream}\ A\\&\mathit{iterate}\ f\ x\ =\ x :: {}^{\sharp}\ \mathit{iterate}\ f\ (f\ x)\end{aligned}$$

The map-iterate property can be proved using guarded corecursion (the term *guarded coinduction* could also be used):

$$\begin{aligned}&\mathit{map\text{-}iterate}\ :\ \{A\ :\ \mathit{Set}\}\ (f\ :\ A\ \to\ A)\ (x\ :\ A)\ \to\\&\qquad\qquad\qquad \mathit{map}_{\mathrm{S}}\ f\ (\mathit{iterate}\ f\ x)\ \approx\ \mathit{iterate}\ f\ (f\ x)\\&\mathit{map\text{-}iterate}\ f\ x\ =\ f\ x :: {}^{\sharp}\ \mathit{map\text{-}iterate}\ f\ (f\ x)\end{aligned}$$

To see how this proof works, consider how it can be built up step by step (as in an interactive Agda session):

$$\mathit{map\text{-}iterate}\ f\ x\ = ?$$

The type of the *goal* ? is $\mathit{map}_{\mathrm{S}}\ f\ (\mathit{iterate}\ f\ x)\ \approx\ \mathit{iterate}\ f\ (f\ x)$. Agda types should always be read up to normalisation, so this is equivalent to[3]

$$f\ x :: {}^{\sharp}\ \mathit{map}_{\mathrm{S}}\ f\ ({}^{\flat}\ ({}^{\sharp}\ \mathit{iterate}\ f\ (f\ x)))\ \approx\ f\ x :: {}^{\sharp}\ \mathit{iterate}\ f\ (f\ (f\ x)).$$

(Note that normalisation does not involve reduction under ${}^{\sharp}\_$, and that ${}^{\flat}\ ({}^{\sharp}\ x)$ reduces to x.) This type matches the result type of the equality constructor $\_{::}\_$, so we can refine the goal:

$$\mathit{map\text{-}iterate}\ f\ x\ =\ f\ x :: ?$$

The new goal type is

$$\infty\ \big(\mathit{map}_{\mathrm{S}}\ f\ (\mathit{iterate}\ f\ (f\ x))\ \approx\ \mathit{iterate}\ f\ (f\ (f\ x))\big),$$

so the proof can be finished by an application of the coinductive hypothesis under the guarding constructor ${}^{\sharp}\_$:

$$\mathit{map\text{-}iterate}\ f\ x\ =\ f\ x :: {}^{\sharp}\ \mathit{map\text{-}iterate}\ f\ (f\ x)$$

[3] This is a simplification of the current behaviour of Agda.

2.4 Mixed Induction and Coinduction

The types above are either inductive or coinductive. Let us now discuss a type which uses both induction and coinduction. Hancock et al. (2009) define a language of stream processors, representing functions of type *Stream A* → *Stream B*, using a nested fixpoint: $\nu Y.\mu X.\ B \times Y + (A \to X)$. We can represent this fixpoint in Agda as follows:

```
data SP (A B : Set) : Set where
  put : B → ∞ (SP A B) → SP A B
  get : (A → SP A B)    → SP A B
```

The stream processor put b sp outputs b, and continues processing according to sp. The processor get f reads one element a from the input stream, and continues processing according to $f\ a$. In the case of put the recursive argument is coinductive, so it is fine to output an infinite number of elements, whereas in the case of get the recursive argument is inductive, which means that one can only read a finite number of elements before writing the next one. This ensures that the output stream can be generated productively.

We can implement a simple stream processor which copies the input to the output as follows:

```
copy : {A : Set} → SP A A
copy = get (λ a → put a (♯ copy))
```

This definition is guarded. Note that *copy* contains an infinite number of get constructors. This is fine, even though get's argument is inductive, because there is never a stretch of infinitely many get constructors without an intervening delay constructor (♯_). On the other hand, the following definition of a sink is not guarded, and is not accepted by Agda:

```
sink : {A B : Set} → SP A B
sink = get (λ _ → sink)
```

As another example we can compute the semantics of a stream processor:

```
⟦_⟧ : {A B : Set} → SP A B → Stream A → Stream B
⟦ put b sp ⟧ as       = b :: ♯ (⟦ ♭ sp ⟧ as)
⟦ get f    ⟧ (a :: as) = ⟦ f a ⟧ (♭ as)
```

(⟦_⟧ is a mixfix operator.) This definition uses a lexicographic combination of guarded corecursion and higher-order structural recursion (see Section 2.5). In the first clause the corecursive call is guarded. In the second clause it "preserves guardedness" (it takes place under zero coinductive constructors rather than one), and the first argument is structurally smaller.

Note that ⟦_⟧ could not have been implemented if *SP A B* had been defined purely coinductively (because then *sink* could be implemented with B equal to the empty type). By using both induction and coinduction in the definition we rule out certain stream processors which would otherwise have been accepted, and in return we can implement functions like ⟦_⟧.

2.5 A Criterion for Totality

Let us now make things more precise by giving a more detailed explanation of Agda's criterion for accepting a function as being total. The results in the paper do not depend on the exact criterion used by Agda, so we only give a conservative approximation of what is currently implemented. The description below is based on the termination checker foetus (Abel and Altenkirch 2002), extended with support for guarded coinduction based on an idea due to Andreas Abel (personal communication).

First we collect some information about the program. For every left-hand side $f\ p_1\ \ldots\ p_m$ and function call $g\ e_1\ \ldots\ e_n$ in the corresponding right-hand side the following information is recorded:

Argument structure. For every pair (p_i, e_j) it is noted if the argument e_j is structurally strictly smaller (denoted by $<$) or equal to $(=)$ the pattern p_i. If neither case applies, then we use the notation ?. Note that x is not structurally smaller than $\sharp\ x$, and that $f\ x$ is strictly smaller than $\mathsf{c}\ f$, for an inductive constructor c.

Guardedness. It is also noted whether the call is guarded by constructors, at least one of which is coinductive $(<)$; or whether guardedness is preserved, i.e. if the call is guarded by inductive constructors $(=)$.

The next step is to combine the information about individual calls into information about all the call paths from one function to itself. We use the notation $(g \mid a_1\ \ldots\ a_n)$ to describe the information computed for a call path; here g is the guardedness information, and a_i describes how the i-th argument is changed. In the case of the function $[\![\_]\!]$ from Section 2.4 we get that there are three kinds of call paths:

1. $(< \mid = = ? =)$, which corresponds to the first recursive call;
2. $(= \mid = = < ?)$, which corresponds to the second recursive call; and
3. $(< \mid = = ? ?)$ for call paths which involve both recursive calls.

Finally we can give the criterion for totality: a function is accepted as total if there is some lexicographic combination of the components for which every call path is strictly decreasing. In the case of $[\![\_]\!]$ it suffices to combine the guardedness with the information about the third argument (the stream processor).

As noted by Danielsson and Altenkirch (2009, Section 7.1) the criterion above works best if all fixpoints have the form $\nu Y.\mu X.\ F\ X\ Y$ (for suitable values of F); we have not yet found a good way to incorporate fixpoints of the form $\mu X.\nu Y.\ F\ X\ Y$. However, this issue does not affect the examples in this paper.

2.6 Relations Using Mixed Induction and Coinduction

As a final example we define a relation using mixed induction and coinduction. Capretta (2005) defines the partiality monad, which can be used to represent potentially non-terminating computations, as follows:

```
data _ ^ν (A : Set) : Set where
  return : A         → A ^ν
  step   : ∞ (A ^ν) → A ^ν
```

The constructor return returns a result, and step postpones a computation. Non-termination is represented as an infinitely postponed computation:

```
⊥ : {A : Set} → A ^ν
⊥ = step (♯ ⊥)
```

A natural definition of equality for partial computations is weak bisimilarity (viewing step as a silent transition):[4]

```
data _≅_ : A ^ν → A ^ν → Set where
  return :                        return v ≅ return v
  step   : ∞ (♭ x ≅ ♭ y) → step x   ≅ step y
  stepʳ  :      x ≅ ♭ y  →      x   ≅ step y
  stepˡ  :    ♭ x ≅   y  → step x   ≅      y
```

This is basically the congruence generated by return and step, but allowing for finite differences in delay. Note that the requirement of *finite* differences in delay is captured by the use of induction for $\mathsf{step}^{\mathrm{r}}$ and $\mathsf{step}^{\mathrm{l}}$, while the use of coinduction for step is necessary to be able to prove that the relation is reflexive.

3 Recursive Types

Brandt and Henglein (1998) define the following language of recursive types:

$$\sigma, \tau ::= \bot \mid \top \mid X \mid \sigma \rightharpoonup \tau \mid \mu X.\, \sigma \rightharpoonup \tau$$

Here $\bot$ and $\top$ are the least and greatest types, respectively, X is a variable, $\sigma \rightharpoonup \tau$ is a function type, and $\mu X.\ \sigma \rightharpoonup \tau$ is a fixpoint, with bound variable X. (The body of the fixpoint is required to be a function type, so types like $\mu X.X$ are ruled out.) The intention is that a fixpoint $\mu X.\sigma \rightharpoonup \tau$ should be equivalent to its unfolding $(\sigma \rightharpoonup \tau)[X := \mu X.\ \sigma \rightharpoonup \tau]$. It would be unproblematic to extend the language with other type constructors, such as products and sums.

The language above can be represented in Agda as follows:

```
data Ty (n : ℕ) : Set where
  ⊥     : Ty n
  ⊤     : Ty n
  var   : Fin n → Ty n
  _⇀_   : Ty n       → Ty n       → Ty n
  μ_⇀_  : Ty (1 + n) → Ty (1 + n) → Ty n
```

[4] In order to reduce clutter the declarations of implicit arguments have been omitted in the remainder of the paper.

Here variables are represented using de Bruijn indices: $Ty\ n$ represents types with at most n free variables, and $Fin\ n$ is a type representing the first n natural numbers. Substitution can also be defined; $\sigma\ [\ \tau\ ]$ is the capture-avoiding substitution of τ for variable 0 in σ:

$$\_[\_] \ :\ Ty\ (1 + n)\ \to\ Ty\ n\ \to\ Ty\ n$$

The following function unfolds a fixpoint one step:

$$\begin{aligned}&unfold\langle\mu\_{\multimap}\_\rangle\ :\ Ty\ (1+n)\ \to\ Ty\ (1+n)\ \to\ Ty\ n\\ &unfold\langle\mu\,\sigma \multimap \tau\,\rangle\ =\ (\sigma \multimap \tau)\ [\ \mu\ \sigma \multimap \tau\ ]\end{aligned}$$

(Note that $\mu\_{\multimap}\_$, $\_[\_]$ and $unfold\langle\mu\_{\multimap}\_\rangle$ are all mixfix operators which take two arguments.)

4 Subtyping via Trees

A natural definition of subtyping goes via subtyping for potentially infinite trees (Gapeyev et al. 2002):

$$\begin{array}{lll}\multicolumn{3}{l}{\textbf{data}\ Tree\ (n\ :\ \mathbb{N})\ :\ Set\ \textbf{where}}\\ \quad\bot & : & Tree\ n\\ \quad\top & : & Tree\ n\\ \quad\mathsf{var} & : & Fin\ n\ \to\ Tree\ n\\ \quad\_{\multimap}\_ & : & \infty\ (Tree\ n)\ \to\ \infty\ (Tree\ n)\ \to\ Tree\ n\end{array}$$

The subtyping relation for trees can be given coinductively as follows:

$$\begin{array}{lll}\multicolumn{3}{l}{\textbf{data}\ \_\leqslant_{\text{Tree}}\_\ :\ Tree\ n\ \to\ Tree\ n\ \to\ Set\ \textbf{where}}\\ \quad\bot & : & \bot\ \leqslant_{\text{Tree}}\ \tau\\ \quad\top & : & \sigma\ \leqslant_{\text{Tree}}\ \top\\ \quad\mathsf{var} & : & \mathsf{var}\ x\ \leqslant_{\text{Tree}}\ \mathsf{var}\ x\\ \quad\_{\multimap}\_ & : & \infty\ ({}^\flat\,\tau_1\ \leqslant_{\text{Tree}}\ {}^\flat\,\sigma_1)\ \to\ \infty\ ({}^\flat\,\sigma_2\ \leqslant_{\text{Tree}}\ {}^\flat\,\tau_2)\ \to\\ & & \sigma_1 \multimap \sigma_2\ \leqslant_{\text{Tree}}\ \tau_1 \multimap \tau_2\end{array}$$

Note the contravariant treatment of the codomain of the function space. Note also that the constructors of *Tree* are overloaded—repeatedly—in order to reduce clutter.

The semantics of a recursive type can be given in terms of its unfolding as a potentially infinite tree:

$$\begin{array}{lll}\multicolumn{3}{l}{[\![\_]\!]\ :\ Ty\ n\ \to\ Tree\ n}\\ [\![\ \bot\ ]\!] & = & \bot\\ [\![\ \top\ ]\!] & = & \top\\ [\![\ \mathsf{var}\ x\ ]\!] & = & \mathsf{var}\ x\\ [\![\quad \sigma \multimap \tau\ ]\!] & = & {}^\sharp\,[\![\ \sigma\ ]\!] \multimap {}^\sharp\,[\![\ \tau\ ]\!]\\ [\![\ \mu\,\sigma \multimap \tau\ ]\!] & = & {}^\sharp\,[\![\ \sigma\ [\ \chi\ ]\ ]\!] \multimap {}^\sharp\,[\![\ \tau\ [\ \chi\ ]\ ]\!]\\ & & \textbf{where}\ \chi\ =\ \mu\,\sigma \multimap \tau\end{array}$$

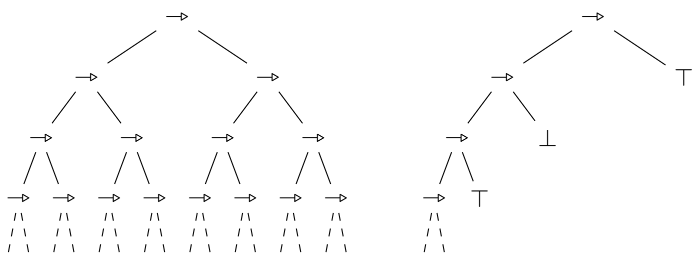


Fig. 1. The first levels of the infinite trees corresponding to the types $\mu X.\ X \rightharpoonup X$ and $\mu X.\ (X \rightharpoonup \bot) \rightharpoonup \top$

The subtyping relation for types can then be defined by combining $\_\leqslant_{\text{Tree}}\_$ and $[\![\_]\!]$:

$$\begin{array}{l} \_\leqslant_{\text{Type}}\_ \ : \ \mathit{Ty}\ n \ \rightarrow \ \mathit{Ty}\ n \ \rightarrow \ \mathit{Set} \\ \sigma \ \leqslant_{\text{Type}} \tau \ = \ [\![\ \sigma\ ]\!] \ \leqslant_{\text{Tree}} [\![\ \tau\ ]\!] \end{array}$$

As a simple example, consider the following two types, $\sigma \ = \ \mu X.\ X \rightharpoonup X$ and $\tau \ = \ \mu X.\ (X \rightharpoonup \bot) \rightharpoonup \top$:

$$\begin{array}{ll} \sigma \ : \ \mathit{Ty}\ 0 & \tau \ : \ \mathit{Ty}\ 0 \\ \sigma \ = \ \mu\ \mathsf{var\ zero} \rightharpoonup \mathsf{var\ zero} & \tau \ = \ \mu\ (\mathsf{var\ zero} \rightharpoonup \bot) \rightharpoonup \top \end{array}$$

The first few levels of the infinite trees corresponding to these types can be seen in Fig. 1. It is straightforward to show that σ is a subtype of τ using a corecursive proof:

$$\begin{array}{l} \sigma{\leqslant}\tau \ : \ \sigma \ \leqslant_{\text{Type}} \tau \\ \sigma{\leqslant}\tau \ = \ ^\sharp\ (^\sharp\ \sigma{\leqslant}\tau \rightharpoonup {}^\sharp\ \bot) \rightharpoonup {}^\sharp\ \top \end{array}$$

(Note that $\sigma{\leqslant}\tau$ is an identifier and not a compound expression; almost any character string which does not contain whitespace can be used as an identifier.)

Amadio and Cardelli (1993) also define subtyping for recursive types by going via potentially infinite trees, but they define a subtyping relation *inductively* on finite trees, and state that an infinite tree σ is a subtype of another tree τ when every finite approximation (of a certain kind) of σ is a subtype of the corresponding approximation of τ. It is easy to show that this definition, as adapted by Brandt and Henglein (1998), is equivalent to the one given above. One direction of the proof uses induction on the depth of approximation, and the other constructs elements of $\sigma \ \leqslant_{\text{Type}} \ \tau$ corecursively; see the code which accompanies the paper (Danielsson 2010a).

5 Subtyping Using Mixed Induction and Coinduction

Subtyping can also be defined directly, without going via trees. The following definition is inspired by one given by Brandt and Henglein (1998), see Section 6:

$$\begin{array}{lll}
\textbf{data}\ \_\leqslant\_ & : \mathit{Ty}\ n \to \mathit{Ty}\ n \to \mathit{Set}\ \textbf{where} & \\
\quad \bot & : \bot \leqslant \tau & \\
\quad \top & : \sigma \leqslant \top & \\
\quad \_\multimap\_ & : \infty\,(\tau_1 \leqslant \sigma_1) \to \infty\,(\sigma_2 \leqslant \tau_2) \to \sigma_1 \multimap \sigma_2 \leqslant \tau_1 \multimap \tau_2 & \\
\quad \mathsf{unfold} & : \mu\,\tau_1 \multimap \tau_2 \leqslant \mathit{unfold}\langle \mu\,\tau_1 \multimap \tau_2 \rangle & \\
\quad \mathsf{fold} & : \mathit{unfold}\langle \mu\,\tau_1 \multimap \tau_2 \rangle \leqslant \mu\,\tau_1 \multimap \tau_2 & \\
\quad \mathsf{refl} & : \tau \leqslant \tau & \\
\quad \mathsf{trans} & : \tau_1 \leqslant \tau_2 \to \tau_2 \leqslant \tau_3 \to \tau_1 \leqslant \tau_3 &
\end{array}$$

Note that the structural rules ($\bot$, $\top$, $\_\multimap\_$) are defined coinductively, while the other rules, most importantly trans, are defined inductively. Note also that the inclusion of refl and trans is essential; if either constructor is removed we get a different relation.

Now, if we can prove that the relation $\_\leqslant\_$ is equivalent to $\_\leqslant_{\text{Type}}\_$ (and thus also equivalent to Amadio and Cardelli's relation), then we have showed what we set out to show: that coinduction and the rule of transitivity can be combined. We can prove completeness by a simple application of guarded corecursion (omitted here):

$$\mathit{complete} : \sigma \leqslant_{\text{Type}} \tau \to \sigma \leqslant \tau$$

The soundness proof is a little more tricky. The following lemmas are easy to prove:

$$\begin{array}{ll}
\mathit{unfold}_{\text{Type}} & : \mu\,\tau_1 \multimap \tau_2 \leqslant_{\text{Type}} \mathit{unfold}\langle \mu\,\tau_1 \multimap \tau_2 \rangle \\
\mathit{fold}_{\text{Type}} & : \mathit{unfold}\langle \mu\,\tau_1 \multimap \tau_2 \rangle \leqslant_{\text{Type}} \mu\,\tau_1 \multimap \tau_2 \\
\mathit{refl}_{\text{Type}} & : \tau \leqslant_{\text{Type}} \tau \\
\mathit{trans}_{\text{Type}} & : \tau_1 \leqslant_{\text{Type}} \tau_2 \to \tau_2 \leqslant_{\text{Type}} \tau_3 \to \tau_1 \leqslant_{\text{Type}} \tau_3
\end{array}$$

Using these lemmas one might think that the following should be accepted as a soundness proof:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{sound} : \sigma \leqslant \tau \to \sigma \leqslant_{\text{Type}} \tau} \\
\mathit{sound}\ \bot & = \bot \\
\mathit{sound}\ \top & = \top \\
\mathit{sound}\ (\tau_1{\leqslant}\sigma_1 \multimap \sigma_2{\leqslant}\tau_2) & = \sharp\ \mathit{sound}\ (\flat\ \tau_1{\leqslant}\sigma_1) \multimap \sharp\ \mathit{sound}\ (\flat\ \sigma_2{\leqslant}\tau_2) \\
\mathit{sound}\ \mathsf{unfold} & = \mathit{unfold}_{\text{Type}} \\
\mathit{sound}\ \mathsf{fold} & = \mathit{fold}_{\text{Type}} \\
\mathit{sound}\ \mathsf{refl} & = \mathit{refl}_{\text{Type}} \\
\mathit{sound}\ (\mathsf{trans}\ \tau_1{\leqslant}\tau_2\ \tau_2{\leqslant}\tau_3) & = \mathit{trans}_{\text{Type}}\ (\mathit{sound}\ \tau_1{\leqslant}\tau_2)\ (\mathit{sound}\ \tau_2{\leqslant}\tau_3)
\end{array}$$

However, consider the case for trans. The arguments to the recursive calls are structurally smaller than the inputs, but $\mathit{trans}_{\text{Type}}$ is not a constructor, so guardedness is not preserved. The proof is productive (given a suitable definition of $\mathit{trans}_{\text{Type}}$), but Agda's termination checker cannot see this.

In the absence of improved termination checking for Agda we provide a workaround, using a technique described by Danielsson (2010b). If $\mathit{trans}_{\text{Type}}$

had been a constructor then the definition of *sound* would have been accepted, and this observation can be used to rescue the proof. First we define a variant of $\_\leqslant_{\text{Tree}}\_$ which includes an extra inductive constructor, trans:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \_\leqslant_{\text{TreeP}}\_ \ :\ \mathit{Tree}\ n\ \rightarrow\ \mathit{Tree}\ n\ \rightarrow\ \mathit{Set}\ \textbf{where}} \\
\quad\bot & : & \bot \ \leqslant_{\text{TreeP}}\ \tau \\
\quad\top & : & \sigma \ \leqslant_{\text{TreeP}}\ \top \\
\quad\mathsf{var} & : & \mathsf{var}\ x\ \leqslant_{\text{TreeP}}\ \mathsf{var}\ x \\
\quad\_\multimap\_ & : & \infty\ ({}^\flat\ \tau_1\ \leqslant_{\text{TreeP}}\ {}^\flat\ \sigma_1)\ \rightarrow\ \infty\ ({}^\flat\ \sigma_2\ \leqslant_{\text{TreeP}}\ {}^\flat\ \tau_2)\ \rightarrow \\
 & & \sigma_1 \multimap \sigma_2\ \leqslant_{\text{TreeP}}\ \tau_1 \multimap \tau_2 \\
\quad\mathsf{trans} & : & \tau_1\ \leqslant_{\text{TreeP}}\ \tau_2\ \rightarrow\ \tau_2\ \leqslant_{\text{TreeP}}\ \tau_3\ \rightarrow\ \tau_1\ \leqslant_{\text{TreeP}}\ \tau_3
\end{array}$$

The letter P stands for “program”; this type defines a small language of equality proof programs. It is easy to turn proofs into proof programs corecursively:

$$\ulcorner\_\urcorner\ :\ \sigma\ \leqslant_{\text{Tree}}\ \tau\ \rightarrow\ \sigma\ \leqslant_{\text{TreeP}}\ \tau$$

We can now write a guarded proof *program* which “proves” soundness:

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{sound}_{\text{P}}\ :\ \sigma\ \leqslant\ \tau\ \rightarrow\ [\![\ \sigma\ ]\!]\ \leqslant_{\text{TreeP}}\ [\![\ \tau\ ]\!]} \\
\mathit{sound}_{\text{P}}\ \bot & = & \bot \\
\mathit{sound}_{\text{P}}\ \top & = & \top \\
\mathit{sound}_{\text{P}}\ (\tau_1{\leqslant}\sigma_1 \multimap \sigma_2{\leqslant}\tau_2) & = & \sharp\ \mathit{sound}_{\text{P}}\ ({}^\flat\ \tau_1{\leqslant}\sigma_1) \multimap \sharp\ \mathit{sound}_{\text{P}}\ ({}^\flat\ \sigma_2{\leqslant}\tau_2) \\
\mathit{sound}_{\text{P}}\ \mathsf{unfold} & = & \ulcorner\ \mathit{unfold}_{\text{Type}}\ \urcorner \\
\mathit{sound}_{\text{P}}\ \mathsf{fold} & = & \ulcorner\ \mathit{fold}_{\text{Type}}\ \urcorner \\
\mathit{sound}_{\text{P}}\ \mathsf{refl} & = & \ulcorner\ \mathit{refl}_{\text{Type}}\ \urcorner \\
\mathit{sound}_{\text{P}}\ (\mathsf{trans}\ \tau_1{\leqslant}\tau_2\ \tau_2{\leqslant}\tau_3) & = & \mathsf{trans}\ (\mathit{sound}_{\text{P}}\ \tau_1{\leqslant}\tau_2)\ (\mathit{sound}_{\text{P}}\ \tau_2{\leqslant}\tau_3)
\end{array}$$

If we can also find a way to turn proof programs into proofs, productively, then we are done. We start by defining a type of weak head normal forms (WHNFs) for the proof programs:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \_\leqslant_{\text{TreeW}}\_ \ :\ \mathit{Tree}\ n\ \rightarrow\ \mathit{Tree}\ n\ \rightarrow\ \mathit{Set}\ \textbf{where}} \\
\quad\bot & : & \bot \ \leqslant_{\text{TreeW}}\ \tau \\
\quad\top & : & \sigma \ \leqslant_{\text{TreeW}}\ \top \\
\quad\mathsf{var} & : & \mathsf{var}\ x\ \leqslant_{\text{TreeW}}\ \mathsf{var}\ x \\
\quad\_\multimap\_ & : & {}^\flat\ \tau_1\ \leqslant_{\text{TreeP}}\ {}^\flat\ \sigma_1\ \rightarrow\ {}^\flat\ \sigma_2\ \leqslant_{\text{TreeP}}\ {}^\flat\ \tau_2\ \rightarrow \\
 & & \sigma_1 \multimap \sigma_2\ \leqslant_{\text{TreeW}}\ \tau_1 \multimap \tau_2
\end{array}$$

Note that the arguments to $\_\multimap\_$ are *programs*, not WHNFs. One can prove by simple case analysis that $\_\leqslant_{\text{TreeW}}\_$ is transitive:

$$\mathit{trans}_{\text{TreeW}}\ :\ \tau_1\ \leqslant_{\text{TreeW}}\ \tau_2\ \rightarrow\ \tau_2\ \leqslant_{\text{TreeW}}\ \tau_3\ \rightarrow\ \tau_1\ \leqslant_{\text{TreeW}}\ \tau_3$$

From this result it follows by structural recursion that programs can be turned into WHNFs:

$$\begin{array}{lcl}
\mathit{whnf} \;:\; \sigma \leqslant_{\mathrm{TreeP}} \tau \;\rightarrow\; \sigma \leqslant_{\mathrm{TreeW}} \tau & & \\
\mathit{whnf}\ \bot & = & \bot \\
\mathit{whnf}\ \top & = & \top \\
\mathit{whnf}\ \mathsf{var} & = & \mathsf{var} \\
\mathit{whnf}\ (\tau_1{\leqslant}\sigma_1 \rightarrowtriangle \sigma_2{\leqslant}\tau_2) & = & {}^{\flat}\ \tau_1{\leqslant}\sigma_1 \rightarrowtriangle {}^{\flat}\ \sigma_2{\leqslant}\tau_2 \\
\mathit{whnf}\ (\mathsf{trans}\ \tau_1{\leqslant}\tau_2\ \tau_2{\leqslant}\tau_3) & = & \mathit{trans}_{\mathrm{TreeW}}\ (\mathit{whnf}\ \tau_1{\leqslant}\tau_2)\ (\mathit{whnf}\ \tau_2{\leqslant}\tau_3)
\end{array}$$

The following mutually recursive functions then turn proof programs into "actual" proofs by using the *whnf* function repeatedly:

$$\begin{array}{lcl}
\llbracket\_\rrbracket_{\mathrm{W}} \;:\; \sigma \leqslant_{\mathrm{TreeW}} \tau \;\rightarrow\; \sigma \leqslant_{\mathrm{Tree}} \tau & & \\
\llbracket\ \bot\ \rrbracket_{\mathrm{W}} & = & \bot \\
\llbracket\ \top\ \rrbracket_{\mathrm{W}} & = & \top \\
\llbracket\ \mathsf{var}\ \rrbracket_{\mathrm{W}} & = & \mathsf{var} \\
\llbracket\ \tau_1{\leqslant}\sigma_1 \rightarrowtriangle \sigma_2{\leqslant}\tau_2\ \rrbracket_{\mathrm{W}} & = & {}^{\sharp}\ \llbracket\ \tau_1{\leqslant}\sigma_1\ \rrbracket_{\mathrm{P}} \rightarrowtriangle {}^{\sharp}\ \llbracket\ \sigma_2{\leqslant}\tau_2\ \rrbracket_{\mathrm{P}}
\end{array}$$

$$\begin{array}{l}
\llbracket\_\rrbracket_{\mathrm{P}} \;:\; \sigma \leqslant_{\mathrm{TreeP}} \tau \;\rightarrow\; \sigma \leqslant_{\mathrm{Tree}} \tau \\
\llbracket\ \sigma{\leqslant}\tau\ \rrbracket_{\mathrm{P}} \;=\; \llbracket\ \mathit{whnf}\ \sigma{\leqslant}\tau\ \rrbracket_{\mathrm{W}}
\end{array}$$

Note that these functions are guarded and hence productive. Finally we get the soundness proof:

$$\begin{array}{l}
\mathit{sound} \;:\; \sigma \leqslant \tau \;\rightarrow\; \sigma \leqslant_{\mathrm{Type}} \tau \\
\mathit{sound}\ \sigma{\leqslant}\tau \;=\; \llbracket\ \mathit{sound}_{\mathrm{P}}\ \sigma{\leqslant}\tau\ \rrbracket_{\mathrm{P}}
\end{array}$$

6 Inductive Axiomatisation of Subtyping

Brandt and Henglein (1998) do not define subtyping using mixed induction and coinduction, as in Section 5, but using an inductive *encoding* of coinduction. Their subtyping relation is ternary: $A \vdash \sigma \leqslant \tau$ means that σ is a subtype of τ given the assumptions in A. An assumption (a hypothesis) is simply a pair of types:

$$\begin{array}{l}
\textbf{data}\ \mathit{Hyp}\ (n \;:\; \mathbb{N}) \;:\; \mathit{Set}\ \textbf{where} \\
\quad \_{\lesssim}\_ \;:\; \mathit{Ty}\ n \;\rightarrow\; \mathit{Ty}\ n \;\rightarrow\; \mathit{Hyp}\ n
\end{array}$$

The subtyping relation is defined as follows:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \_{\vdash}\_{\leqslant}\_\ (A \;:\; \mathit{List}\ (\mathit{Hyp}\ n)) \;:\; \mathit{Ty}\ n \;\rightarrow\; \mathit{Ty}\ n \;\rightarrow\; \mathit{Set}\ \textbf{where}} \\
\quad \bot & : & A \vdash \bot \leqslant \tau \\
\quad \top & : & A \vdash \sigma \leqslant \top \\
\quad \_{\rightarrowtriangle}\_ & : & \textbf{let}\ H \;=\; \sigma_1 \rightarrowtriangle \sigma_2 \lesssim \tau_1 \rightarrowtriangle \tau_2\ \textbf{in} \\
 & & H :: A \vdash \tau_1 \leqslant \sigma_1 \;\rightarrow\; H :: A \vdash \sigma_2 \leqslant \tau_2 \;\rightarrow \\
 & & A \vdash \sigma_1 \rightarrowtriangle \sigma_2 \leqslant \tau_1 \rightarrowtriangle \tau_2 \\
\quad \mathsf{unfold} & : & A \vdash \mu\,\tau_1 \rightarrowtriangle \tau_2 \leqslant \mathit{unfold}\langle\mu\,\tau_1 \rightarrowtriangle \tau_2\ \rangle \\
\quad \mathsf{fold} & : & A \vdash \mathit{unfold}\langle\mu\,\tau_1 \rightarrowtriangle \tau_2\ \rangle \leqslant \mu\,\tau_1 \rightarrowtriangle \tau_2
\end{array}$$

```
refl  : $A \vdash \tau \leqslant \tau$
trans : $A \vdash \tau_1 \leqslant \tau_2 \to A \vdash \tau_2 \leqslant \tau_3 \to A \vdash \tau_1 \leqslant \tau_3$
hyp   : $\sigma \lesssim \tau \in A \to A \vdash \sigma \leqslant \tau$
```

Here $\_{\in}\_$ encodes list membership. Note that coinduction is encoded in the $\_{\multimap}\_$ rule by inclusion of the consequent in the lists of assumptions of the antecedents.

Brandt and Henglein prove that their relation (with an empty list of assumptions) is equivalent to Amadio and Cardelli's. Their proof is considerably more complicated than the proof outlined above which shows that $\_{\leqslant}\_$ is equivalent to Amadio and Cardelli's definition, but as part of the proof they show that subtyping is decidable. By composing the two equivalence proofs we get that subtyping as defined in Section 5 is also decidable.

Brandt and Henglein use a classical argument to show that their algorithm terminates, so it is not entirely obvious that it can be implemented in a total, constructive type theory like Agda. However, we have adapted the algorithm to this setting:

$$\_{\leqslant?}\_ : (\sigma\ \tau : \mathit{Ty}\ n) \to \mathit{Dec}\ ([\,] \vdash \sigma \leqslant \tau)$$

A value in *Dec A* is either a value in A, or a proof showing that no such value exists, so this decision procedure does not merely say "yes" or "no", it backs up its verdict with solid evidence. Details of the implementation of $\_{\leqslant?}\_$ are available in the code accompanying the paper (Danielsson 2010a).

We know that $\_{\vdash}\_{\leqslant}\_$ is equivalent to $\_{\leqslant}\_$, because both relations are equivalent to Amadio and Cardelli's. However, it can still be instructive to see a direct proof of soundness of $\_{\vdash}\_{\leqslant}\_$ with respect to $\_{\leqslant}\_$. The proof below uses a cyclic (but productive) proof to turn the inductive encoding of coinduction used in $\_{\vdash}\_{\leqslant}\_$ into the "actual" coinduction used in $\_{\leqslant}\_$.

To state soundness the type *All* is used; *All P xs* means that all elements in *xs* satisfy *P*:

```
data All (P : A → Set) : List A → Set where
  []   : All P []
  _::_ : P x → All P xs → All P (x :: xs)
```

The soundness proof shows that if $A \vdash \sigma \leqslant \tau$, where all pairs $\sigma' \lesssim \tau'$ in A satisfy $\sigma' \leqslant \tau'$, then $\sigma \leqslant \tau$:

```
Valid : (Ty n → Ty n → Set) → Hyp n → Set
Valid _R_ ($\sigma_1 \lesssim \sigma_2$) = $\sigma_1$ R $\sigma_2$
sound : All (Valid _≤_) A → A ⊢ σ ≤ τ → σ ≤ τ
```

The interesting cases of *sound* are the ones for trans, hyp and $\_{\multimap}\_$. Transitivity can be handled recursively, hypotheses can be looked up in the list of valid assumptions (using $\mathit{lookup} : \mathit{All}\ P\ \mathit{xs} \to x \in \mathit{xs} \to P\ x$), and function spaces can be handled by defining a cyclic proof:

```
sound valid (trans τ₁≤τ₂ τ₂≤τ₃) = trans (sound valid τ₁≤τ₂)
                                        (sound valid τ₂≤τ₃)
sound valid (hyp h)             = lookup valid h
sound valid (τ₁≤σ₁ ⇾ σ₂≤τ₂)    = proof
  where proof = ♯ sound (proof :: valid) τ₁≤σ₁ ⇾
                ♯ sound (proof :: valid) σ₂≤τ₂
```

Note that the last two calls to *sound* extend the list of valid assumptions with the proof currently being defined.

The definition of *proof* above is not guarded, but it would be if *sound* were a constructor. We use the technique from Section 5 to make the proof guarded. The program and WHNF types can be defined mutually as follows:

```
data _≤P_ : Ty n → Ty n → Set where
  sound : All (Valid _≤W_) A → A ⊢ σ ≤ τ → σ ≤P τ
data _≤W_ : Ty n → Ty n → Set where
  done  : σ ≤ τ → σ ≤W τ
  _⇾_   : ∞ (τ₁ ≤P σ₁) → ∞ (σ₂ ≤P τ₂) → σ₁ ⇾ σ₂ ≤W τ₁ ⇾ τ₂
  trans : τ₁ ≤W τ₂ → τ₂ ≤W τ₃ → τ₁ ≤W τ₃
```

The cases of *sound* listed above are now part of a function *sound*$_W$ which is used by *whnf* to interpret sound:

```
soundW : All (Valid _≤W_) A → A ⊢ σ ≤ τ → σ ≤W τ
...
soundW valid (trans τ₁≤τ₂ τ₂≤τ₃) = trans (soundW valid τ₁≤τ₂)
                                         (soundW valid τ₂≤τ₃)
soundW valid (hyp h)             = lookup valid h
soundW valid (τ₁≤σ₁ ⇾ σ₂≤τ₂)    = proof
  where proof = ♯ sound (proof :: valid) τ₁≤σ₁ ⇾
                ♯ sound (proof :: valid) σ₂≤τ₂
whnf : σ ≤P τ → σ ≤W τ
whnf (sound valid σ≤τ) = soundW valid σ≤τ
```

Note that *proof* is now guarded. For the definitions of $[\![\_]\!]_W$, $[\![\_]\!]_P$ and *sound*, see the accompanying code (Danielsson 2010a).

We have not found a proof of completeness of $\_\vdash\_\leqslant\_$ with respect to $\_\leqslant\_$ which does not use a decision procedure for subtyping. This is not entirely surprising: such a completeness proof must turn a potentially infinite proof of $\sigma \leqslant \tau$ into a finite proof of $[\,] \vdash \sigma \leqslant \tau$, so some "trick" is necessary. With a suitably formulated decision procedure at hand the trick is simple. We have implemented a decision procedure *dec* which gives either a proof of $[\,] \vdash \sigma \leqslant \tau$, or a proof which shows that $\sigma \leqslant \tau$ is impossible. In the first case we are done, and in the second case a contradiction can be derived. (The decision procedure *dec*, together with the proof of soundness of $\_\vdash\_\leqslant\_$, is used to implement the decision procedure $\_\leqslant?\_$ mentioned above.)

7 Postulating an Admissible Rule May Not Be Sound

Given an inductively defined inference system one can add a new rule corresponding to an admissible property without changing the set of derivable properties. It is easy to prove this statement by defining functions which translate between the two inference systems. Translating derivations from the old to the new inference system is trivial. When translating in the other direction one can replace all occurrences of the new rule with instances of the proof of admissibility; this process can be implemented using recursion over the structure of the input derivation.

However, when coinduction comes into the picture this property no longer holds (de Vries 2009). The proof given above breaks down because there is no guarantee that the second translation can be implemented in a productive way. The problem is that, although the admissible rule has a proof, this proof may not be sufficiently "contractive" (for instance, the proof may replace coinductive rules in the input derivation with inductive rules in the output derivation).

The following example illustrates the problem. Recall the definition of the partiality monad in Section 2.6. One can prove that the equality _≅_ is an equivalence relation, and that it is not trivial (assuming that the result type A is inhabited). Let us now add transitivity as an inductive rule:

```
data _≅_ : A ν → A ν → Set where
  ...
  trans : x ≅ y → y ≅ z → x ≅ z
```

Given this new constructor we can prove, using guarded coinduction, that the relation is trivial:

```
trivial : (x y : A ν) → x ≅ y
trivial x y = trans (stepʳ (refl x))
                    (trans (step (♯ trivial x y))
                           (stepˡ (refl y)))
```

The proof uses the following steps: $x \;\cong\;$ step (♯ x) $\;\cong\;$ step (♯ y) $\;\cong\; y$. (The function *refl* is a proof of reflexivity.)

This problem does not affect the definition of subtyping given above, which has been proved to be equivalent to other definitions from the literature. However, it means that one should exercise caution when defining relations using mixed induction and coinduction, and avoid relying on results or intuitions which are only valid in the inductive case. Note that the problem with _≅_ is closely related to the problem of weak bisimulation up to weak bisimilarity (Sangiorgi and Milner 1992); presumably some of the techniques which have been developed to address the latter problem are also applicable to the former.

There are actually several different ways in which one can close a coinductively defined binary relation

```
data _∼_ : A → A → Set where
  ...
```

under transitivity. We list three:

1. One can include transitivity as a coinductive constructor:

   ```
   data _∼_ : A → A → Set where
     ...
     trans : ∞ (x ∼ y) → ∞ (y ∼ z) → x ∼ z
   ```

   This amounts to defining the largest relation which is closed under transitivity, and is not very useful, as pointed out in the introduction.
2. One can define the least relation which includes _∼_ and is closed under transitivity:

   ```
   data _∼′_ : A → A → Set where
     include : x ∼ y → x ∼′ y
     trans   : x ∼′ y → y ∼′ z → x ∼′ z
   ```

   This "solves" the problem outlined above, because if _∼_ is transitive, then _∼_ and _∼′_ are equivalent. However, in any given proof trans can only be used a finite number of times, and this can be a rather severe restriction. For instance, the definition of _⩽_ in Section 5 would not have been correct if trans had been defined using this method.
3. Finally one can include transitivity as an inductive constructor, like in the definition of _⩽_:

   ```
   data _∼_ : A → A → Set where
     ...
     trans : x ∼ y → y ∼ z → x ∼ z
   ```

   This definition often gives a more useful notion of transitivity than the one above, because transitivity can be used anywhere in a proof, infinitely often, as long as there is never a stretch of infinitely many transitivity constructors without any intervening coinductive constructor. However, this notion of transitivity can sometimes be too strong, as illustrated for the partiality monad equality _≅_ above: the "infinitely transitive closure" is sometimes the trivial relation.

8 Conclusions

We have showed that coinduction can be usefully combined with the rule of transitivity, and discussed under what conditions the technique is applicable. We have also defined subtyping for recursive types in a new way, and compared this definition to a similar axiomatisation given by Brandt and Henglein (1998). Brandt and Henglein note that their inductive encoding of coinduction seems to be closely related to guarded coinduction, but leave a precise comparison to future work. This paper provides a precise comparison, albeit not for the general case, but only for a particular example (the subtyping relations given in Sections 5 and 6).

It is our hope that this paper provides a compelling example of the use of mixed induction and coinduction. We have found this technique useful in a number of situations (Danielsson and Altenkirch 2009), and encourage more programming language researchers—as well as programmers interested in guaranteed totality—to become familiar with it.

Acknowledgements. Nils Anders Danielsson would like to thank Peter Hancock for introducing him to the technique of mixed induction and coinduction. Thorsten Altenkirch would like to thank Nicolas Oury for joint work on *ΠΣ* which has had an impact on the work in this paper, and Graham Hutton for earlier joint but unpublished work on coinductive reasoning. Both authors would like to thank Conor McBride, Nicolas Oury and Anton Setzer for many discussions about coinduction which have influenced this work, and Graham Hutton, Henrik Nilsson and some anonymous reviewers for feedback which improved the presentation. Both authors gratefully acknowledge funding from EPSRC (grant codes: EP/E04350X/1 and EP/G034109/1).

References

Abel, A.: Mixed inductive/coinductive types and strong normalization. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 286–301. Springer, Heidelberg (2007)

Abel, A., Altenkirch, T.: A predicative analysis of structural recursion. Journal of Functional Programming 12(1), 1–41 (2002)

The Agda Team. The Agda Wiki (2010), `http://wiki.portal.chalmers.se/agda/`

Amadio, R.M., Cardelli, L.: Subtyping recursive types. ACM Transactions on Programming Languages and Systems 15(4), 575–631 (1993)

Barthe, G., Frade, M.J., Giménez, E., Pinto, L., Uustalu, T.: Type-based termination of recursive definitions. Mathematical Structures in Computer Science 14(1), 97–141 (2004)

Barwise, J.: Mixed Fixed Points. In: The Situation in Logic. CSLI Lecture Notes, vol. 17, Center for the Study of Language and Information, Leland Stanford Junior University (1989)

Bradfield, J., Stirling, C.: Modal mu-calculi. In: Handbook of Modal Logic. Studies in Logic and Practical Reasoning, vol. 3. Elsevier, Amsterdam (2007)

Brandt, M., Henglein, F.: Coinductive axiomatization of recursive type equality and subtyping. Fundamenta Informaticae 33(4), 309–338 (1998)

Capretta, V.: General recursion via coinductive types. Logical Methods in Computer Science 1(2), 1–28 (2005)

Coquand, T.: Infinite objects in type theory. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 62–78. Springer, Heidelberg (1994)

Cousot, P., Cousot, R.: Inductive definitions, semantics and abstract interpretations. In: POPL '92, Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 83–94 (1992)

Danielsson, N.A.: Code accompanying the paper (2010a), `http://www.cs.nott.ac.uk/~nad/`

Danielsson, N.A.: Beating the productivity checker using embedded languages. Draft (2010b)

Danielsson, N.A., Altenkirch, T.: Mixing induction and coinduction. Draft (2009)

de Vries, E.: Re: [Coq-Club] Adding (inductive) transitivity to weak bisimilarity not sound? (was: Need help with coinductive proof). Message to the Coq-Club mailing list (August 2009)

Gapeyev, V., Levin, M.Y., Pierce, B.C.: Recursive subtyping revealed. Journal of Functional Programming 12(6), 511–548 (2002)

Gibbons, J., Hutton, G.: Proof methods for corecursive programs. Fundamenta Informaticae 66(4), 353–366 (2005)

Giménez, E.: Un Calcul de Constructions Infinies et son Application à la Vérification de Systèmes Communicants. PhD thesis, Ecole Normale Supérieure de Lyon (1996)

Gordon, A.D.: Bisimilarity as a theory of functional programming. Theoretical Computer Science 228(1-2), 5–47 (1999)

Hagino, T.: A Categorical Programming Language. PhD thesis, University of Edinburgh (1987)

Hancock, P., Pattinson, D., Ghani, N.: Representations of stream processors using nested fixed points. Logical Methods in Computer Science 5(3:9) (2009)

Hensel, U., Jacobs, B.: Proof principles for datatypes with iterated recursion. In: Moggi, E., Rosolini, G. (eds.) CTCS 1997. LNCS, vol. 1290, pp. 220–241. Springer, Heidelberg (1997)

Hughes, J., Moran, A.: Making choices lazily. In: FPCA '95, Proceedings of the seventh international conference on Functional programming languages and computer architecture, pp. 108–119 (1995)

Kozen, D., Palsberg, J., Schwartzbach, M.I.: Efficient recursive subtyping. Mathematical Structures in Computer Science 5(1), 113–125 (1995)

Leroy, X., Grall, H.: Coinductive big-step operational semantics. Information and Computation 207(2), 284–304 (2009)

Levy, P.B.: Infinitary Howe's method. In: Proceedings of the Eighth Workshop on Coalgebraic Methods in Computer Science (CMCS 2006). ENTCS, vol. 164, pp. 85–104 (2006)

Mendler, P.F.: Inductive Definition in Type Theory. PhD thesis, Cornell University (1988)

Milner, R.: Operational and algebraic semantics of concurrent processes. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics. The MIT Press and Elsevier (1990)

Milner, R., Tofte, M.: Co-induction in relational semantics. Theoretical Computer Science 87(1), 209–220 (1991)

Müller, O., Nipkow, T., von Oheimb, D., Slotosch, O.: HOLCF = HOL + LCF. Journal of Functional Programming 9(2), 191–223 (1999)

Nakata, K., Uustalu, T.: Trace-based coinductive operational semantics for While; Big-step and small-step, relational and functional styles. In: Urban, C. (ed.) TPHOLs 2009. LNCS, vol. 5674, pp. 375–390. Springer, Heidelberg (2009)

Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology and Göteborg University (2007)

Park, D.: On the semantics of fair parallelism. In: Bjorner, D. (ed.) Abstract Software Specifications. LNCS, vol. 86, pp. 504–526. Springer, Heidelberg (1980)

Raffalli, C.: L'Arithmétique Fonctionnelle du Second Ordre avec Points Fixes. PhD thesis, Université Paris VII (1994)

Sangiorgi, D., Milner, R.: The problem of "weak bisimulation up to". In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 32–46. Springer, Heidelberg (1992)

Turner, D.A.: Total functional programming. Journal of Universal Computer Science 10(7), 751–768 (2004)

Wadler, P., Taha, W., MacQueen, D.: How to add laziness to a strict language, without even being odd. In: Proceedings of the 1998 ACM SIGPLAN Workshop on ML (1998)

Compositional Action System Derivation Using Enforced Properties

Brijesh Dongol and Ian J. Hayes

School of Information Technology and Electrical Engineering,
The University of Queensland
Brisbane, Australia
{brijesh,ianh}@itee.uq.edu.au

Abstract. Action systems have been shown to be applicable for modelling and constructing both sequential and concurrent systems. This paper presents an approach to program construction where the concrete implementation is derived from its specification — via a series of small refinements — using incomplete proofs to motivate changes to the program. Formalisation of our approach is provided by *enforced properties*, which restrict the traces of a program to those that satisfy the enforced properties. The goal of the derivation is to refine a program with enforced properties to a program (with no enforced properties) whose code satisfies the enforced properties. An advantage of this approach is that the code in the earlier versions of the program need not be complete; incorrect execution of the program is avoided by including enforced properties in the specification. Enforced properties may be any temporal formula or relation, and hence we may reason about both safety and progress in a compositional setting.

1 Introduction

The theory of action systems is well established and has been applied to a number of problem domains. Back and von Wright have presented rules for compositional refinement of two action systems in a parallel context [4]. That is, given that $\mathscr{A}$, $\mathscr{A}'$ and $\mathscr{B}$ are action systems, Back and von Wright's rules allow the refinement $\mathscr{A} \| \mathscr{B} \sqsubseteq \mathscr{A}' \| \mathscr{B}$ to be verified without examining the code of $\mathscr{B}$. However, program development using their method follows the usual refinement process where concrete code is constructed, followed by a proof that the concrete action system refines the abstract (original) specification.

In this paper, we describe an alternative approach to developing concrete implementations using the calculational verify-while-develop paradigm first advocated by Dijkstra [8] for sequential programs, which has been extended to concurrent programs [11,12,13]. In [11,12,13], a program consists of some incomplete code together with *queried properties* that express the desired properties of the program. Modifications to both the program statements and queried properties are motivated using weakest precondition calculations. Although this technique has been successfully applied to derive a number of concurrent programs that satisfy safety [13] and progress properties [11,12], the derivations themselves remain informal because program code and queried properties may be arbitrarily modified.

The verify-while-develop paradigm is related to refinement by redefining queried properties as *enforced properties* [9,10]. Enforced properties are specified using Linear Temporal Logic [19], and hence, both safety and progress properties may be specified.

As a concrete example, let us consider the action system in Fig. 1, which initialises x and y to 0, and contains actions that increment x and y. The program also includes enforced invariant $\Box(x \leq y)$, which is identified using '?'. This enforced invariant ensures that $x \leq y$ always holds in every state of the program. That is, although the code of the action system may arbitrarily increment x and y, traces that do not satisfy $\Box(x \leq y)$ are not permitted.

```
$x, y := 0, 0$ ;
**do** $\mathit{true} \longrightarrow x := x + 1$
$\sqcap$  $\mathit{true} \longrightarrow y := y + 1$
**od** $? \Box(x \leq y)$
```

Fig. 1. Example program

The derivation process consists of modifications to the program code where we aim to obtain a program whose code does not generate traces other than those that satisfy the enforced properties. For instance, a possible modification to the program in Fig. 1 is to strengthen the first guard to $x < y$, which would guarantee the enforced property $\Box(x \leq y)$ holds. In the course of the derivation, we may also introduce or modify the enforced properties themselves. Furthermore, we note that the final program may contain enforced properties that formalise system assumptions such as fairness or environmental assumptions.

It is often necessary to introduce fresh variables to a program. To this end, we extend actions systems with *frames* [21]. Execution of an action with a non-empty frame involves execution of the action followed by modification of the frame variables to any value within their type. However, in the presence of enforced properties, the frame variables may only be modified in a manner that satisfies the enforced properties.

Using a combination of frames and enforced properties, we model the environment of a component within the component itself by introducing a separate action that performs environment transitions. This treatment allows us to specify the environment variables and restrictions on how they may be modified in a straightforward, yet precise manner. Embedding the environment within the component also allows us to re-use much of the existing theory on standard action systems refinement [3] and enforced properties [9,10].

Enforced properties have been incorporated into a framework of concurrently executing sequential programs [9,10]. However, this framework uses a rigid program counter model to simplify progress proofs at the cost of a more complicated theory. Extending action systems with enforced properties allows the theory to be simplified and we show how an action system that satisfies safety and progress may be derived. We introduce and use a more general temporal logic over relations to allow compositional proofs and show how enforced properties using this temporal logic may be used to formalise system assumptions such as fairness and environmental assumptions.

In Section 2 we describe the syntax and semantics of action systems; in Section 3 we formalise action systems with enforced properties; and in Section 4 we present an example derivation.

2 Action Systems

We present the syntax, predicate transformer and trace semantics of action systems in Section 2.1; refinement in Section 2.2; the temporal logic we use in Section 2.3; and action systems with frames in Section 2.4.

2.1 Syntax and Semantics

We define a *state space* as $\Sigma_{VAR} \mathrel{\widehat{=}} VAR \to VAL$ where VAR is a set of variables and VAL a set of values. We leave out the subscript if VAR is clear from the context. A *state* is a member of Σ. An expression maps states to values $\Sigma \to VAL$. A *predicate* is a member of the set $\mathcal{P}\Sigma \mathrel{\widehat{=}} \Sigma \longrightarrow \mathbb{B}$ that maps each state to $true$ or $false$. For state spaces Σ and Γ, a *predicate transformer* of type $\mathcal{P}\Gamma \longrightarrow \mathcal{P}\Sigma$ is a function that maps predicates over Γ to predicates over Σ. We use '.' for function application.

Suppose S, S_1, and S_2 are statements; b is a predicate; x is a variable; E is an expression; V is a set-valued expression; A, A_1 and A_2 are actions; l is a label; and $\mathscr{A}$ is an action system. We define:

$$\begin{aligned} S &::= \textbf{diverge} \mid \textbf{skip} \mid x := E \mid x :\in V \mid S_1 \; ; \; S_2 \\ A &::= l\colon b \longrightarrow S \mid A_1 \sqcap A_2 \end{aligned}$$

Fig. 2. Syntax of action systems

Thus a statement, S, may either be "**diverge**", which does not terminate, "**skip**", which does nothing, "$x := E$" which assigns E to x, "$x :\in V$", which non-deterministically assigns a value from V to x, or "$S_1 \; ; \; S_2$", which is the sequential composition of S_1 and S_2. An action, A, is either a *guarded statement*, $b \longrightarrow S$, or the *demonic choice* between two actions, $A_1 \sqcap A_2$. We use notation $[b]$ for $b \longrightarrow \textbf{skip}$. Furthermore, guarded statements may be labelled to allow them to be referred to more succinctly. Thus, the label of a guarded statement may be used interchangeably with the guarded statement the label represents. Given that A_0 and A are actions, an *action system* $\mathscr{A}$ takes the following form:

$$\mathscr{A} \mathrel{\widehat{=}} A_0 \; ; \; \textbf{do } A \textbf{ od}$$

where *initialisation action* A_0 is followed by a (possibly infinite) loop that executes action A.

We identify an action with the predicate transformer that given a postcondition, p, returns the *weakest precondition* for the action to establish p (see Fig. 3). Given that '$\oplus$' is the *functional override* operator we define $p[E/x] \mathrel{\widehat{=}} \lambda\,\sigma{:}\,\Sigma \bullet p.(\sigma \oplus \{x \mapsto E.\sigma\})$.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mathbf{diverge}.p & \equiv & \mathit{false} & (S_1\ ;\ S_2).p & \equiv & S_1.(S_2.p) \\
\mathbf{skip}.p & \equiv & p & (b \longrightarrow S).p & \equiv & (b \Rightarrow S.p) \\
(x := E).p & \equiv & p[E/x] & (A_1 \sqcap A_2).p & \equiv & A_1.p \wedge A_2.p \\
(x :\in V).p & \equiv & \forall v{:}\, V \bullet p[v/x] & & &
\end{array}$$

Fig. 3. Statements and actions as predicate transformers

The *termination* and *guard* of an action A are given by predicates $t.A \mathrel{\widehat{=}} A.\mathit{true}$ and $\mathit{grd}.A \mathrel{\widehat{=}} \neg A.\mathit{false}$, respectively.

Relations. For an action A, we let $\mathit{rel}.A{:}\,\mathbb{P}(\Sigma^{\uparrow} \times \Gamma^{\uparrow})$ denote the relation that corresponds to A (see Fig. 4), where $\uparrow$ is a special state that denotes *divergence* and $\Gamma^{\uparrow} \mathrel{\widehat{=}} \Gamma \cup \{\uparrow\}$. The values of all variables in $\uparrow$ are undefined, and we assume that $\uparrow \notin \Sigma \cup \Gamma$. Furthermore, for any relation $r \in \mathbb{P}(\Sigma^{\uparrow} \times \Gamma^{\uparrow})$, we assume $r[\{\uparrow\}] = \{\uparrow\}$, where $r[S]$ denotes the *relational image* of r on a set S. Hence, there cannot be a transition from a divergent state to a non-divergent state. We say r is *divergent* iff there exists a $\sigma \in \Sigma$ such that $(\sigma, \uparrow) \in r$, and *non-divergent* otherwise.

Relations may be written as formulas consisting of primed and unprimed variables. The values of primed variables are interpreted in the next state, and unprimed variables are interpreted in the current state (cf [18]). For a set of variables X and relation r, we define:

$$X \mid r \mathrel{\widehat{=}} \{(\sigma, \sigma'){:}\, \Sigma \times \Gamma \mid (\sigma, \sigma') \in r \wedge (\forall y{:}\, \mathrm{dom}.\sigma - X \bullet \sigma.y = \sigma'.y)\} \cup \{(\uparrow, \uparrow)\}$$

Thus, variables not in X remain unchanged in $X \mid r$.

For relations $r \in \mathbb{P}(\Sigma^{\uparrow} \times \Gamma^{\uparrow})$, $r_1 \in \mathbb{P}(\Sigma^{\uparrow} \times \Delta^{\uparrow})$ and $r_2 \in \mathbb{P}(\Delta^{\uparrow} \times \Gamma^{\uparrow})$, and predicate p, we define *relational composition* and *domain restriction*, respectively as follows [27]:

$$\begin{array}{rl}
r_1 \mathbin{⨾} r_2 \mathrel{\widehat{=}} & \{(\sigma, \sigma'){:}\, \Sigma^{\uparrow} \times \Gamma^{\uparrow} \mid \exists \sigma''{:}\, \Delta^{\uparrow} \bullet (\sigma, \sigma'') \in r_1 \wedge (\sigma'', \sigma') \in r_2\} \\
p \lhd r \mathrel{\widehat{=}} & \{(\sigma, \sigma'){:}\, \Sigma^{\uparrow} \times \Gamma^{\uparrow} \mid (\sigma, \sigma') \in r \wedge (\sigma \neq \uparrow \Rightarrow p.\sigma)\}
\end{array}$$

While rel allows actions to be interpreted as relations, any relation r may in turn be interpreted as an action using the *demonic update statement*, denoted $\langle r \rangle$ [4]. For any predicate p, its predicate transformer semantics is defined by

$$\langle r \rangle.p \mathrel{\widehat{=}} (\lambda\,\sigma{:}\,\Sigma \bullet (\sigma, \uparrow) \notin r \wedge (\forall \sigma'{:}\,\Gamma \bullet (\sigma, \sigma') \in r \Rightarrow p.\sigma'))$$

For any action A and non-divergent relation r, we have $\langle \mathit{rel}.A \rangle = A$ and $\mathit{rel}.\langle r \rangle = r$.

Given that E is an expression, V is a set-valued expression, and x' does not occur free in E and V we have:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{rel}.\mathbf{diverge} & \mathrel{\hat=} & \lambda\,\sigma \bullet \uparrow & \mathit{rel}.(S_1\ ;\ S_2) & \mathrel{\hat=} & \mathit{rel}.S_1 \mathbin{⨾} \mathit{rel}.S_2 \\ \mathit{rel}.\mathbf{skip} & \mathrel{\hat=} & \lambda\,\sigma \bullet \sigma & \mathit{rel}.(b \longrightarrow S) & \mathrel{\hat=} & b \lhd \mathit{rel}.S \\ \mathit{rel}.(x := E) & \mathrel{\hat=} & x \mid x' = E & \mathit{rel}.(A_1 \sqcap A_2) & \mathrel{\hat=} & \mathit{rel}.A_1 \cup \mathit{rel}.A_2 \\ \mathit{rel}.(x :\in V) & \mathrel{\hat=} & x \mid x' \in V & & & \end{array}$$

Fig. 4. Statements and actions as relations

Traces. We define $K^+ \mathrel{\hat=} K - \{0\}$ for a set $K \subseteq \mathbb{N}$.

Definition 1. *We let* $\mathit{initial}.\mathscr{A} \mathrel{\hat=} \{\sigma' : \Sigma^{\uparrow} \mid \exists \sigma : \Sigma \bullet (\sigma, \sigma') \in \mathit{rel}.A_0\}$ *be the set of* initial states *of* $\mathscr{A}$*. A possibly infinite sequence of states* $s \in \mathrm{seq}\,.\Sigma^{\uparrow}$ *is a* trace *of* $\mathscr{A}$ *iff* $s_0 \in \mathit{initial}.\mathscr{A} \wedge \forall u : (\mathrm{dom}.s)^+ \bullet s_{u-1} \neq \uparrow \wedge (s_{u-1}, s_u) \in \mathit{rel}.A$.

Note that if a trace includes $\uparrow$, then $\uparrow$ must be the last element of the trace.

Suppose s is a trace of $\mathscr{A}$. If $\mathrm{dom}.s \neq \mathbb{N}$, we say $\mathscr{A}$ *terminates* in s iff $\mathit{last}.s \neq \uparrow \wedge (\neg \mathit{grd}.A).(\mathit{last}.s)$ and *diverges* in s iff $\mathit{last}.s = \uparrow$. Trace s is *complete* iff $\mathrm{dom}.s = \mathbb{N} \vee \neg(\exists \sigma : \Sigma^{\uparrow} \bullet (\mathit{last}.s, \sigma) \in \mathit{rel}.A)$. Thus, a complete trace represents either an infinite, terminating, or diverging execution of the corresponding action system. We use $\mathsf{Tr}.\mathscr{A}$ to denote the set of all *complete traces* of $\mathscr{A}$.

We assume that an action system executes within an environment that may modify *environment variables* as described by a *rely* relation [16,7]. Although the environment is external to the action system, modelling it as such causes problems with our trace-based framework because environment transitions may occur in between program transitions. In particular, environment transitions may have an effect on progress properties. If an action system, say $\mathscr{A}$, executes within an environment described by a rely condition, say r, we replace A by $A \sqcap \langle r \rangle$. Statement $\langle r \rangle$ may be executed at any point during the execution of $\mathscr{A}$, which represents an environment transition. Thus environment transitions of $\mathscr{A}$ are recorded within the traces of $\mathscr{A}$.

2.2 Refinement

In this section we present a theory for refining actions and action systems. A *concrete* action system $\mathscr{C}$ refines an *abstract* action system $\mathscr{A}$ iff every observable behaviour of $\mathscr{C}$ is a possible observable behaviour of $\mathscr{A}$. Given that $OV \subseteq VAR$ is the set of *observable variables*, we let $rL : \mathrm{seq}\,.\Sigma_{VAR} \to \mathrm{seq}\,.\Sigma_{OV}$ be the function that removes unobservable variables from the given sequence of states. After removal of unobservable variables, e.g., if $s = rL.t$ where $t \in \mathrm{seq}\,.\Sigma_{VAR}$, it is common for *stuttering* to exist within s, i.e., consecutive states s_u and s_{u+1} such that $s_u = s_{u+1}$. States s_u and s_{u+1} are stuttering exactly when transition $(t_u, t_{u+1}) \in \mathit{rel}.A$ does not modify any observable variables. A finite number of consecutive stutterings of s_u may be represented by a single s_u, however, if s_u consecutively stutters an infinite number of times, we will never observe a change in state, and hence, we treat infinite stuttering as being equivalent to execution of $\mathbf{diverge}$. We define function $rS: \text{seq}.\Sigma \to \text{seq}.\Sigma^{\uparrow}$ that removes finite stuttering from a sequence of states and replaces infinite stuttering with an execution of $\mathbf{diverge}$. (See [9] for formal definitions of rL and rS.)

Definition 2. *Action system* $\mathscr{C}$ trace refines *action system* $\mathscr{A}$ *iff* $\mathscr{A} \sqsubseteq_{\mathsf{Tr}} \mathscr{C}$ *holds, where* $\mathscr{A} \sqsubseteq_{\mathsf{Tr}} \mathscr{C} \mathrel{\widehat{=}} \forall t: \mathsf{Tr}.\mathscr{C} \bullet \exists s: \mathsf{Tr}.\mathscr{A} \bullet rS.(rL.s) = rS.(rL.t)$.

Definition 2 is the same as the definition of trace refinement provided by Back and von Wright [2].

Lemma 3. *If* $\mathsf{Tr}.\mathscr{C} \subseteq \mathsf{Tr}.\mathscr{A}$ *then* $\mathscr{A} \sqsubseteq_{\mathsf{Tr}} \mathscr{C}$.

An action A is refined by action C iff any behaviour of C is a possible behaviour of A. A refinement may, for example, reduce the non-determinism or strengthen the guard of an action.

Definition 4. *An action A is refined by an action C iff $A \sqsubseteq C$ holds, where $A \sqsubseteq C \mathrel{\widehat{=}} \forall p: \mathcal{P}\Sigma \bullet A.p \Rightarrow C.p$.*

If $A \sqsubseteq C$ and $C \sqsubseteq A$, we write $A \sqsubseteq\!\!\!\sqsupseteq C$. The following lemmas are trivial to prove.

Lemma 5. *Suppose* $\mathscr{A} \mathrel{\widehat{=}} A_0;\ \mathbf{do}\ A\ \mathbf{od}$ *and* $\mathscr{C} \mathrel{\widehat{=}} C_0;\ \mathbf{do}\ C\ \mathbf{od}$. *If* $A_0 \sqsubseteq C_0$, $A \sqsubseteq C$ *and* $grd.A \equiv grd.C$ *then* $\mathscr{A} \sqsubseteq_{\mathsf{Tr}} \mathscr{C}$.

Lemma 6. *If A, A_1 and A_2 are actions such that $A_1 \sqsubseteq A_2$ and $grd.(A \sqcap A_1) \Rightarrow grd.(A \sqcap A_2)$, then* $\mathbf{do}\ A \sqcap A_1\ \mathbf{od} \sqsubseteq \mathbf{do}\ A \sqcap A_2\ \mathbf{od}$.

Corollary 7. *If A, A_1, A_2 are actions such that $A_1 \sqsubseteq A_2$, then* $\mathbf{do}\ A \sqcap A_1\ \mathbf{od} \sqsubseteq \mathbf{do}\ A \sqcap A_1 \sqcap A_2\ \mathbf{od}$.

2.3 Temporal Logic

To specify properties of traces, we use an extension of Linear Temporal Logic (LTL) [19], which we call Relational Linear Temporal Logic (RLTL). An LTL formula [19] takes the following form, where $\odot \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$, and p is a predicate.

$$F ::= p \mid \Box F \mid \Diamond F \mid F_1\, \mathcal{U}\, F_2 \mid F_1\, \mathcal{W}\, F_2 \mid F_1 \odot F_2 \mid \forall x \bullet F \mid \exists x \bullet F$$

We extend these and define RLTL formulae, which may also contain relations.

$$\begin{aligned} Q &::= r \mid \Box Q \mid \Diamond Q \mid Q_1\, \mathcal{U}\, Q_2 \mid Q_1\, \mathcal{W}\, Q_2 \mid Q_1 \odot Q_2 \mid \forall x \bullet Q \mid \exists x \bullet Q \\ R &::= F \mid Q \mid R_1 \odot R_2 \mid \forall x \bullet R \mid \exists x \bullet R \end{aligned}$$

The semantics for RLTL formulae is provided in Fig. 5 where we assume s is a trace and Tr is a set of traces. The notation $(s, u) \vdash R$ states that RLTL formula R holds for trace s starting from index $u \in \text{dom}.s$. By definition $(s, u) \vdash p$ holds iff p holds in state s_u. Similarly, $(s, u) \vdash r$ holds iff $u + 1 \in \text{dom}.s$ and $(s_u, s_{u+1}) \in r$ holds. Operators '$\Box$', '$\Diamond$', '$\mathcal{U}$' and '$\mathcal{W}$' express 'always', 'eventually', 'until' and 'unless', respectively. The semantics for $\Diamond F$, $F_1 \mathcal{U} F_2$ and $F_1\, \mathcal{W}\, F_2$ are identical to their corresponding definitions on relations.

Notation	Meaning
$(s, u) \vdash p$	$p.s_u$
$(s, u) \vdash r$	$u + 1 \in \text{dom}.s \wedge (s_u, s_{u+1}) \in r$
$(s, u) \vdash \Box F$	$\forall v : \text{dom}.s \bullet v \geq u \Rightarrow (s, v) \vdash F$
$(s, u) \vdash \Box Q$	$\forall v : \text{dom}.s \bullet v \geq u \wedge v + 1 \in \text{dom}.s \Rightarrow (s, v) \vdash Q$
$(s, u) \vdash \Diamond Q$	$\exists v : \text{dom}.s \bullet v \geq u \wedge (s, v) \vdash Q$
$(s, u) \vdash Q_1 \,\mathcal{U}\, Q_2$	$\exists v : \text{dom}.s \bullet v \geq u \wedge (s, v) \vdash Q_2 \wedge (\forall w : u..v - 1 \bullet (s, w) \vdash Q_1)$
$(s, u) \vdash Q_1 \,\mathcal{W}\, Q_2$	$(s, u) \vdash (\Box Q_1 \vee (Q_1 \,\mathcal{U}\, Q_2))$
$(s, u) \vdash R_1 \odot R_2$	$((s, u) \vdash R_1) \odot ((s, u) \vdash R_2)$
$s \vdash R$	$(s, 0) \vdash R$
$Tr \models R$	$(\forall s : Tr \bullet s \vdash R)$

Fig. 5. RLTL semantics

RLTL allows reasoning with two-state relations, which facilitates properties such as "if p holds now, q must hold in the next state" to be expressed. Note that LTL already includes a 'next' operator ($\bigcirc$) which may be used to express two-state relations, however, incorporating $\bigcirc$ within a compositional framework makes the underlying logic significantly more complex, which deters us from making this choice [17]. Furthermore, trace refinement only preserves temporal properties given that the property does not mention the $\bigcirc$ operator [14].

Lemma 8

a. $\mathsf{Tr}.\mathscr{A} \models \Box p$ *holds provided* $A_0.p \wedge (p \Rightarrow A.p)$
b. $\mathsf{Tr}.\mathscr{A} \models \Box r$ *holds provided* $rel.A \subseteq r$

RLTL formulae allow various progress properties to be specified. There are a number of theorems for manipulating LTL formulae [9,19] which also hold for RLTL formulae. In this paper, we focus on the leads-to operator, denoted $\rightsquigarrow$, where $(p \rightsquigarrow q) = \Box(p \Rightarrow \Diamond q)$. Thus if $p \rightsquigarrow q$, then q will eventually hold if p ever holds. Leads-to properties are proved in a calculational manner using unless, hence we present the following lemma.

Lemma 9 (Unless)

a. $\mathsf{Tr}.\mathscr{A} \models p \,\mathcal{W}\, q$ *holds provided* $\mathsf{Tr}.\mathscr{A} \models (p \vee q) \wedge \Box(p \wedge \neg q \Rightarrow p' \vee q')$
b. $\mathsf{Tr}.\mathscr{A} \models \Box(p \Rightarrow (p \,\mathcal{W}\, q))$ *holds provided* $\mathsf{Tr}.\mathscr{A} \models \Box(p \wedge \neg q \Rightarrow p' \vee q')$

By Lemma 9 (a), $p \,\mathcal{W}\, q$ holds in $\mathscr{A}$ if the initialisation establishes $p \vee q$, and every action in $\mathscr{A}$ establishes $p \vee q$ from a state that satisfies $p \wedge \neg q$. Similarly, Lemma 9 (b).

When proving leads-to properties, we must take the underlying fairness assumption into account. For this paper, we consider *weak fairness*, which is formalised for an action A by the following RLTL formula [9,18]:

$$WFair.A \mathrel{\widehat{=}} \Box\Diamond\neg grd.A \vee \Box\Diamond rel.A$$

Note that $WFair.A$ is equivalent to $\Diamond\Box grd.A \Rightarrow \Box\Diamond rel.A$. Thus, if the guard of action A is continuously enabled, then A is eventually executed. For an action system $\mathscr{A}$, we let $acts.\mathscr{A}$ be the set of top-level actions of A, i.e., $A = \sqcap_{\alpha : acts.\mathscr{A}} \alpha$ and each

α: $acts.\mathscr{A}$ is of the form $b \longrightarrow S$. For example, if $A = b_1 \longrightarrow S_1 \sqcap b_2 \longrightarrow S_2$, then $acts.\mathscr{A} = \{b_1 \longrightarrow S_1, b_2 \longrightarrow S_2\}$. We define

$$WFair.\mathscr{A} \mathrel{\widehat{=}} \forall \alpha\colon acts.\mathscr{A} \bullet WFair.\alpha$$

which states an action system $\mathscr{A}$ is weakly fair. The following lemma allows us to prove leads-to properties in a calculational manner. It is inspired by the calculational proof of leads-to in UNITY [6].

Lemma 10 (Dongol [9])

a. $\mathsf{Tr}.\mathscr{A} \models p \rightsquigarrow q$ *holds provided* $\mathsf{Tr}.\mathscr{A} \models WFair.\mathscr{A} \wedge \Box(p \Rightarrow (p \,\mathcal{W}\, q))$ *and* $\exists \alpha\colon acts.\mathscr{A} \bullet p \wedge \neg q \Rightarrow grd.\alpha \wedge \alpha.q$.
b. $\mathsf{Tr}.\mathscr{A} \models p_1 \rightsquigarrow p_2$ *if there exists a* q *such that* $\mathsf{Tr}.\mathscr{A} \models (p_1 \rightsquigarrow q) \wedge (q \rightsquigarrow p_2)$.
c. $(p \rightsquigarrow q)$ *iff* $(p \wedge \neg q \rightsquigarrow q)$

By Lemma 10 (a), $p \rightsquigarrow q$ holds in $\mathscr{A}$ if $\mathscr{A}$ is weakly fair, $p\mathcal{W}q$ holds, and there exists an action α in $\mathscr{A}$ such that if $p \wedge \neg q$ holds, α is enabled and execution of α establishes q. By Lemma 10 (b), we may establish $p_1 \rightsquigarrow p_2$ transitively via an intermediate predicate q. Lemma 10 (c) states that q is established from a state that satisfies p if and only if q is established from a state that satisfies $p \wedge \neg q$.

2.4 Frames

When developing an implementation, it is often necessary to allow additional internal behaviour without introducing new observable traces. A formal and elegant way of achieving this within the refinement calculus is by using *frames* [22]. An action A with its frame extended by x, denoted $x \cdot [\![A]\!]$, behaves as A but in addition may also modify x to any value within the type of x.

Definition 11. *If* p *is a predicate,* A *is an action, and* x *is a variable of non-empty type* T*, then* $(x \cdot [\![A]\!]).p \mathrel{\widehat{=}} A.(\forall x\colon T \bullet p)$.

Each of the following is a consequence of this definition:

$$\begin{aligned} x \cdot [\![S]\!] &\sqsubseteq S \,;\; x :\in T \\ x \cdot [\![b \longrightarrow S]\!] &\sqsubseteq b \longrightarrow x \cdot [\![S]\!] \\ x \cdot [\![A_1 \sqcap A_2]\!] &\sqsubseteq x \cdot [\![A_1]\!] \sqcap x \cdot [\![A_2]\!] \end{aligned}$$

To facilitate introduction of fresh unobservable variables to an action system, we introduce actions $\mathbf{add}\, x$ and $\mathbf{rem}\, x$, that add and remove x from the current state space. Assuming $x \notin VAR$, we use the following predicate transformers which returns a predicate on the pre-state for a given predicate on the post-state:

$$\begin{aligned} &\mathbf{add}\, x\colon \mathcal{P}\Sigma_{VAR \cup \{x\}} \to \mathcal{P}\Sigma_{VAR} \\ &\mathbf{rem}\, x\colon \mathcal{P}\Sigma_{VAR} \to \mathcal{P}\Sigma_{VAR \cup \{x\}} \end{aligned}$$

These are defined as follows

$$\begin{aligned} (\mathbf{add}\, x).p &= (\forall x\colon T \bullet p) && \text{provided } x \text{ is of non-empty type } T \\ (\mathbf{rem}\, x).p &= p && \text{provided } x \text{ does not occur free in } p. \end{aligned}$$

Note that although the definitions of $\mathbf{add}\, x$ and $x \cdot [\![\mathbf{skip}]\!]$ are similar, their predicate transformers are of different types.

Lemma 12. *Suppose p is a predicate; A is an action; x is a variable that does not occur free in p and A; and V is a non-empty set of expressions. Then each of the following holds.*

a. $\mathbf{add}\, x\ ;\ \mathbf{rem}\, x \sqsupseteq \mathbf{skip}$
b. $x \cdot [\![A]\!]\ ;\ \mathbf{rem}\, x \sqsupseteq \mathbf{rem}\, x\ ;\ A$
c. $[p]\ ;\ \mathbf{rem}\, x \sqsupseteq \mathbf{rem}\, x\ ;\ [p]$
d. $x :\in V\ ;\ \mathbf{rem}\, x \sqsupseteq \mathbf{rem}\, x$

Definition 13. *If $\mathscr{A} \mathrel{\widehat{=}} A_0;\ \mathbf{do}\ A\ \mathbf{od}$ is an action system, and x is a variable, then*

$$[\![\mathrm{var}\, x \bullet \mathscr{A}]\!] \mathrel{\widehat{=}} \mathbf{add}\, x\ ;\ x \cdot [\![A_0]\!]\ ;\ \mathbf{do}\ x \cdot [\![A]\!]\ \mathbf{od}\ ;\ \mathbf{rem}\, x$$

The following lemma facilitates introduction of a new variable to the frame of an action system.

Lemma 14. *If $x \notin VAR$ is an unobservable variable then $\mathscr{A} \sqsubseteq [\![\mathrm{var}\, x \bullet \mathscr{A}]\!]$.*

Introducing a fresh variable to the frame of an action system constitutes a single refinement step. In our approach, further refinements may be performed by restricting the possible values of the variables in the frame by introducing new enforced properties to the action system, which effectively reduces non-determinism. A variable may be removed from the frame altogether using the following lemma.

Lemma 15. *Suppose x is unobservable, $x \cdot [\![\alpha]\!] \in acts.\mathscr{A}$, and C is obtained from A by replacing $x \cdot [\![\alpha]\!]$ by α. Then $\mathscr{A} \sqsubseteq \mathscr{C}$.*

3 Enforced Properties

An enforced property is an RLTL formula that restricts the set of traces of an action system to those that satisfy the enforced property [10]. Traces of the action system that do not satisfy the enforced property are not permitted.

Definition 16. *Action system $\mathscr{A}$ with* enforced property *R, denoted $\mathscr{A}\ ?\ R$, is an action system such that $\mathsf{Tr}.(\mathscr{A}\ ?\ R) \mathrel{\widehat{=}} \{s : \mathsf{Tr}.\mathscr{A} \mid s \vdash R\}$.*

Thus, although $\mathsf{Tr}.\mathscr{A}$ may contain traces that do not satisfy R, by definition, R is guaranteed to hold for any trace of $\mathscr{A}\ ?\ R$. The goal then is to obtain an action system $\mathscr{B}$ with no enforced properties such that $\mathscr{A}\ ?\ R \sqsubseteq \mathscr{B}$, i.e., $\mathscr{B}$ is a refinement of $\mathscr{A}$ whose traces satisfy R.

The following lemma describes trace refinement of action systems with enforced properties.

Lemma 17. *For action systems $\mathscr{A}$ and $\mathscr{C}$, and RLTL formulae R and R' each of the following holds:*

a.	$\mathscr{A} \sqsubseteq_{\mathsf{Tr}} \mathscr{A} \,?\, R$	
b.	$\mathscr{A} \,?\, R \sqsubseteq_{\mathsf{Tr}} \mathscr{A} \,?\, R'$	*provided* $R' \Rrightarrow R$
c.	$\mathscr{A} \,?\, R \sqsubseteq_{\mathsf{Tr}} \mathscr{C} \,?\, R$	*provided* $\mathscr{A} \sqsubseteq_{\mathsf{Tr}} \mathscr{C}$
d.	$\mathscr{A} \,?\, R \sqsupseteq\sqsubseteq_{\mathsf{Tr}} \mathscr{A}$	*provided* $\mathsf{Tr}.\mathscr{A} \models R$
e.	$\mathscr{A} \,?(R \wedge R') \sqsupseteq\sqsubseteq_{\mathsf{Tr}} (\mathscr{A} \,?\, R) \,?\, R'$	

The proof of this lemma is straightforward due to Lemma 3.

By Lemma 17, introducing a new enforced property or strengthening existing enforced properties results in a refinement. If an action system without an enforced property refines another, then the refinement holds with the enforced property included. An enforced property, say R, may be removed from an action system if the action system without enforced property R satisfies R. Furthermore, enforcing a conjunction of enforced properties is equivalent to enforcing the properties one at a time.

Two important forms of enforced properties are *enforced invariants* (formulae of the form $\Box p$) and *enforced guarantees* (formulae of the form $\Box r$) [7,16]. We first consider actions with enforced properties and define:

$$A \,!\, p \mathrel{\widehat{=}} [p] \,;\; A \,;\; [p]$$
$$A \,!\, r \mathrel{\widehat{=}} \langle rel.A \cap r \rangle$$

Thus, $A \,!\, p$ blocks if p does not hold prior to executing A, or if the execution of A does not establish p. Statement $A \,!\, r$ blocks if A can only execute in a manner that does not maintain r. The following lemma allows us to modify enforced properties within a single action.

Lemma 18. *For actions A, A_1, and A_2; predicates p, p_1, and p_2; and relations r, r_1, and r_2, each of the following holds:*

a.	$A \,!\, (p_1 \wedge p_2) \sqsupseteq\sqsubseteq (A \,!\, p_1) \,!\, p_2$	
b.	$A \,!\, (r_1 \cap r_2) \sqsupseteq\sqsubseteq (A \,!\, r_1) \,!\, r_2$	
c.	$A \,!\, p \sqsupseteq\sqsubseteq [p];\; A$	*provided* $p \Rrightarrow A.p$
d.	$A \,!\, r \sqsupseteq\sqsubseteq A$	*provided* $rel.A \subseteq r$
e.	$(A_1 \sqcap A_2) \,!\, p \sqsupseteq\sqsubseteq (A_1 \,!\, p) \sqcap (A_2 \,!\, p)$	
f.	$(A_1 \sqcap A_2) \,!\, r \sqsupseteq\sqsubseteq (A_1 \,!\, r) \sqcap (A_2 \,!\, r)$	

The following lemma states that an enforced invariant and guarantee may be removed from $\mathscr{A} \,?(\Box p \wedge \Box r)$ if the initialisation is replaced by $A_0;\; [p]$ and the main action is replaced by $A \,!\, p \,!\, r$.

Lemma 19. *Suppose R and R' are RLTL formulae and $rel.A_0$ is non-divergent. Then*

$$\mathscr{A} \,?(\Box p \wedge \Box r) \sqsupseteq\sqsubseteq_{\mathsf{Tr}} A_0;\; [p];\; \mathbf{do} A \,!\, p \,!\, r \;\mathbf{od} \tag{1}$$

Proof (1). Let $\mathscr{C}$ be the action system in the right hand side of (1). Then $C_0 = A_0;\; [p]$ and $C = A \,!\, p \,!\, r$. We prove $\mathscr{A} \,?(\Box p \wedge \Box r) \sqsupseteq\sqsubseteq_{\mathsf{Tr}} \mathscr{C}$ using Lemma 3, i.e, show that $\mathsf{Tr}.\mathscr{C} = \mathsf{Tr}.(\mathscr{A} \,?(\Box p \wedge \Box r))$. Using the definitions of $\langle\rangle$ and rel, it is straightforward to show that

$$rel.(A \,!\, p \,!\, r) = rel.(A \,!\, p) \cap r \tag{2}$$

$$
\begin{array}{ll}
& s \in \mathsf{Tr}.\mathscr{C} \\
= & \quad \{\text{Definition 1}\} \\
& (\exists \sigma{:}\, \Sigma \bullet (\sigma, s_0) \in \mathit{rel}.C_0) \wedge (\forall u{:}\, (\mathrm{dom}.s)^+ \bullet (s_{u-1}, s_u) \in \mathit{rel}.C) \\
= & \quad \{\text{definitions of } C_0 \text{ and } C\} \\
& (\exists \sigma{:}\, \Sigma \bullet (\sigma, s_0) \in \mathit{rel}.(A_0\ ;\ [p])) \wedge (\forall u{:}\, (\mathrm{dom}.s)^+ \bullet (s_{u-1}, s_u) \in \mathit{rel}.(A \,!\, p \,!\, r)) \\
= & \quad \{\text{property (2) above}\} \\
& (\exists \sigma{:}\, \Sigma \bullet (\sigma, s_0) \in \mathit{rel}.(A_0\ ;\ [p])) \wedge \\
& \qquad\qquad (\forall u{:}\, (\mathrm{dom}.s)^+ \bullet (s_{u-1}, s_u) \in \mathit{rel}.(A \,!\, p) \wedge (s_{u-1}, s_u) \in r) \\
= & \quad \{\text{definition of } \mathit{rel}\} \\
& (\exists \sigma{:}\, \Sigma \bullet (\sigma, s_0) \in \mathit{rel}.A_0) \wedge p.s_0 \wedge (\forall u{:}\, (\mathrm{dom}.s)^+ \bullet p.s_{u-1} \wedge \\
& \qquad\qquad (s_{u-1}, s_u) \in \mathit{rel}.A \wedge p.s_u \wedge (s_{u-1}, s_u) \in r) \\
= & \quad \{\text{logic}\} \\
& (\exists \sigma{:}\, \Sigma \bullet (\sigma, s_0) \in \mathit{rel}.A_0) \wedge (\forall u{:}\, (\mathrm{dom}.s)^+ \bullet (s_{u-1}, s_u) \in \mathit{rel}.A) \wedge \\
& \qquad\qquad (\forall u{:}\, \mathrm{dom}.s \bullet p.s_u) \wedge (\forall u{:}\, (\mathrm{dom}.s)^+ \bullet (s_{u-1}, s_u) \in r) \\
= & \quad \{\text{Definition 1}\}\{\text{definitions of } \Box p \text{ and } \Box r\} \\
& s \in \mathsf{Tr}.\mathscr{A} \wedge (s \vdash \Box p \wedge \Box r) \\
= & \quad \{\text{Definition 16}\} \\
& s \in \mathsf{Tr}.(\mathscr{A}\, ?(\Box p \wedge \Box r)) \qquad \Box
\end{array}
$$

Note that the enforced invariants must appear as a guard after each action, i.e., moving the blocking to the start of the action using predicate transformer rules is not always valid. To see this, consider the case where b is a variable of type $\mathbb{B}$ (boolean) and $\alpha \mathrel{\widehat{=}} \mathit{true} \longrightarrow (b :\in \mathbb{B})$. We have $\alpha\ ;\ [b] \sqsubseteq b := \mathit{true}$. Thus, $\alpha\ ;\ [b]$ has a trace that extends past α because b can be assigned true. If we attempt to move the blocking to the start, we obtain action $\gamma \mathrel{\widehat{=}} ((b :\in \mathbb{B}).b) \longrightarrow b :\in \mathbb{B}$, which is equivalent to $[\mathit{false}]$, whose trace is blocked at γ.

4 Industrial Press

We develop a controller for an industrial press [20,15]. We present the specification of the press in Section 4.1, present an overview of our methodology in Section 4.2, derive for safety in Section 4.3 and derive for progress in Section 4.4.

4.1 Specification

The press (see Fig. 6) consists of a weight that may be lifted or dropped; a lock (whose state is determined by $\mathit{locked} \in \mathbb{B}$); a button (whose state is determined by $\mathit{pressed} \in \mathbb{B}$); a $\mathit{motor} \in \{\mathit{On}, \mathit{Off}\}$; and three sensors that detect the position of the weight: $\mathit{top}, \mathit{pnr}, \mathit{bot} \in \mathbb{B}$ which are located at the top, point of no return and bottom of the press, respectively.

The expected operation of the press is as follows [15]. These requirements are formalised in Fig. 7. If the weight is locked at the top, then the motor must be off and the top sensor must be on (3). When the weight is locked at the top, if the operator presses the button, $\neg\mathit{locked}$ is established (8). The weight may not be released from the top unless the button is pressed (4). The motor remains off (i.e., the weight continues to drop)

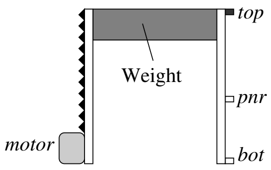


Fig. 6. Industrial press

unless the weight has reached bot or has not passed pnr, and the button is released (5). If the button is released while the weight is dropping but it has not passed pnr, the motor should be turned on (9), which lifts the weight. However, once the dropping weight passes pnr, the motor may not be turned on until the weight reaches bot and the button is no longer pressed (6). If the weight is at bot and the button is released, then the motor is turned on, which lifts the weight (10). Once the weight is lifting, the motor must remain on until the weight is locked at the top (7).

Safety requirements:

$$\Box(\mathit{locked} \Rightarrow \mathit{top} \wedge \mathit{motor} = \mathit{Off}) \tag{3}$$

$$\Box(\mathit{locked} \Rightarrow (\mathit{locked}\ \mathcal{W}\ \mathit{pressed})) \tag{4}$$

$$\begin{aligned}&\Box(\neg \mathit{locked} \wedge \mathit{motor} = \mathit{Off} \Rightarrow \\ &\qquad (\neg \mathit{locked} \wedge \mathit{motor} = \mathit{Off})\ \mathcal{W}\ ((\mathit{bot} \vee \neg \mathit{pnr}) \wedge \neg \mathit{pressed}))\end{aligned} \tag{5}$$

$$\Box(\mathit{pnr} \wedge \mathit{motor} = \mathit{Off} \Rightarrow ((\mathit{pnr} \wedge \mathit{motor} = \mathit{Off})\ \mathcal{W}\ (\mathit{bot} \wedge \neg \mathit{pressed}))) \tag{6}$$

$$\Box(\mathit{motor} = \mathit{On} \Rightarrow (\mathit{motor} = \mathit{On}\ \mathcal{W}\ \mathit{locked})) \tag{7}$$

Progress requirements:

$$(\mathit{locked} \wedge \mathit{pressed}) \rightsquigarrow \neg \mathit{locked} \tag{8}$$

$$(\neg \mathit{locked} \wedge \neg \mathit{pnr} \wedge \neg \mathit{pressed}) \rightsquigarrow \mathit{motor} = \mathit{On} \tag{9}$$

$$(\mathit{bot} \wedge \neg \mathit{pressed}) \rightsquigarrow \mathit{motor} = \mathit{On} \tag{10}$$

Fig. 7. Requirements of the controller

Assumptions on the environment of the controller are formalised by the conditions in Fig. 8. The sensors satisfy the invariant in (11), i.e., if top is on, then both pnr and bot must be off, and if bot is on, then pnr must also be on. Relation (12) describes the allowable transitions of the sensors. For example, if the top sensor is on, then in the next state both pnr and bot must be off, i.e., the environment may not transition directly from a state in which the top sensor is on, to a state where either pnr or bot is on.

RLTL formulae (13) and (14) describe the conditions under which the different sensors are guaranteed to turn on (or off). For example, by (13), if eventually always the weight is not locked at the top and the motor is off (i.e., the weight is falling), then top is guaranteed to turn off, and both pnr and bot are guaranteed to turn on. Note

Safety assumptions:

$$(top \Rightarrow \neg pnr \wedge \neg bot) \wedge (bot \Rightarrow pnr) \tag{11}$$

$$\begin{aligned} top, pnr, bot \mid\ & (top \Rightarrow \neg pnr' \wedge \neg bot') \wedge (\neg top \wedge \neg pnr \Rightarrow \neg bot') \\ & (pnr \Rightarrow \neg top') \wedge (bot \Rightarrow pnr') \end{aligned} \tag{12}$$

Progress assumptions:

$$\Diamond\Box(\neg locked \wedge motor = Off) \Rightarrow \Box\Diamond\neg top \wedge \Box\Diamond pnr \wedge \Box\Diamond bot \tag{13}$$

$$\Diamond\Box(motor = On) \Rightarrow \Box\Diamond top \wedge \Box\Diamond\neg bot \wedge \Box\Diamond\neg pnr \tag{14}$$

Fig. 8. Assumptions on the environment

that although (13) is a formula of the form $\Diamond\Box p \Rightarrow \Box\Diamond q$, it does not require p to eventually be true forever, because the formula can hold if the consequent ($\Box\Diamond q$) becomes true. Ignoring the terms $\Box\Diamond pnr$ and $\Box\Diamond bot$, (13) may be equivalently stated as $\Box\Diamond(locked \vee motor = On \vee \neg top)$.
We define

$$\begin{aligned} Safe &\mathrel{\widehat{=}} (3) \wedge (4) \wedge (5) \wedge (6) \wedge (7) \\ Prog &\mathrel{\widehat{=}} (8) \wedge (9) \wedge (10) \\ RelyProg &\mathrel{\widehat{=}} (13) \wedge (14) \end{aligned}$$

which describe the safety and progress requirements, and progress assumptions of the program. The initial action system representing the controller is thus provided in Fig. 9, where e and c label the environment and controller actions, respectively. The action corresponding to the environment is defined to be

$$env \mathrel{\widehat{=}} pressed \cdot [\![\langle(12)\rangle]\!] \mathrel{!} (11)$$

Action env may modify sensor variables (top, bot, and pnr) as described by relation (12), and the state of the button ($pressed$) arbitrarily. However, the manner in which the variables may be modified is restricted by the enforced safety invariant, (11).

The program in Fig. 9 meets our requirements, but it is not yet executable code.

4.2 Methodology

We now present an overview of the methodology used in our derivation. We start with the temporal formulae in Figs. 7 and 8, which describe requirements on the program as

```
do e: $true \longrightarrow env$
⊓  c: $true \longrightarrow motor, locked \cdot [\![\mathbf{skip}]\!]$
od $?(Safe \wedge Prog \wedge RelyProg)$
```

Fig. 9. Initial action system

a whole and on the environment, respectively. Using these, we arrive at a simple action system (Fig. 9) consisting of actions e and c that represent environment and component transitions, respectively. The frames of e and c describe the variables that e and c may modify, while the enforced properties in Fig. 9 ensure that the frame variables are only modified in a manner consistent with the enforced properties.

We consider safety properties in Section 4.3 where we replace the temporal logic requirements in Fig. 7 by relations. Then, we introduce a new (unobservable) state variable to keep track of the different states of the controller, and calculate modifications for controlling the motor and the lock. We perform the actual modification using our lemmas and theorems in Sections 2 and 3, which ensure that each modification results in a refinement of the original program in Fig. 9.

In Section 4.4, we consider progress properties. We replace temporal properties with relations, introduce fairness constraints, and introduce new actions to satisfy the progress requirements. We perform a number of simplifications to the enforced properties before obtaining the final program in Fig. 10.

4.3 Safety

Replace unless properties. To make better sense of the unless properties within *Safe* and to simplify the rest of the derivation, we use Lemma 9 to replace the unless properties in *Safe* with relations on pre and post states. We present the calculation for the more complicated condition (5). We have

$$\begin{aligned}
& (5) \\
\Leftarrow\quad & \{\text{Lemma 9}\}\{\text{logic}\} \\
& \Box(\neg locked \wedge motor = \mathit{Off} \Rightarrow \\
& \qquad\qquad (\neg locked' \wedge motor' = \mathit{Off}) \vee ((bot' \vee \neg pnr') \wedge \neg pressed') \\
\Leftarrow\quad & \{\text{logic}\} \\
& \Box(\neg locked \wedge motor = \mathit{Off} \Rightarrow \\
& \qquad\qquad \neg locked' \wedge (motor' = \mathit{Off} \vee ((bot' \vee \neg pnr') \wedge \neg pressed'))) \\
\Leftarrow\quad & \{\text{logic}\} \\
& \Box(\neg locked \wedge motor = \mathit{Off} \Rightarrow \neg locked') \wedge \\
& \Box(motor = \mathit{Off} \Rightarrow motor' = \mathit{Off} \vee ((bot' \vee \neg pnr') \wedge \neg pressed'))
\end{aligned}$$

which is logically equivalent to (16). Similarly, we obtain conditions (15), (17) and (18) from (4), (6) and (7), respectively. These properties describe additional constraints on the transitions of the controller and its environment.

$$\Box(locked \wedge \neg locked' \Rightarrow pressed') \tag{15}$$

$$\begin{gathered}\Box(\neg locked \wedge locked' \Rightarrow motor = On) \wedge \\ \Box(motor = \mathit{Off} \wedge motor' = On \Rightarrow (bot' \vee \neg pnr') \wedge \neg pressed')\end{gathered} \tag{16}$$

$$\begin{gathered}\Box(pnr \wedge \neg pnr' \Rightarrow motor = On) \wedge \\ \Box(pnr \wedge motor = \mathit{Off} \wedge motor' = On \Rightarrow bot' \wedge \neg pressed')\end{gathered} \tag{17}$$

$$\Box(motor = On \wedge motor' = \mathit{Off} \Rightarrow locked') \tag{18}$$

We use Lemma 17 (b) to replace *Safe* in the program in Fig. 9 with $(3) \wedge \mathit{Safe}2$ where

$$\mathit{Safe}2 \mathrel{\widehat{=}} (15) \wedge (16) \wedge (17) \wedge (18)$$

Introduce new state variable. To simplify the final program, we introduce a fresh variable $mode$ that allows us to distinguish between the different states of the controller. From the informal description, the controller has a number of modes corresponding to the position of the weight. It can be locked at the top (mode $Idle\_T$), falling (mode $Falling$), resting at the bottom (mode $Idle\_B$), or lifting (mode $Lifting$).

Introduction of variable $mode$ is justified using Lemma 14. Because $mode$ should not be changed by the environment, we use Lemma 15 to remove it from the frame of env. Thus, we obtain an action system where c in Fig. 9 is replaced with

$$true \longrightarrow motor, locked, mode \cdot [\![\mathbf{skip}]\!] \,.$$

Controlling the motor. To control the motor, there will be actions that turn the motor on and off. We derive the conditions under which these actions should occur. We calculate the following condition which ensures the action that sets $motor = On$ maintains (3) $\wedge$ $Safe2$.

$$(motor = Off \Rightarrow (bot' \vee \neg pnr') \wedge \neg pressed') \wedge \neg locked$$

The second conjunct is satisfied by strengthening the guard to $\neg locked$. Because bot, pnr and $pressed$ are not modified by the controller, the first conjunct may be satisfied by strengthening the guard with conjunct $motor = Off \wedge (bot \vee \neg pnr) \wedge \neg pressed$. Thus, we obtain the following action:

$$motor = Off \wedge (bot \vee \neg pnr) \wedge \neg pressed \wedge \neg locked \longrightarrow motor := On$$

which may be split into two cases, respectively as follows:

$$\begin{array}{c} motor = Off \wedge bot \wedge \neg pressed \wedge \neg locked \longrightarrow motor := On \\ \sqcap\ motor = Off \wedge \neg pnr \wedge \neg pressed \wedge \neg locked \longrightarrow motor := On \end{array}$$

Using (11), $bot \Rightarrow \neg top$ holds, and by (3), $\neg top \Rightarrow \neg locked$ holds. Hence, the first guard simplifies to $motor = Off \wedge bot \wedge \neg pressed$. Then, using Lemma 17 (a), we introduce enforced invariants:

$$\Box(\mathsf{Idle\_B} \Rightarrow motor = Off \wedge bot) \tag{19}$$

$$\Box(\mathsf{Falling} \Rightarrow motor = Off \wedge \neg locked) \tag{20}$$

where

$$\begin{array}{l} \mathsf{Idle\_B} \mathrel{\widehat{=}} mode = Idle\_B \\ \mathsf{Falling} \mathrel{\widehat{=}} mode = Falling \end{array}$$

denote the "idle at bottom" and "falling" modes, respectively. Invariants (19) and (20) allow us to simplify the guards of the action above and we obtain:

(C1) $\mathsf{Idle\_B} \wedge \neg pressed \longrightarrow motor, mode := On, Lifting$

(C2) $\mathsf{Falling} \wedge \neg pnr \wedge \neg pressed \longrightarrow motor, mode := On, Lifting$

Furthermore C1 and C2 set $mode$ to $Lifting$. Applying similar reasoning to the action that turns the motor off, using Lemma 17, we introduce enforced invariant

$$\Box(\mathsf{Lifting} \Rightarrow motor = On) \tag{21}$$

and obtain action C3 below.

(C3) $\mathsf{Lifting} \wedge top \longrightarrow locked, motor, mode := true, Off, Idle\_T$

Controlling the lock. We now calculate the actions that modify variable $locked$. Recall that the action that sets $locked$ to true has already been introduced as C3 above. From our calculation against (3) $\wedge$ $Safe2$, we obtain requirement

$$pressed' \wedge (motor = Off \vee motor' = On)$$

We then introduce the following enforced property using Lemma 17

$$\Box(\mathsf{Idle\_T} \Rightarrow locked \wedge motor = Off) \tag{22}$$

which denotes a mode where the weight is idle at the top. Thus, we obtain the following action, which also sets $mode$ to $Falling$.

(C4) $\mathsf{Idle\_T} \wedge pressed \longrightarrow locked, mode := false, Falling$

Introduce actions. We have calculated the actions C1, C2, C3 and C4 that modify $motor$ and $locked$. Each action C1-C4 refines c, and hence may be introduced to the action system thus far using Corollary 7 and Lemma 17 (c). Then, using Lemma 15, we remove $motor$ and $locked$ from the frame of each statement. Thus, we obtain the following refined action system where

$$ES \mathrel{\widehat{=}} (19) \wedge (20) \wedge (21) \wedge (22)$$

is the conjunction of the newly introduced enforced properties.

```
do $e$:  $true \longrightarrow env$
⊓  $c$:  $true \longrightarrow mode \cdot [\![\mathbf{skip}]\!]$
⊓  $c_1$: $\mathsf{Idle\_B} \wedge \neg pressed \longrightarrow motor, mode := On, Lifting$
⊓  $c_2$: $\mathsf{Falling} \wedge \neg pressed \wedge \neg pnr \longrightarrow motor, mode := On, Lifting$
⊓  $c_3$: $\mathsf{Lifting} \wedge top \longrightarrow motor, locked, mode := Off, true, Idle\_T$
⊓  $c_4$: $\mathsf{Idle\_T} \wedge pressed \longrightarrow locked, mode := false, Falling$
od ?((3) $\wedge$ $Safe2 \wedge Prog \wedge RelyProg \wedge ES$)
```

Note that this program can only establish $mode = Idle\_B$ via action c, which sets $mode$ to an arbitrary value that maintains the enforced properties. We derive the action that establishes $mode = Idle\_B$ in Section 4.4.

4.4 Progress

We verify $Prog$ using Lemma 10, and hence we need to introduce the following fairness assumption as an enforced property to the action system:

$$WFair \mathrel{\widehat{=}} WFair.e \wedge WFair.c \wedge \bigwedge_i WFair.c_i$$

Recall that the label of a guarded action represents the guarded action. Thus, *WFair* states that each top-level action of the action system is weakly fair. We verify the more difficult progress property, (10), which requires introduction of an additional action. To prove (10), using Lemma 17 (a), we introduce the following enforced property to the program:

$$\Box(bot \wedge motor = Off \Rightarrow \mathsf{Falling} \vee \mathsf{Idle\_B}) \tag{23}$$

and obtain the following calculation:

$$\begin{array}{ll} & (10) \\ \Leftarrow & \quad \{(23)\}\{\text{Lemma 10 (c)}\} \\ & bot \wedge \neg pressed \wedge motor = Off \wedge (\mathsf{Falling} \vee \mathsf{Idle\_B}) \leadsto motor = On \\ \Leftarrow & \quad \{\text{temporal logic}\}\{(19) \text{ and } (23)\} \\ & (bot \wedge \neg pressed \wedge \mathsf{Falling} \leadsto \neg pressed \wedge \mathsf{Idle\_B}) \wedge \\ & (\neg pressed \wedge \mathsf{Idle\_B} \leadsto motor = On) \end{array}$$

The first conjunct holds by Lemma 10 if we introduce enforced property

$$\begin{array}{l} \Box(bot \wedge \neg pressed \wedge \mathsf{Falling} \Rightarrow \\ \qquad \neg pressed' \wedge ((bot' \wedge mode' = Falling) \vee mode' = Idle\_B) \end{array} \tag{24}$$

which by Lemma 9 implies the unless condition of Lemma 10 (a). The lemma also requires an action that establishes $\neg pressed \wedge \mathsf{Idle\_B}$ when $bot \wedge \neg pressed \wedge \mathsf{Falling}$ holds. Thus, using Corollary 7 and Lemma 17 (c), we introduce action

(C5) $\qquad \mathsf{Falling} \wedge bot \longrightarrow mode := Idle\_B$

to the action system.

The second conjunct holds by Lemma 10 (a) together with Lemma 9 if we introduce enforced property

$$\begin{array}{l} \Box(\neg pressed \wedge \mathsf{Idle\_B} \Rightarrow \\ \qquad (\neg pressed' \wedge mode' = Idle\_B) \vee motor' = On) \end{array} \tag{25}$$

and let action c_1 be the action that establishes $motor = On$. Note that (25) is not necessarily satisfied by an environment that repeatedly presses and releases the button before the controller is able to react. By enforcing (25), we are stating that progress condition (10) only holds if the environment leaves the button unpressed whenever the weight is at the bottom, unless the motor is turned on (i.e., the controller has had an opportunity to react). Similarly, for (8), we introduce enforced properties:

$$\Box(locked \Rightarrow \mathsf{Idle\_T}) \tag{26}$$

$$\Box(\mathsf{Idle\_T} \wedge pressed \Rightarrow (mode' = Idle\_T \wedge pressed') \vee \neg locked') \tag{27}$$

and let c_4 be the action that falsifies *locked*. The issues with *pressed* in (27) are similar to those in (25).

For (9), we introduce enforced properties

$$\Box(\neg locked \wedge \neg pnr \wedge motor = \mathit{Off} \Rightarrow \mathsf{Falling}) \tag{28}$$

$$\begin{array}{l}\Box(\mathsf{Falling} \wedge \neg pnr \wedge \neg pressed \Rightarrow \\ \quad (mode' = \mathit{Falling} \wedge \neg pnr' \wedge \neg pressed') \vee motor' = On)\end{array} \tag{29}$$

and let c_2 be the action that establishes $motor = On$. Unlike (25) and (27), which describe restrictions on the user input, property (29) consists of an additional constraint on the sensor values that cannot be satisfied by any implementation. For example, the button may be released when the weight is just above the pnr sensor causing $\mathsf{Falling} \wedge \neg pnr \wedge \neg pressed$ to become true. However, by the time the controller reacts, the weight may have already passed pnr, which means (29) does not hold. If the button is released just above the pnr sensor, there is a race condition between the weight reaching the pnr sensor (pnr becoming true) and the controller detecting that the button has been released before the weight reaches the pnr sensor. Thus, we have discovered that (9) is an unimplementable requirement. Nevertheless, in those instances in which the environment does behave as described by (29), the final program in Fig. 10 satisfies (9). To provide a requirement similar to (9) that is implementable, we would have to use a timed model and allow timing tolerance. This is beyond the scope of this paper.

We define

$$\begin{array}{r}EP \mathrel{\widehat{=}} (23) \wedge (28) \wedge (26) \\ EPR \mathrel{\widehat{=}} (24) \wedge (25) \wedge (29) \wedge (27)\end{array}$$

to be the temporal formulae on predicates and relations, respectively. We have shown that $Prog$ holds if we introduce EP and EPR to the program. Thus, we use Lemma 17 (d) to remove $Prog$ from the program. Note that the program satisfies $Prog$, but it no longer needs to be enforced.

Final modifications. We now simplify the action system by removing as many of the unnecessary actions and enforced properties as possible. We use Lemma 15 to remove $mode$ from the frame of c. Then, using Lemma 5 we remove action $true \longrightarrow$ **skip** from the action system altogether.

We introduce function ρ which is defined for RLTL formulae of the form $\bigwedge_i \Box R_i$. We define $\rho.(\Box R) \mathrel{\widehat{=}} R$ for any RLTL formula R and $\rho.(R_1 \wedge R_2) \mathrel{\widehat{=}} \rho.R_1 \wedge \rho.R_2$ for RLTL formulae R_1 and R_2.

Using Lemma 19 then Lemma 18, we distribute enforced properties $!\rho.((3) \wedge ES \wedge EP)$ and $!\rho.(Safe2 \wedge EPR)$ within the body of the **do** loop. By the derivation above, each action c_1-c_5 satisfies $\rho.(Safe2 \wedge ES \wedge EP)$ and $\rho.EPR$, and hence, using parts 3 and 4 of Lemma 18, we may remove all '!' properties on c_1-c_5 (see in Fig. 10).

Due to the restrictions on the variables that e may modify, we are presented with an opportunity to simplify the '!' properties on e. Because e does not modify $locked$, $motor$, or $mode$, $\rho.(3)$, $\rho.Safe2$, and $\rho.ES$ reduce to:

$$(top \wedge \neg top' \Rightarrow \neg locked) \wedge ((pnr \wedge \neg pnr') \vee (bot \wedge \neg bot') \Rightarrow motor = On) \tag{30}$$

Essentially, (30) describes some additional constraints on the sensors. Similarly for $\rho.EP$, condition $\rho.(26)$ is trivially satisfied because e does not modify *locked* or *mode*, while $\rho.((23) \wedge (28))$ is satisfied by the second conjunct of (30).

Condition $\rho.EPR$ is necessary for progress and describes requirements on the state of the button. It disallows the state of the button from changing until the controller has reacted to a press or release of the button. Unlike $\rho.((3) \wedge ES \wedge EP \wedge Safe2)$, condition $\rho.EPR$ cannot be simplified any further. By the calculations above, we have:

$$env \,!\, \rho.((3) \wedge ES \wedge EP) \,!\, \rho.(Safe2 \wedge EPR) \;\sqsubseteq\; env \,!\, (30) \,!\, \rho.EPR$$

Thus, using (6), we may replace $env \,!\, \rho.((3) \wedge ES \wedge EP) \,!\, \rho.(Safe2 \wedge EPR)$ in e by the simpler $env \,!\, (30) \,!\, \rho.EPR$ and obtain the program in Fig. 10.

```
do $e$:  $true \longrightarrow env \,!\, (30) \,!\, \rho.EPR$
⊓  $c_1$: $\mathsf{Idle\_B} \wedge \neg pressed \longrightarrow motor, mode := On, Lifting$
⊓  $c_2$: $\mathsf{Falling} \wedge \neg pressed \wedge \neg pnr \longrightarrow motor, mode := On, Lifting$
⊓  $c_3$: $\mathsf{Lifting} \wedge top \longrightarrow motor, locked, mode := Off, true, Idle\_T$
⊓  $c_4$: $\mathsf{Idle\_T} \wedge pressed \longrightarrow locked, mode := false, Falling$
⊓  $c_5$: $\mathsf{Falling} \wedge bot \longrightarrow mode := Idle\_B$
od $?(WFair \wedge RelyProg)$
```

Fig. 10. Final action system

A similar process can be used to derive the initialisation code for the action system, which must guarantee to establish $\rho.((3) \wedge ES \wedge EP)$.

The action system we have derived makes the assumption that all sampling is perfect which does not accurately reflect the real world. For example, for condition (9), the program may not result in lifting behaviour because the controller may miss the state $\neg locked \wedge \neg pnr \wedge \neg pressed$ because the state only occurred transiently just before the weight reached the point of no return, and the sampling of the sensors did not detect the transient state. For example, suppose the weight is falling and the environment transitions such that the button is released, the weight passes pnr, and the button is pressed. The intermediate state between these transitions will not be detected by the controller unless it makes a transition in that intermediate state. Troubitsyna ensures that the controller does not miss transient states by strictly alternating between the controller and its environment, i.e., disallowing the environment from making multiple transitions [24,25]. We feel that this requirement is too strict and prefer our approach where problems in the specification are elucidated by the derivation, as opposed to being hidden by restrictions on the order of execution of the program.

Other issues arise from the fact that we assume all sampling takes place simultaneously because all the guards of the action system are evaluated in a single atomic step. In a real implementation, when sampling multiple sensors, the environment may modify the sensor values between sampling events, causing non-existing states to be detected. Addressing such issues requires more sophisticated timing properties, which are beyond the scope of this paper. As part of future work, we will explore whether sampling issues can be better addressed using specialised sampling logics.

5 Conclusion

We have extended action systems with enforced properties and frames which facilitates formal derivation of their code from a specification. We have used our developed theory to derive a controller for an industrial press. Extending action systems with enforced properties has allowed the theory from [9,10] to be simplified and we have shown how an action system that satisfies safety and progress may be derived. Using a more general temporal logic over relations has allowed us to derive the action systems in a compositional manner. We have also shown how enforced properties may be used to formalise environmental assumptions.

In normal refinement, the introduction of a new variable is tightly coupled with the operations and invariants that refer to the variable, and hence all operations and invariants that use a new variable must be introduced at the same time. As a result, each refinement step can become complex and thus difficult to prove [1]. Our refinement technique involves actions with frames and enforced properties, which may be manipulated independently of each other. Hence we achieve a decoupling between variables and operations that modify the variables, allowing refinement via a series of small steps.

Enforced properties allow fairness to be formally specified in a straightforward manner. Our treatment of fairness is simpler than refinement methods described by Back and Xu [5] and Wabenhorst [26]. Algebraic methods for refinement under fair choice are described by Sekerinski [23] who introduces different forms of fair non-deterministic choice operators. Our method achieves the same result without the need for specialised operators, which makes our techniques more general.

Acknowledgements. This research was supported by Australian Research Council Discovery Grant, *Combining Time Bands and Teleo-Reactive programs for Advanced Dependable Real-Time Systems* (DP0987452) and The University of Queensland New Staff Research Fund.

References

1. Abrial, J.R., Cansell, D., Méry, D.: Refinement and reachability in Event B. In: Treharne, H., King, S., Henson, M.C., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 222–241. Springer, Heidelberg (2005)
2. Back, R.J., von Wright, J.: Trace refinement of action systems. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 367–384. Springer, Heidelberg (1994)
3. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, New York (1998)
4. Back, R.J., von Wright, J.: Compositional action system refinement. Formal Asp. Comput. 15(2-3), 103–117 (2003)
5. Back, R.J., Xu, Q.: Refinement of fair action systems. Acta Inf. 35(2), 131–165 (1998)
6. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (1988)
7. Coleman, J.W., Jones, C.B.: A structural proof of the soundness of rely/guarantee rules. J. Log. Comput. 17(4), 807–841 (2007)
8. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18(8), 453–457 (1975)

9. Dongol, B.: Progress-based verification and derivation of concurrent programs. Ph.D. thesis, The University of Queensland (2009)
10. Dongol, B., Hayes, I.J.: Enforcing safety and progress properties: An approach to concurrent program derivation. In: 20th Australian Software Engineering Conference, pp. 3–12. IEEE Computer Society, Los Alamitos (2009)
11. Dongol, B., Mooij, A.J.: Progress in deriving concurrent programs: Emphasizing the role of stable guards. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 140–161. Springer, Heidelberg (2006)
12. Dongol, B., Mooij, A.J.: Streamlining progress-based derivations of concurrent programs. Formal Aspects of Computing 20(2), 141–160 (2008)
13. Feijen, W.H.J., van Gasteren, A.J.M.: On a Method of Multiprogramming. Springer, Heidelberg (1999)
14. Groslambert, J.: Verification of LTL on B event systems. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 109–124. Springer, Heidelberg (2006)
15. Hayes, I.J.: Dynamically detecting faults via integrity constraints. In: Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.) Methods, Models and Tools for Fault Tolerance. LNCS, vol. 5454, pp. 85–103. Springer, Heidelberg (2009)
16. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Transactions on Programming Languages and Systems 5(4), 596–619 (1983)
17. Jonsson, B., Tsay, Y.K.: Assumption/guarantee specifications in linear-time temporal logic. Theoretical Computer Science 167(1-2), 47–72 (1996)
18. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
19. Manna, Z., Pnueli, A.: Temporal Verification of Reactive and Concurrent Systems: Specification. Springer, New York (1992)
20. McDermid, J., Kelly, T.: Industrial press: Safety case. Tech. rep., High Integrity Systems Engineering Group, University of York (1996)
21. Morgan, C.: Programming from specifications, 2nd edn. Prentice Hall International (UK) Ltd., Hertfordshire (1994)
22. Morgan, C., Vickers, T.: On the Refinement Calculus. Springer, New York (1992)
23. Sekerinski, E.: An algebraic approach to refinement with fair choice. Electr. Notes Theor. Comput. Sci. 214, 51–79 (2008)
24. Troubitsyna, E.: Enhancing dependability via parameterized refinement. In: PRDC, p. 120. IEEE Computer Society, Los Alamitos (1999)
25. Troubitsyna, E.: Reliability assessment through probabilistic refinement. Nord. J. Comput. 6(3), 320–342 (1999)
26. Wabenhorst, A.: Stepwise development of fair distributed systems. Acta Inf. 39(4), 233–271 (2003)
27. Woodcock, J., Davies, J.: Using Z: specification, refinement, and proof. Prentice-Hall, Inc., Upper Saddle River (1996)

Designing an Algorithmic Proof of the Two-Squares Theorem

João F. Ferreira*

School of Computer Science
University of Nottingham
Nottingham NG8 1BB, England
joao@joaoff.com

Abstract. We show a new and constructive proof of the two-squares theorem, based on a somewhat unusual, but very effective, way of rewriting the so-called extended Euclid's algorithm. Rather than simply verifying the result — as it is usually done in the mathematical community — we use Euclid's algorithm as an interface to *investigate* which numbers can be written as sums of two positive squares. The precise formulation of the problem as an algorithmic problem is the key, since it allows us to use algorithmic techniques and to avoid guessing. The notion of invariance, in particular, plays a central role in our development: it is used initially to observe that Euclid's algorithm can actually be used to represent a given number as a sum of two positive squares, and then it is used throughout the argument to prove other relevant properties. We also show how the use of program inversion techniques can make mathematical arguments more precise.

Keywords: algorithm derivation, sum of two squares, Euclid's algorithm, invariant, program inversion.

1 Introduction

Which numbers can be written as sums of two squares? According to Dickson [1, p. 225], this classic question in number theory was first discussed by Diophantus, but it is usually associated with Fermat, who stated in 1659 that he possessed an irrefutable proof that every prime of the form $4k + 1$ can be written as the sum of two squares. (He first communicated the result to Mersenne, in a letter dated December 25, 1640; for this reason, this result is sometimes called *Fermat's Christmas Theorem*. Incidentally, Dickson names this result after Albert Girard, who, in 1632, was the first to state it. We follow Dickson's convention and we also refer to the two-squares theorem as Girard's result.) However, as with many other of his results, Fermat did not record his proof. The first recorded proof of Girard's result is due to Euler who proved it in 1749, "after he had struggled, off and on, for *seven years* to find a proof" [2, p. 69]. Euler communicated his

* Funded by Fundação para a Ciência e a Tecnologia (Portugal) under grant SFRH/BD/24269/2005.

five-step argument in a letter to Goldbach dated 6 May 1747, but the fifth step was only made precise in a second letter written in 1749. In 1801, Gauss proved for the first time that such prime numbers are *uniquely* represented as the sum of two positive integers [3, Art. 182].

This classic theorem attracted the attention of many mathematicians. Since Euler's proof by the method of infinite descent, Lagrange proved it using quadratic forms (subsequently, Gauss simplified Lagrange's proof in [3, Art. 182]); Dedekind used Gaussian integers; Serret and Hermite used continued fractions [4,5]; Brillhart improved Hermite's argument using Euclid's algorithm [6]; Smith used continuants [7]; more recently, Zagier [8] published a one-sentence proof based on an involution of a particular finite set (see also [9, chapter 4] and [10] for a detailed explanation of the proof); and Wagon [11] gave a self-contained proof based on Euclid's algorithm and on [6].

Like Brillhart and Wagon, we present a proof that is also based on Euclid's algorithm, but, rather than simply verifying Girard's result, we use the algorithm as an interface to *investigate* which numbers can be written as sums of two positive squares[1]. The precise formulation of the problem as an algorithmic problem is the key, since it allows us to use algorithmic techniques and to avoid guessing. The notion of invariance, in particular, plays a central role in our development: it is used initially to observe that Euclid's algorithm can actually be used to represent a given number as a sum of two positive squares, and then it is used throughout the argument to prove other relevant properties. We also show how the use of program inversion techniques can make mathematical arguments more precise. As we will see, the end result is also more general than the one conjectured by Girard.

In the next section we show how we can use our formulation of Euclid's algorithm to prove the theorem. At the end of the section, we describe how the argument and the paper are organised.

2 Euclid's Algorithm

We start with 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. (This is sometimes called the extended Euclid's algorithm. See [12] for a derivation of the algorithm and [13] for another problem whose solution is based on the algorithm. Also, we use "∇" to denote "greatest common divisor". We prefer to use an infix notation whenever — as in this case — the operator is symmetric and associative).

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 $(m\nabla n\ \ m\nabla n)$. (This is a

[1] Every square number m^2 can be written as $m^2{+}0^2$. However, this type of solution is not considered in this paper, since our formulation of Euclid's algorithm deals only with positive numbers. Therefore, our construction aims to express a number as the sum of two *positive* squares.

consequence of the invariant $(x\ y) = (m\nabla n\ \ m\nabla n)$. We omit this invariant in the comments below to simplify the presentation.) 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 $2{\times}2$ matrices; $\mathbf{I}$ is the identity matrix $\left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$, $\mathbf{A}$ is the matrix $\left(\begin{smallmatrix}1 & 0\\ -1 & 1\end{smallmatrix}\right)$, and $\mathbf{B}$ is the matrix $\left(\begin{smallmatrix}1 & -1\\ 0 & 1\end{smallmatrix}\right)$. The assignment $(x\ y) := (x\ y){\times}\mathbf{A}$ is equivalent to $x\,,y := x{-}y\,,y$, as can be easily checked.

```
$\{\ 0<m\ \wedge\ 0<n\ \}$
$(x\ y)\,,\mathbf{C} := (m\ n)\,,\mathbf{I}$ ;
$\{$ **Invariant**: $(x\ y) = (m\ n)\times\mathbf{C}\ \}$
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
$\{\ (x\ y) = (m\nabla n\ \ m\nabla n) = (m\ n)\times\mathbf{C}\ \}$
```

The verification of the supplied invariant is a simple consequence of the associativity of matrix multiplication. Also, note that the algorithm constructs two linear combinations of m and n equal to their greatest common divisor.

A key insight in our development is that matrices $\mathbf{A}$ and $\mathbf{B}$ are invertible, which allows us to rewrite the invariant as $(x\ y)\times\mathbf{C}^{-1} = (m\ n)$, where the matrix $\mathbf{C}^{-1}$ is a finite product of the matrices $\mathbf{A}^{-1}$ and $\mathbf{B}^{-1}$, which are, respectively, $\left(\begin{smallmatrix}1 & 0\\ 1 & 1\end{smallmatrix}\right)$ and $\left(\begin{smallmatrix}1 & 1\\ 0 & 1\end{smallmatrix}\right)$. In fact, we can change the above algorithm to compute the matrix $\mathbf{C}^{-1}$ instead; renaming $\mathbf{C}^{-1}$ to $\mathbf{D}$, $\mathbf{A}^{-1}$ to $\mathbf{L}$, and $\mathbf{B}^{-1}$ to $\mathbf{R}$, we rewrite it as follows:

```
$\{\ 0<m\ \wedge\ 0<n\ \}$
$(x\ y)\,,\mathbf{D} := (m\ n)\,,\mathbf{I}$ ;
$\{$ **Invariant**: $(x\ y)\times\mathbf{D} = (m\ n)\ \}$
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
$\{\ (x\ y) = (m\nabla n\ \ m\nabla n)\ \ \wedge\ \ (m\nabla n\ \ m\nabla n)\times\mathbf{D} = (m\ n)\ \}$
```

It is this form of the algorithm that is the starting point for our investigation. Note that if $\mathbf{D}{=}\left(\begin{smallmatrix}a & b\\ c & d\end{smallmatrix}\right)$, the invariant is equivalent to

$$(m\ n) \;=\; (x\ y)\times\mathbf{D} \;=\; (x{\times}a+y{\times}c\ \ \ x{\times}b+y{\times}d)\quad ,$$

which means that if, at any point in the execution of the algorithm, $(x\ y)$ equals $(a\ c)$, we can conclude that m is a sum of two positive squares, that is:

$$(m\ n) \quad = \quad (a\ c) \times \mathbf{D} \quad = \quad (a{\times}a + c{\times}c \quad a{\times}b + c{\times}d) \quad .$$

Symmetrically, if, at any point in the execution of the algorithm, $(x\ y)$ equals $(b\ d)$, we can conclude that n is a sum of two positive squares.

It may help to visualise an execution trace of the algorithm. Table 1 depicts the execution trace when the arguments are $m = 17$ and $n = 4$. Each row of the table shows the state-space and the value of the invariant after each iteration of the algorithm. The first two columns show the values of the variables $(x\ y)$ and $\mathbf{D}$, respectively. The third column shows how the invariant is satisfied, according to the values of the first two columns. The first row corresponds to the initial state and the last row corresponds to the final state.

Table 1. Execution trace of Euclid's algorithm for arguments $m = 17$ and $n = 4$

$(x\ y)$	$\mathbf{D}$, the same as $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$	**Invariant:** $(m\ n) = (x{\times}a + y{\times}c \quad x{\times}b + y{\times}d)$
$(17\ 4)$	$\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right) = \mathbf{I}$	$(17\ 4) = (17{\times}1 + 4{\times}0 \quad 17{\times}0 + 4{\times}1)$
$(13\ 4)$	$\left(\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right) = \mathbf{L}$	$(17\ 4) = (13{\times}1 + 4{\times}1 \quad 13{\times}0 + 4{\times}1)$
$(9\ 4)$	$\left(\begin{smallmatrix} 1 & 0 \\ 2 & 1 \end{smallmatrix}\right) = \mathbf{LL}$	$(17\ 4) = (9{\times}1 + 4{\times}2 \quad 9{\times}0 + 4{\times}1)$
$(5\ 4)$	$\left(\begin{smallmatrix} 1 & 0 \\ 3 & 1 \end{smallmatrix}\right) = \mathbf{LLL}$	$(17\ 4) = (5{\times}1 + 4{\times}3 \quad 5{\times}0 + 4{\times}1)$
$(\mathbf{1}\ \mathbf{4})$	$\left(\begin{smallmatrix} \mathbf{1} & 0 \\ \mathbf{4} & 1 \end{smallmatrix}\right) = \mathbf{LLLL}$	$(17\ 4) = (\mathbf{1}{\times}\mathbf{1} + \mathbf{4}{\times}\mathbf{4} \quad 1{\times}0 + 4{\times}1)$
$(1\ 3)$	$\left(\begin{smallmatrix} 5 & 1 \\ 4 & 1 \end{smallmatrix}\right) = \mathbf{RLLLL}$	$(17\ 4) = (1{\times}5 + 3{\times}4 \quad 1{\times}1 + 3{\times}1)$
$(1\ 2)$	$\left(\begin{smallmatrix} 9 & 2 \\ 4 & 1 \end{smallmatrix}\right) = \mathbf{RRLLLL}$	$(17\ 4) = (1{\times}9 + 2{\times}4 \quad 1{\times}2 + 2{\times}1)$
$(1\ 1)$	$\left(\begin{smallmatrix} 13 & 3 \\ 4 & 1 \end{smallmatrix}\right) = \mathbf{RRRLLLL}$	$(17\ 4) = (1{\times}13 + 1{\times}4 \quad 1{\times}3 + 1{\times}1)$

As we can see in table 1, there is a point at which $x = a = 1$ and $y = c = 4$; it follows directly from the invariant that 17 can be expressed as the sum of two positive squares ($17 = 1^2{+}4^2$).

One question that now arises is what is so special about the numbers 17 and 4 that made the vectors $(x\ y)$ and $(a\ c)$ to be equal. (Had we used as arguments the numbers 17 and 5, for example, x would never equal a.) Put more generally, how can we characterise the arguments that make the vectors $(x\ y)$ and $(a\ c)$ to be equal at some point in the execution of the algorithm?

A closer inspection of the values shown in table 1 can help us answering the general question. If we ignore the first row, we see that the sequence of successive values of the vector $(x\ y)$ is the reverse of the sequence of successive values of $(a\ c)$. Also, because the length of these sequences is the same and odd, there is a middle point at which $(x\ y)$=$(a\ c)$. So, one way of proving that at some point in the execution of the algorithm the vectors $(x\ y)$ and $(a\ c)$ are equal is to prove that the sequences of successive values of the vectors $(x\ y)$ and $(a\ c)$, with the exception of the initial values, are reverses of each other and that both sequences have odd length. (In the example above, the length is 7).

Taking this analysis into account, the question can be reformulated as: for which arguments m and n does Euclid's algorithm produce odd-length sequences of successive values of the vectors $(x\ y)$ and $(a\ c)$ that are reverse of each other?

Our answer to this question is divided in three parts. First, in section 3, we invert Euclid's algorithm to prove that the operations performed on the vector $(x\ y)$ are the same as those performed on the vector $(a\ c)$ when running the algorithm backwards. Second, in section 4, we determine necessary and sufficient conditions on the arguments m and n to make the initial value of the vector $(x\ y)$ equal the final value of the vector $(a\ c)$. These two parts together characterise the arguments for which the sequences of vectors are each other's reverses. Finally, in section 5, we show that if the sequences are the reverses of each other, they must have odd length.

Note that our investigation aims at expressing the argument m as a sum of two positive squares — that is why we focus on vectors $(x\ y)$ and $(a\ c)$. This means that, given a value m, we want to characterise which values n can be chosen to be passed along with m as arguments of the algorithm (we perform this characterisation in section 4).

For brevity, and whenever the context is unambiguous, we shall refer to "the sequences" to mean "the sequences of successive values of the vectors $(x\ y)$ and $(a\ c)$" and to "the sequences are reversed" to mean "the sequences are the reverses of each other". Also, we assume throughout that $\mathbf{D} = \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$.

3 Inverting Euclid's Algorithm

Inverting an algorithm S consists in finding another algorithm, usually denoted by S^{-1}, that when composed with S leaves the program state unchanged. In other words, executing S^{-1} after S amounts to doing nothing, that is, if we provide to S^{-1} some output of S, it will compute a corresponding input to S.

Some statements are easy to invert. The inverse of skip, for example, is skip itself. Also, the inverse of $x := x{-}y$ is $x := x{+}y$. However, other statements are difficult or impossible to invert. For example, we cannot invert $x := 1$ without knowing the value of x before the assignment; we can only invert it if we know the precondition. The inverse of

$$\{\ x = 0\ \}\ \ x\ :=\ 1$$

is

$$\{\ x = 1\ \}\ \ x\ :=\ 0\ \ .$$

Note that the assertion becomes an assignment and the assignment becomes an assertion. This simple example shows that we may be able to compute inverses only when the precondition is given. Therefore, we define the inverse of a statement with respect to a precondition. That is, S^{-1} is the (right) inverse of S with respect to R, if for every Q

$$\{\ R \wedge Q\ \}\ \ S\,;S^{-1}\ \ \{\ Q\ \}\ \ .$$

An important aspect of the above characterisation is that it distributes through program constructs. This allows us to reduce the inversion of a program into the inversion of its components. For example, the inverse of a sequence of commands is the reverse of the sequence of inverses of the individual commands:

$$(S_0;S_1;\cdots;S_n)^{-1}\ =\ S_n^{-1};\cdots;S_1^{-1};S_0^{-1}\quad .$$

Also, if c_0 and c_1 are constants, the inverse of

(1) $\quad v\ :=\ c_0\ \ ;\ \ S\ \ \{\ v = c_1\ \}$

is

$$v\ :=\ c_1\ \ ;\ \ S^{-1}\ \ \{\ v = c_0\ \}\quad .$$

In (1), variable v is initialised to a value c_0, S is executed, and upon termination v has the final value c_1. The inverse assigns c_1 to v, undoes what S did, and terminates with $v = c_0$. Note, again, how the assignment and the assertion switch places. Since Euclid's algorithm is an instance of (1) — instantiate v with the variables $(x\ y)$ and $\mathbf{D}$, and consider S to be the loop — its inverse is:

$$\begin{array}{l} (x\ y)\ :=\ (m\nabla n\ \ m\nabla n)\ ; \\ \text{initialise } \mathbf{D} \text{ such that } (m\nabla n\ \ m\nabla n)\times\mathbf{D}\ =\ (m\ n)\,; \\ S^{-1} \\ \{\ \ (x\ y)\ =\ (m\ n)\quad\wedge\quad \mathbf{D}\ =\ \mathbf{I}\ \ \}\ \ . \end{array}$$

That is, provided that we initialise $(x\ y)$ to $(m\nabla n\ \ m\nabla n)$ and the matrix $\mathbf{D}$ in a way that satisfies $(m\nabla n\ \ m\nabla n)\times\mathbf{D}\ =\ (m\ n)$, undoing S terminates in a state where $(x\ y)$ and $\mathbf{D}$ equal their initial values in Euclid's algorithm. But we still have to guarantee that there is only one way of initialising $\mathbf{D}$. This is indeed the case, since

$$\begin{array}{ll} & (m\nabla n\ \ m\nabla n)\times\mathbf{D}\ =\ (m\ n) \\ = & \\ & (1\ \ 1)\times\mathbf{D}\ =\ (m/(m\nabla n)\ \ n/(m\nabla n)), \end{array}$$

where $(m/(m\nabla n)\ \ n/(m\nabla n))$ can be seen as a positive rational number in so-called lowest-form representation. We know from [12] (see also [13]) that there is a bijection between finite products of the matrices $\mathbf{L}$ and $\mathbf{R}$ and the positive rationals. Therefore, $\mathbf{D}$ (which is a finite product of $\mathbf{L}$s and $\mathbf{R}$s) is uniquely defined (more specifically, it represents the path from the origin to the rational $\frac{n/(m\nabla n)}{m/(m\nabla n)}$ in the Stern-Eisenstein tree of rationals).

Now, since the alternative statement in the loop of Euclid's algorithm is deterministic ($y < x$ and $x < y$ are mutually exclusive), we can use the inversion rule for deterministic alternative statements together with the inversion rule for iterative statements. Suppose we have the following loop

```
{ $G_0 \vee G_1$ }
do $G_0$ → $S_0$ { $C_0$ }
□ $G_1$ → $S_1$ { $C_1$ }
od
{ $C_0 \vee C_1$ } .
```

Execution of the loop must begin with one of the guards true, so the disjunction of the guards has been placed before the statement. Execution terminates with either C_0 or C_1 true, depending on which command is executed, so $C_0 \vee C_1$ is the postcondition. Also, to invert this loop we must know whether to perform the inverse of S_0 or to perform the inverse of S_1. Therefore, C_0 and C_1 cannot be true at the same time (i.e., $\neg(C_0 \wedge C_1)$). For symmetry, we also require $\neg(G_0 \wedge G_1)$.

Because the loop ends in a state satisfying $C_0 \vee C_1$, its inverse must begin in a state satisfying $C_0 \vee C_1$. Also, execution of $G_1 \rightarrow S_1 \ \{ C_1 \}$ means that beginning with G_1 true, S_1 is executed, and C_1 is established. The inverse must express that beginning with C_1 true, S_1 is undone, and G_1 is established:

$$C_1 \rightarrow S_1^{-1} \ \{ G_1 \} \quad .$$

Note how, when inverting a guarded command with a postcondition, the guard and postcondition switch places. Continuing to read backwards yields the inverse of the loop:

```
{ $C_0 \vee C_1$ }
do $C_1$ → $S_1^{-1}$ { $G_1$ }
□ $C_0$ → $S_0^{-1}$ { $G_0$ }
od
{ $G_0 \vee G_1$ } .
```

We now have to insert appropriate assertions in Euclid's algorithm so that the rules presented above can be used. Recall that, as explained in section 2, we want to ignore the initial values (in effect, this corresponds to ignoring the first row of table 1). This motivates moving the first step out of the loop body. Assuming that $n < m$, we can rewrite the algorithm as follows (note the new annotations and recall that $\mathbf{D} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$):

```
{ $0 < n < m$ }
$(x\ y), \mathbf{D} := (m-n\ \ n), \mathbf{L}$ ;
{ **Invariant**: $(x\ y) \times \mathbf{D} = (m\ n)$ }
{ $y < x \ \vee\ x < y$ }
do $y < x$ → $(x\ y), \mathbf{D} := (x\ y) \times \mathbf{L}^{-1}\ ,\ \mathbf{L} \times \mathbf{D}$ { $a < c$ }
□ $x < y$ → $(x\ y), \mathbf{D} := (x\ y) \times \mathbf{R}^{-1}\ ,\ \mathbf{R} \times \mathbf{D}$ { $c < a$ }
```

```
od
{ $a<c \vee c<a$ }
{ $(x\ y) = (m\nabla n\ \ m\nabla n) \quad \wedge \quad (m\nabla n\ \ m\nabla n)\times \mathbf{D} = (m\ n)$ }
```

From this point on, whenever we refer to Euclid's algorithm, the intended reference is to this algorithm. The removal of the first step out of the loop body forces $n<m$ and $1<m$, but it allows us to include assertions after each assignment, making the inversion of the loop body a straightforward application of the rules mentioned above. (The new assertions after the assignments follow from the facts that premultiplying a matrix by $\mathbf{L}$ corresponds to adding the first row to the second, and premultiplying a matrix by $\mathbf{R}$ corresponds to adding the second row to the first). Because the assignments in the loop body are easily inverted, the inverse of Euclid's algorithm becomes:

```
{ $0<n<m$ }
$(x\ y) := (m\nabla n\ \ m\nabla n)$ ;
initialise $\mathbf{D}$ such that $(m\nabla n\ \ m\nabla n)\times\mathbf{D} = (m\ n)$ ;
{ Invariant: $(x\ y)\times\mathbf{D} = (m\ n)$ }
{ $a<c \vee c<a$ }
do $a<c \rightarrow (x\ y),\mathbf{D} := (x\ y)\times\mathbf{L}\ ,\ \mathbf{L}^{-1}\times\mathbf{D}$   { $y<x$ }
□ $c<a \rightarrow (x\ y),\mathbf{D} := (x\ y)\times\mathbf{R}\ ,\ \mathbf{R}^{-1}\times\mathbf{D}$   { $x<y$ }
od
{ $y<x \vee x<y$ }
{ $(x\ y) = (m-n\ \ n) \wedge \mathbf{D}=\mathbf{L}$ }
```

Comparing the two algorithms, we see that the assignments to $(x\ y)$ and to $(a\ c)$ are interchanged: in the original algorithm we have

```
  $y<x \rightarrow (x\ y) := (x-y\ \ y)$
□ $x<y \rightarrow (x\ y) := (x\ \ y-x)$ ,
```

and in the inverted algorithm we have

```
  $a<c \rightarrow (a\ c) := (a\ \ c-a)$
□ $c<a \rightarrow (a\ c) := (a-c\ \ c)$ .
```

(We leave the reader to check the matrix arithmetic.) In other words, the inverse of Euclid's algorithm is Euclid's algorithm itself, but on different variables: the inverted version computes the greatest common divisor using the variables a and c. This means that to make the sequences of successive values of the vectors $(x\ y)$ and $(a\ c)$ the reverse of each other, we only need to guarantee that the initial value of $(x\ y)$ in the non-inverted algorithm is the same as the initial value of

$(a\ c)$ in the inverted one. In other words, we need to guarantee that in Euclid's algorithm, the initial value of $(x\ y)$ is the same as the final value of $(a\ c)$.

The initial assignments of the inverted algorithm may seem strange at first sight, but the important fact to retain is that if we compose both algorithms, the program state remains unchanged. The inversion of the algorithm serves only as a formal proof that the process applied to $(x\ y)$ in one direction is the same as the one applied to $(a\ c)$ in the opposite direction. In the remainder of our investigation, we base our discussion on Euclid's algorithm, i.e., on the non-inverted version.

For more details on the inversion rules shown in this section, we recommend the expositions in [14, chapter 21] and [15, chapter 11]. As far as we know, the technique of program inversion first appeared in [16, pp. 351–354] and, since then, it has been mentioned and used in many places (see, for example, [17,18,19,20]).

4 Reversed Sequences of Vectors

Given the result of the previous section, saying that the sequences of vectors $(x\ y)$ and $(a\ c)$ are reversed is equivalent to saying that the initial value of $(a\ c)$ is equal to the final value of $(x\ y)$ and the initial value of $(x\ y)$ is equal to the final value of $(a\ c)$.

Looking at the algorithm, we see that the initial value of $(a\ c)$ is $(1\ 1)$ and the final value of $(x\ y)$ is $(m\nabla n\ \ m\nabla n)$. So, for the sequences to be reversed, $m\nabla n$ has to be 1, i.e., m and n have to be co-prime. We thus assume henceforth that $m\nabla n = 1$.

Also, the initial value of $(x\ y)$ is $(m-n\ \ n)$. So, because $m\nabla n = 1$, we have the following equality:

$$\begin{array}{ll} & \text{“The sequences are reversed”} \\ = & \\ & \text{“The final value of } (a\ c) \text{ is } (m-n\ \ n)\text{”} \quad . \end{array}$$

We can rewrite this equality in terms of matrix $\mathbf{D}$:

$$\begin{array}{ll} & \text{“The sequences are reversed”} \\ = & \\ & \text{“The final value of } \mathbf{D} \text{ is } \left(\begin{smallmatrix} m-n & b \\ n & d \end{smallmatrix}\right) \text{ for some } b \text{ and } d\text{”} \quad . \end{array}$$

Now, because $\mathbf{D}$ is the product of matrices whose determinant equals 1, its determinant also equals 1; this allows us to calculate b and d:

$$\begin{array}{ll} & det.\mathbf{D} = 1 \\ = & \{ \quad \mathbf{D} \text{ has the shape } \left(\begin{smallmatrix} m-n & b \\ n & d \end{smallmatrix}\right) \quad \} \\ & (m-n)\times d - n\times b \ = \ 1 \\ = & \{ \quad \text{arithmetic} \quad \} \end{array}$$

$$
\begin{array}{ll}
 & m\times d - n\times(d{+}b) \;=\; 1 \\
= & \{ \quad \text{we have assumed that } m\nabla n = 1\text{, so, on termination,} \\
 & \quad\;\; \text{the invariant states that } (1\;\,1)\times\mathbf{D} \;=\; (m\;\,n)\text{;} \\
 & \quad\;\; \text{this means that } n = b{+}d \quad \} \\
 & m\times d = n^2{+}1 \\
= & \{ \quad 0 < m \quad \} \\
 & d = \dfrac{n^2{+}1}{m} \quad .
\end{array}
$$

The value of b is simply $n{-}d$, since on termination we have $n = b{+}d$ (it follows from the invariant). Because $\mathbf{D}$ is a matrix of integer values, d has to be an integer, and so, a necessary condition is that $m\backslash(n^2{+}1)$, that is, $n^2 \cong -1 \pmod{m}$. (We write $m\backslash n$ to denote that m is a divisor of n. Although the notation $m|n$ is more common, we prefer to use an asymmetric symbol such as the backward slash to denote an asymmetric relation. Moreover, as the authors of [21, p. 102] point out, vertical bars are overused and $m\backslash n$ gives an impression that m is the denominator of an implied ratio. Also, $a{\cong}b \pmod{m}$ means that $m\backslash(a{-}b)$ and we read it as "a and b are congruent modulo m".) We can thus conclude that

$$n^2 \;\cong\; -1 \pmod{m} \Leftarrow \text{“The sequences are reversed”} \quad .$$

A question that now arises is whether $n^2 \cong -1 \pmod{m}$ is a sufficient condition for the sequences to be reversed. That is, can we prove

$$(2) \quad \text{“The final value of } \mathbf{D} \text{ is } \begin{pmatrix} m{-}n & n{-}{}^{(n^2+1)}\!/_{m} \\ n & {}^{(n^2+1)}\!/_{m} \end{pmatrix}\text{”} \;\Leftarrow\; n^2 \cong -1 \pmod{m}\,?$$

Using the assumption that $\mathbf{D} = \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$, we can simplify (2) to:

$$(3) \quad \text{“The final value of } c \text{ is } n\text{”} \quad \Leftarrow \quad n^2 \;\cong\; -1 \pmod{m} \quad ,$$

since c uniquely determines all the other entries (recall that $m = a{+}c$, $n = b{+}d$ and $det.\mathbf{D} = 1$). To prove (3), we first show that $n{\cong}c \pmod{m}$ follows from $n^2 \cong -1 \pmod{m}$ and then we use the range of n and c to conclude that $n = c$. The following lemma is used to prove that $n{\cong}c \pmod{m}$.

Lemma 1. *For all integers m, n, and c, the following holds:*

$$n \cong c \;(mod\, m) \;\Leftarrow\; n^2 \;\cong\; -1 \;\;(mod\, m) \;\wedge\; n{\times}c \;\cong\; -1 \;\;(mod\, m) \quad .$$

Proof. *Using the fact that, for all integers a, b, and c, the following law on congruences holds*

$$(4) \quad a{-}c \;\cong\; b{-}d \;\;(mod\, m) \Leftarrow a \cong b \;\;(mod\, m) \wedge c \cong d \;\;(mod\, m) \quad ,$$

we can prove the lemma as follows:

$$n{\cong}c \;\;(mod\, m)$$

$=\quad \{ \quad \textit{arithmetic} \quad \}$

$\quad n{-}c \cong 0 \pmod{m}$

$\Leftarrow\quad \{ \quad m\triangledown n = 1$ *and Euclid's lemma; see below for details* $\quad \}$

$\quad n{\times}(n{-}c) \cong 0 \pmod{m}$

$=\quad \{ \quad \textit{arithmetic} \quad \}$

$\quad n^2 - n{\times}c \cong 0 \pmod{m}$

$=\quad \{ \quad n^2 \cong -1 \pmod{m}$ *and* $n{\times}c \cong -1 \pmod{m}$ *and (4)* $\quad \}$

$\quad \mathsf{true}$.

In the second step we can safely assume that $m\triangledown n = 1$*, since it follows from the congruence* $n^2 \cong -1 \pmod{m}$ *. A short proof of this fact is:*

$\quad n^2 \cong -1 \pmod{m}$

$=\quad \{ \quad \textit{definition} \quad \}$

$\quad \langle \exists q{::}\ n^2{+}1 = q{\times}m \rangle$

$=\quad \{ \quad \textit{arithmetic} \quad \}$

$\quad \langle \exists q{::}\ 1 = q{\times}m - n{\times}n \rangle$

$\Rightarrow\quad \{ \quad (m\triangledown n)\backslash(q{\times}m - n{\times}n)$*, so* $(m\triangledown n)\backslash 1$*;*

division is anti-symmetric $\quad \}$

$\quad m\triangledown n = 1$.

Also, Euclid's lemma states that for all integers a, b, and c:

$$a\backslash c \quad \Leftarrow \quad a \backslash b{\times}c \ \wedge\ a\triangledown b = 1 \quad . \qquad \square$$

Now, if, on termination, we have that $n{\times}c \cong -1 \pmod{m}$, we can use lemma 1 to conclude that, on termination, we also have that $n{\cong}c \pmod{m}$ follows from $n^2 \cong -1 \pmod{m}$. Recall that an invariant of the algorithm is

$$(x\ y) \times \mathbf{D} \quad = \quad (m\ n) \quad = \quad (x{\times}a + y{\times}c \quad x{\times}b + y{\times}d) \quad .$$

Because the determinant of $\mathbf{D}$ equals 1, the inverse of $\mathbf{D}$ is $\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$, making the following property also invariant:

$$(5) \quad (x\ y) \quad = \quad (m\ n) \times \mathbf{D}^{-1} \quad = \quad (m{\times}d - n{\times}c \quad a{\times}n - b{\times}m) \quad .$$

It follows that on termination, when $(x\ y) = (1\ 1)$, we have that $n{\times}c \cong -1 \pmod{m}$, as the following calculation shows:

$\quad n{\times}c \cong -1 \pmod{m}$

$=\quad \{ \quad \text{definition} \quad \}$

$\quad m\backslash(n{\times}c + 1)$

$\Leftarrow$ { division properties }

$m{\times}d = n{\times}c+1$

$=$ { arithmetic }

$m{\times}d - n{\times}c = 1$

$=$ { invariant (5), $(x\ y) = (1\ 1)$ on termination }

true .

By lemma 1, we deduce that on termination $n{\cong}c \pmod{m}$ follows from $n^2 \cong -1 \pmod{m}$. Finally, because $0 < a$ and $m = a{+}c$ we have that $0 < c < m$; this allows us to conclude that $n = c$:

$n{\cong}c \pmod{m}$

$=$ { definition }

$m\backslash(n{-}c)$

$=$ { $0 < n < m$ and $0 < c < m$ imply that $-m < n{-}c < m$;
the only multiple of m in that range is 0 }

$n{-}c = 0$

$=$ { arithmetic }

$n = c$.

The conclusion is that $n^2 \cong -1 \pmod{m}$ is also a sufficient condition for the sequences to be reversed, leading to the equality:

"The sequences are reversed"

$=$

$n^2 \cong -1 \pmod{m}$.

To summarise, in the following algorithm

```
{ $0 < n < m$ }
$(x\ y)\,,\mathbf{D}\ :=\ (m{-}n\ \ n)\,,\mathbf{L}$ ;
{ **Invariant**:   $(m\ n) = (x\ y)\times\mathbf{D} = (x{\times}a + y{\times}c\quad x{\times}b + y{\times}d)$
  $\wedge$             $(m\ n)\times\mathbf{D}^{-1} = (x\ y) = (m{\times}d - n{\times}c\quad a{\times}n - b{\times}m)$ }
do $y < x\ \rightarrow\ (x\ y)\,,\mathbf{D}\ :=\ (x\ y)\times\mathbf{L}^{-1}\ ,\ \mathbf{L}{\times}\mathbf{D}$   { $a < c$ }
□  $x < y\ \rightarrow\ (x\ y)\,,\mathbf{D}\ :=\ (x\ y)\times\mathbf{R}^{-1}\ ,\ \mathbf{R}{\times}\mathbf{D}$   { $c < a$ }
od
{ $(x\ y) = (1\ 1)$   $\wedge$   $(m\ n) = (1\ 1)\times\mathbf{D}$ },
```

the sequences of vectors $(x\ y)$ and $(a\ c)$ are reverses of each other exactly when $n^2 \cong -1 \pmod{m}$.

5 Length of the Sequence of Vectors

We now have to prove that the final value of matrix $\mathbf{D}$ is decomposed into an odd-length product of the matrices $\mathbf{L}$ and $\mathbf{R}$. However, because $\mathbf{D}$ is initially $\mathbf{L}$ and because it is iteratively premultiplied, $\mathbf{D} = \mathbf{M}\times\mathbf{L}$ for some $\mathbf{M}$. So we can alternatively prove that $\mathbf{M}$ is decomposed into an even-length product of the matrices $\mathbf{L}$ and $\mathbf{R}$. Observing that

$$\mathbf{M} = \mathbf{D}\times\mathbf{L}^{-1} = \begin{pmatrix} m-(2\times n - {}^{(n^2+1)}\!/_{m}) & n-{}^{(n^2+1)}\!/_{m} \\ n-{}^{(n^2+1)}\!/_{m} & {}^{(n^2+1)}\!/_{m} \end{pmatrix} \quad ,$$

we see that $\mathbf{M}$ has the top-right and bottom-left corners equal, which means that $\mathbf{M} = \mathbf{M}^T$ ($\mathbf{M}$ equals the transpose of $\mathbf{M}$). We also know that $\mathbf{R} = \mathbf{L}^T$ and $\mathbf{L} = \mathbf{R}^T$.

There are also two functions from finite products of $\mathbf{L}$ and $\mathbf{R}$ to naturals, $\#\mathbf{L}$ and $\#\mathbf{R}$, that give, respectively, the number of $\mathbf{L}$s and the number of $\mathbf{R}$s in the decomposition of their argument[2]. Now, a fundamental property is that $\#\mathbf{L}.\mathbf{M} = \#\mathbf{R}.\mathbf{M}^T$, whenever $\mathbf{M}$ is a product of $\mathbf{L}$s and $\mathbf{R}$s. This fundamental property means that the number of $\mathbf{L}$s in the decomposition of $\mathbf{M}$ equals the numbers of $\mathbf{R}$s in the decomposition of $\mathbf{M}^T$, which is easy to see because $\mathbf{R} = \mathbf{L}^T$ and $\mathbf{L} = \mathbf{R}^T$. Using these observations, a simple calculation showing that the length of $\mathbf{M}$ is an even number is:

$$\begin{array}{ll} & \mathsf{length}.\mathbf{M} \\ = & \quad \{ \quad \mathbf{M} \text{ is a product of } \mathbf{L}\text{s and } \mathbf{R}\text{s} \quad \} \\ & \#\mathbf{L}.\mathbf{M} + \#\mathbf{R}.\mathbf{M} \\ = & \quad \{ \quad \#\mathbf{L}.\mathbf{M} = \#\mathbf{R}.\mathbf{M}^T \quad \} \\ & \#\mathbf{R}.\mathbf{M}^T + \#\mathbf{R}.\mathbf{M} \\ = & \quad \{ \quad \mathbf{M}^T = \mathbf{M} \quad \} \\ & 2 \times \#\mathbf{R}.\mathbf{M} \quad . \end{array}$$

Hence, the length of $\mathbf{M}$ is an even number. Subsequently, the length of the final value of $\mathbf{D}$ is odd.

6 Sum of Two Positive Squares

In the above sections we have proved the following theorem:

Theorem 1. *A number m at least 2 can be written as the sum of two positive squares if there is a number n such that $0 < n < m$ and $n^2 \cong -1$ (mod m).* □

[2] Note that, given that we can easily provide algorithms that compute them, functions length, $\#\mathbf{L}$, and $\#\mathbf{R}$ are well-defined. As proved in [13] and [12], there is a bijection between finite products of matrices $\mathbf{L}$ and $\mathbf{R}$, and binary strings made of the symbols L and R; defining these functions in the realm of strings is easy.

The argument we provide is constructive because we show how to use Euclid's algorithm to represent a number as the sum of two positive squares. Indeed we can extend Euclid's algorithm so that it expresses a given number m as the sum of two positive squares:

```
{ $1<m \wedge \langle \exists n : 0<n<m : n^2 \cong -1 \pmod{m} \rangle$ }
  • Find a number $n$ such that $0<n<m$ and $n^2 \cong -1 \pmod{m}$;
{ $0<n<m \wedge n^2 \cong -1 \pmod{m}$ }
$(x\ y), \mathbf{D} := (m-n\ \ n), \mathbf{L}$ ;
{ Invariant: $(x\ y) \times \mathbf{D} = (m\ n) = (x \times a + y \times c\ \ x \times b + y \times d)$ }
do $(x\ y) \neq (a\ c)$ →
        $y<x$ → $(x\ y), \mathbf{D} := (x\ y) \times \mathbf{L}^{-1}$ , $\mathbf{L} \times \mathbf{D}$
      □ $x<y$ → $(x\ y), \mathbf{D} := (x\ y) \times \mathbf{R}^{-1}$ , $\mathbf{R} \times \mathbf{D}$
od
{ $(x\ y) = (a\ c) \wedge m = x^2+y^2 = a^2+c^2$ }
```

Theorem 1 is more general than Girard's result: while Girard's theorem is only on odd prime numbers, theorem 1 concerns all positive integers at least 2. As an example, we can say that the number 10 is expressible as the sum of two positive squares, since $3^2 \cong -1 \pmod{10}$ (and, in fact, we have that $10 = 3^2+1^2$). Moreover, given the following lemma (see [9, p. 17, Lemma 1]), Girard's result is an immediate corollary of theorem 1.

Lemma 2. *For primes $p = 4k+1$ the equation $s^2 \cong -1 \pmod{p}$ has two solutions $s \in \{1 .. p-1\}$, for $p=2$ there is only one such solution, while for primes of the form $p = 4k+3$ there is no solution.* □

Although we believe that theorem 1 may be known by some number-theorists, we have not found it in the literature.

Please note that developing an algorithm to find a number n such that $0<n<m$ and $n^2 \cong -1 \pmod{m}$ is beyond the scope of this paper. For more details on this topic, we recommend [11] and [22], where the authors discuss different algorithms that can be used to find such a number n.

Finally, the algorithm shown above can be generalised. In a recent private communication, Wagon told us that the method of using Euclid's algorithm to write a number as a sum of two squares (or, more generally, as $a^2 + g \times c^2$) is known as the Smith-Cornacchia algorithm (he referred us to [22] and [23]). Also, in [24], Hardy, Muskat, and Williams show a more general algorithm for solving $m = f \times a^2 + g \times c^2$ in coprime integers a and c. The algorithm presented in this paper treats the case $f = g = 1$. At the moment, we do not know how to adapt it to solve the more general problem. Recall that we have started our argument by observing that if, at any point in the execution of the algorithm, $(x\ y)$ equals $(a\ c)$, it follows from the invariant

$$(m\ n) \quad = \quad (x\ y) \times \mathbf{D} \quad = \quad (x{\times}a+y{\times}c \quad x{\times}b+y{\times}d)$$

that m can be written as a sum of two positive squares. To solve the general problem, we have to investigate when it is possible to have, at any point in the execution of the algorithm, $a\backslash x$ and $c\backslash y$. If this happens, that is, if there are two integers f and g such that $x = f{\times}a$ and $y = g{\times}c$, it follows from the invariant that $m = f{\times}a^2 + g{\times}c^2$:

$$(m\ n) \quad = \quad (f{\times}a \quad g{\times}c) \times \mathbf{D} \quad = \quad (f{\times}a^2 + g{\times}c^2 \quad f{\times}a{\times}b + g{\times}c{\times}d) \quad .$$

7 Discussion

This paper shows a new and constructive proof of the two-squares theorem based on a somewhat unusual, but very effective, way of rewriting the so-called extended Euclid's algorithm. As mentioned in the introduction, the use of Euclid's algorithm to prove the theorem is not new: Brillhart [6] and Wagon [11] have used it to *verify* the theorem. Effectively, given the close relationship between Euclid's algorithm and continued fractions, we can say that Serret [5] and Hermite [4] were the first to provide the germ of the essential idea presented here (in fact, Brillhart's note is described as an improvement on Hermite's method: in using Euclid's algorithm, Brillhart avoids the calculation of the convergents arising in the continued fractions).

The novel contribution of this paper is the use of the algorithm to *investigate* which numbers can be written as the sum of two positive squares. The precise formulation of the problem as an algorithmic problem is the key, since it allows us to use algorithmic techniques and to avoid guessing. The notion of invariance, in particular, plays a central role in our development: it is used initially to observe that Euclid's algorithm can actually be used to represent a given number as a sum of two positive squares, and then it is used throughout the argument to prove relevant properties. Also, section 3 is an example of how the use of program inversion can make our arguments more precise.

We conclude by mentioning that this paper is part of a larger endeavour which aims at reinvigorating mathematics education by exploiting mathematics' algorithmic nature [13,12,25]. In our view, the combination of practicality with mathematical elegance that arises from an adequate focus on the algorithmic content of mathematics can enrich and improve, not only mathematics education, but also the process of constructing computer programs. Moreover, the emphasis on investigation and construction rather than verification brings tremendous benefits. As Leibniz once put it:

> *Nothing is more important than to see the sources of invention which are, in my opinion, more interesting than the inventions themselves.*

Acknowledgements. I would like to thank Roland Backhouse for helping me with the initial analysis of the problem, for suggesting the use of program inversion techniques, and for his comments on earlier drafts of this paper. Also, thanks to Alexandra Mendes for helping me simplifying section 5 and to Wei Chen and Shin-Cheng Mu for reading and checking earlier drafts of the paper. I would also like to thank Stan Wagon for sending me references on the Smith-Cornacchia algorithm and the anonymous referees, whose comments and corrections have led to significant improvements.

This work was developed in the context of the MATHIS project, which is supported by Fundação para a Ciência e a Tecnologia (Portugal) under contract PTDC/EIA/73252/2006.

References

1. Dickson, L.E.: History of the Theory of Numbers: Diophantine Analysis: Diophantine Analysis, vol. 2. AMS/Chelsea Publication, American Mathematical Society (August 1999)
2. Bell, E.T.: Men of Mathematics - The Lives and Achievements of the Great Mathematicians from Zeno to Poincaré. Touchstone (July 2008)
3. Gauss, C.F.: Disquisitiones Arithmeticae. G. Fleischer, Leipzig (1801) English translation by Clarke, A. A. Springer, Heidelberg (1986)
4. Hermite: Note au sujet de l'article précédent. Journal de Mathématiques Pures et Appliquées 13, 15 (1848)
5. Serret, J.A.: Sur un théorème relatif aux nombres entiers. Journal de Mathématiques Pures et Appliquées 13, 12–14 (1848)
6. Brillhart, J.: Note on representing a prime as a sum of two squares. Mathematics of Computation 26(120), 1011–1013 (1972)
7. Clarke, F.W., Everitt, W.N., Littlejohn, L.L., Vorster, S.J.R.: H. J. S. Smith and the Fermat two squares theorem. The American Mathematical Monthly 106(7), 652–665 (1999)
8. Zagier, D.: A one-sentence proof that every prime $p \equiv 1 (\mathrm{mod}\ 4)$ is a sum of two squares. The American Mathematical Monthly 97(2), 144 (1990)
9. Aigner, M., Ziegler, G.: Proofs From The Book, 3rd edn. Springer, Heidelberg (2004)
10. Dijkstra, E.W.: A derivation of a proof by D. Zagier. Circulated privately (August 1993), `http://www.cs.utexas.edu/users/EWD/ewd11xx/EWD1154.PDF`
11. Wagon, S.: Editor's corner: The Euclidean algorithm strikes again. The American Mathematical Monthly 97(2), 125–129 (1990)
12. Backhouse, R., Ferreira, J.F.: On Euclid's algorithm and elementary number theory. To appear in the journal Science of Computer Programming (2010), `http://joaoff.com/publications/2009/euclid-alg/`
13. Backhouse, R., Ferreira, J.F.: Recounting the rationals: Twice! In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 79–91. Springer, Heidelberg (2008)
14. Gries, D.: The Science of Programming. Springer, Heidelberg (1981)
15. van de Snepscheut, J.L.: What Computing Is All About. Springer, New York (1993)
16. Dijkstra, E.W.: Program inversion. In: Selected Writings on Computing: A Personal Perspective, pp. 351–354. Springer, Heidelberg (1982)

17. van de Snepscheut, J.L.A.: Inversion of a recursive tree traversal. Inf. Process. Lett. 39(5), 265–267 (1991)
18. Chen, W.: A formal approach to program inversion. In: CSC '90: Proceedings of the 1990 ACM annual conference on Cooperation, pp. 398–403. ACM Press, New York (1990)
19. von Wright, J.: Program inversion in the refinement calculus. Inf. Process. Lett. 37(2), 95–100 (1991)
20. Mu, S.C., Bird, R.: Rebuilding a tree from its traversals: A case study of program inversion. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 265–282. Springer, Heidelberg (2003)
21. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics: a Foundation for Computer Science, 2nd edn. Addison-Wesley Publishing Company, Reading (1994)
22. Buhler, J., Wagon, S.: Basic algorithms in number theory. In: Buhler, J.P., Stevenhagen, P. (eds.) Algorithmic Number Theory. Lattices, Number Fields, Curves and Cryptography, pp. 25–68. Cambridge University Press, Cambridge (2008)
23. Cornacchia, G.: Su di un metodo per la risoluzione in numeri interi dell'equazione $\sum_{h=0}^{n} C_h x^{n-h} y^h = P$. Giornale di Matematiche di Battaglini 46, 33–90 (1908)
24. Hardy, K., Muskat, J.B., Williams, K.S.: A deterministic algorithm for solving $n = fu^2 + gv^2$ in coprime integers u and v. Mathematics of Computation 55(191), 327–343 (1990)
25. Ferreira, J.F., Mendes, A., Backhouse, R., Barbosa, L.S.: Which mathematics for the information society? In: Oliveira, J. (ed.) TFM 2009. LNCS, vol. 5846, pp. 39–56. Springer, Heidelberg (2009)

Partial, Total and General Correctness

Walter Guttmann

Institut für Programmiermethodik und Compilerbau
Universität Ulm, 89069 Ulm, Germany
walter.guttmann@uni-ulm.de

Abstract. We identify weak semirings, which drop the right annihilation axiom $a0 = 0$, as a common foundation for partial, total and general correctness. It is known how to extend weak semirings by operations for finite and infinite iteration and domain. We use the resulting weak omega algebras with domain to define a semantics of while-programs which is valid in all three correctness approaches. The unified, algebraic semantics yields program transformations at once for partial, total and general correctness. We thus give a proof of the normal form theorem for while-programs, which is a new result for general correctness and extends to programs with non-deterministic choice.

By adding specific axioms to the common ones, we obtain partial, total or general correctness as a specialisation. We continue our previous investigation of axioms for general correctness. In particular, we show that a subset of these axioms is sufficient to derive a useful theory, which includes the Egli-Milner order, full recursion, correctness statements and a correctness calculus. We also show that this subset is necessary.

1 Introduction

Partial, total and general correctness are three approaches to the semantics of programs distinguished by [25]. Leaving the characterisation to Section 3, at this point we just give a sample of the available literature: For example, partial correctness is supported by Hoare logic [21], weakest liberal preconditions [12] and Kleene algebra with tests [28]; total correctness is supported by weakest preconditions [12], the Unifying Theories of Programming [22], demonic refinement algebra [37] and demonic algebra [5]; general correctness is supported by the works [2,4,25,3,34,14,32,17].

Despite various links between partial, total and general correctness, the approaches differ essentially by giving distinct semantics to programs and distinct laws about programs. As a consequence, a particular program transformation may be applicable in one approach, but not in the others. Even if a transformation rule is valid in all three approaches, this has to be proved separately for each of them.

Here comes into play the algebraic approach of identifying basic laws that programs satisfy, collecting them as axioms of algebraic structures, and taking programs as elements of these algebras. The first benefit is that reasoning carried out at the algebraic level is valid in any concrete model that satisfies the axioms; typically there are several in the literature. The second benefit is that reasoning can be supported by automated theorem provers and counterexample generators such as Prover9/Mace4, since the axioms are usually first order conditional equations.

It turns out that partial, total and general correctness have a fairly large set of common axioms, certainly large enough to prove complex program transformations. Such a transformation will then be valid in all three approaches, since each of them satisfies the common axioms, and possibly further ones. So the third benefit is that reasoning has to be performed only once, provided the transformation is stated so that it meaningfully talks about programs in each particular approach. At least for while-programs this can be achieved, as this paper shows.

In Section 2, we present the common set of axioms, namely weak omega algebra with domain.

In Section 3, we discuss which further axioms have to be added to obtain partial, total or general correctness. We particularly focus on general correctness, since it is developed less well than the others. The main contribution here is a reduction of our previous axioms [17] to the minimum necessary to represent the Egli-Milner order, which is used for the semantics of loops and recursion.

In Section 4, we show that this minimal axiomatisation is still sufficient to develop a useful theory of general correctness, which includes full recursion, correctness statements and a correctness calculus. Some results require new proofs due to the reduced set of axioms. Although carried out in a general correctness setting, the development provides a semantics of while-loops that also suits partial and total correctness.

This is established in Section 5, where we return to the common axioms. We use the unified semantics to state and prove transformations that bring while-programs to a normal form. This result is known for partial correctness [27] and has recently been extended to total correctness [35], but both proofs use a program semantics and axioms specific to the respective correctness approach. Our proof avoids this and thus gives a transformation valid in all three approaches; in particular, this is new for general correctness. Additionally, we extend the result to programs with non-deterministic choice.

2 Common Axioms

In this section, we present the axioms which are common to partial, total and general correctness. They are grouped in algebraic structures which are fundamental to many branches of computer science, not only program semantics.

Former investigations have shown that these structures underly various correctness approaches. For a detailed discussion and concrete models we refer to [28,31,9] for partial correctness, to [37,19,30,5,20] for total correctness, and to [32,17] for general correctness.

2.1 Choice and Composition

A *weak semiring* is a structure $(S, +, 0, \cdot, 1)$ such that $(S, +, 0)$ is a commutative monoid, $(S, \cdot, 1)$ is a monoid, the operation $\cdot$ distributes over $+$ in both arguments and 0 is a left annihilator, that is, $0 \cdot a = 0$. We assume $0 \neq 1$, otherwise S would be trivial. A *semiring* is a weak semiring in which 0 is also a right annihilator, that is, $a \cdot 0 = 0$. We frequently abbreviate $a \cdot b$ with ab.

A weak semiring is *idempotent* if $+$ is, that is, if $a + a = a$. In an idempotent weak semiring the relation $a \leq b \Leftrightarrow_{\text{def}} a + b = b$ is a partial order, called the *natural order* on S [9]. The operations $+$ and $\cdot$ are isotone and $a + b$ is the join of a and b with respect to the natural order. An idempotent weak semiring is *bounded* if it has a greatest element $\top$, such that $a \leq \top$.

Our notation reflects the intended relational model, where elements of S represent the state transition relation or input/output behaviour of programs. In this model, the operations $+$ and $\cdot$ are non-deterministic choice and sequential composition, respectively. Moreover, 0, 1 and $\top$ are the empty, identity and universal relations, respectively, and $\leq$ is the subset order, so that $0 \leq 1 \leq \top$ holds, for example. Hence $a \leq b$ states that a is a refinement of b. To avoid confusion, it should be kept in mind that other approaches in the literature use different conventions, such as the reverse order in [37].

2.2 Finite and Infinite Iteration

A *(weak) Kleene algebra* [26,29] is a structure $(S, ^*)$ such that S is an idempotent (weak) semiring and the operation star * satisfies the unfold and induction laws

$$\begin{array}{ll} 1 + a \cdot a^* \leq a^* & \qquad b + a \cdot c \leq c \Rightarrow a^* \cdot b \leq c \\ 1 + a^* \cdot a \leq a^* & \qquad b + c \cdot a \leq c \Rightarrow b \cdot a^* \leq c \end{array}$$

for $a, b, c \in S$. As a consequence, a^*b is the least fixpoint of $\lambda x.ax + b$ and * is isotone with respect to $\leq$. We also have the sliding law $a(ba)^* = (ab)^*a$ and the decomposition laws $(a + b)^* = a^*(ba^*)^* = (a^*b)^*a^*$.

A *(weak) omega algebra* [6,29] is a structure $(S, ^\omega)$ such that S is a (weak) Kleene algebra and the operation omega $^\omega$ satisfies the unfold and co-induction laws

$$a^\omega = a \cdot a^\omega \qquad\qquad c \leq a \cdot c + b \Rightarrow c \leq a^\omega + a^* \cdot b$$

for $a, b, c \in S$. As a consequence, $a^\omega + a^*b$ is the greatest fixpoint of $\lambda x.ax + b$ and $^\omega$ is isotone with respect to $\leq$. Any weak omega algebra is bounded since $1^\omega = \top$. Moreover, $a^\omega \top = a^\omega = a^*a^\omega$ and $a^*0 \leq a^*a^\omega 0 = a^\omega 0$ and $c \leq a \cdot c \Rightarrow c \leq a^\omega$, whence a^ω is the greatest fixpoint of $\lambda x.ax$.

The Kleene star and omega operations represent finite and infinite iteration, respectively. They are used to conveniently express the semantics of loops.

2.3 Tests and Domain

A *test semiring* [29] is an idempotent weak semiring $(S, +, 0, \cdot, 1)$ with a distinguished set of elements $\text{test}(S) \subseteq S$ called *tests* and a *negation* operation $\neg$ on tests such that $(\text{test}(S), +, 0, \cdot, 1, \neg)$ is a Boolean algebra. We also write $\overline{p}$ for $\neg p$.

Tests allow us to represent the conditional statement by $\mathsf{if}\ p\ \mathsf{then}\ a\ \mathsf{else}\ b =_{\mathrm{def}} pa+\overline{p}b$. Note that negation applies only to elements of $\mathrm{test}(S)$, not to all elements of S. We use the letters p, q, r to denote tests in this paper.

A *domain semiring* [9] is a structure $(S, \ulcorner)$ such that S is a test semiring and the domain operation $\ulcorner : S \to \mathrm{test}(S)$ satisfies the axioms

$$a \le \ulcorner a \cdot a \qquad \ulcorner(p \cdot a) \le p \qquad \ulcorner(a \cdot \ulcorner b) \le \ulcorner(a \cdot b)$$

for $a, b \in S$ and $p \in \mathrm{test}(S)$. The alternative axiomatisation of [11,10] can also be used. Useful properties for $a, b \in S$ and $p \in \mathrm{test}(S)$ are

$$\begin{array}{llll}
\ulcorner a \le p \Leftrightarrow a \le pa & a \le b \Rightarrow \ulcorner a \le \ulcorner b & a = \ulcorner a a & \ulcorner p = p \\
a \le 0 \Leftrightarrow \ulcorner a \le 0 & \ulcorner(a+b) = \ulcorner a + \ulcorner b & \ulcorner(pa) = p\ulcorner a & \ulcorner(a \cdot \ulcorner b) = \ulcorner(a \cdot b)
\end{array}$$

If S is bounded, a characterisation of domain is $\ulcorner a \le p \Leftrightarrow a \le p\top$ [1]. We also use its instance $a \le \ulcorner a\top$.

An element $a \in S$ is *total* if $\ulcorner a = 1$. In a bounded setting without domain, we can use $a\top = \top$ instead. In a bounded domain semiring, $a\top = \top$ implies $\ulcorner a = 1$, but the converse does not hold in general.

The domain of the program a represents the initial states from which a transition under a is possible. In this paper, we use it for the axiomatic treatment of general correctness in Sections 3.3 and 4; see [31,5,20] for applications of the domain operation in partial and total correctness.

3 Axioms for Partial, Total and General Correctness

In this section, we characterise partial, total and general correctness, and show how to obtain each by extending the axioms introduced in Section 2. We particularly focus on general correctness and the axioms necessary to represent the Egli-Milner order. The general setting is a bounded domain semiring; recursion is treated in Section 4.

To motivate the axioms, let loop denote the endless loop or the program which never terminates [34]. In a conventional setting, we expect that the endless loop satisfies $\mathsf{loop} \cdot a = \mathsf{loop}$ for all programs a, and $a \cdot \mathsf{loop} = \mathsf{loop}$ if a is total. Based on these requirements, loop is formally characterised below. For a theory without the law $\mathsf{loop} \cdot a = \mathsf{loop}$, describing lazy execution of programs, we refer to [18].

3.1 Partial Correctness

Because partial correctness ignores termination issues, there is no need to represent that the execution of a program might not terminate. Nevertheless a semantics must be assigned to loop, and this is done by interpreting the absence of terminating executions as non-termination. As a consequence, if a program has both a terminating and a non-terminating execution, the terminating one is chosen, simply because it is present. Thus partial correctness can be informally characterised by ‘termination absorbs non-termination’.

In the algebraic setting this translates to the law $\mathsf{loop}+a = a$ for each program a, hence $+$ models angelic non-determinism according to [36]. By definition of the natural order, we thus obtain $\mathsf{loop} \leq a$ for each a, and therefore $\mathsf{loop} = 0$. It follows that loop is the least solution of the equation $a = a$, or the least fixpoint of the corresponding identity function. This motivates why recursion is modelled by least fixpoints with respect to the natural order.

Another consequence of $\mathsf{loop} = 0$ is the law $a \cdot 0 = 0$ for total a, and hence even for all a. This law cannot be proved from the axioms introduced in Section 2, but has to be added as the characteristic axiom of partial correctness. Hence the underlying structures are semirings, Kleene algebras and omega algebras, without the qualifier 'weak'. Observe that the new axiom can equivalently be expressed as $\top \cdot 0 = 0$.

Theories of partial correctness include Hoare logic [21], weakest liberal preconditions [12], Kleene algebra with tests [28] and with domain [9].

3.2 Total Correctness

On the other hand, total correctness takes the issue of termination very seriously; in concrete models it is typically represented by adding a special value, predicate or variable. To guarantee that the execution of a program terminates irrespective of non-deterministic choices, only the complete absence of non-terminating executions is interpreted as termination. As a consequence, if a program has both a terminating and a non-terminating execution, the non-terminating one is chosen. Thus total correctness can be informally characterised by 'non-termination absorbs termination'.

In the algebraic setting this translates to the law $\mathsf{loop} + a = \mathsf{loop}$ for each program a, hence $+$ models demonic non-determinism according to [36]. By definition of the natural order, we thus obtain $a \leq \mathsf{loop}$ for each a, and therefore $\mathsf{loop} = \top$. Thus recursion in total correctness is modelled by greatest fixpoints with respect to the natural order.

Another consequence of $\mathsf{loop} = \top$ is the law $\top \cdot a = \top$ for all a, which also cannot be proved from the axioms in Section 2. It can equivalently be expressed as $\top \cdot 0 = \top$, and has to be added as the characteristic axiom of total correctness.

Theories of total correctness include weakest preconditions [12], the Unifying Theories of Programming [22], demonic refinement algebra [37] and demonic algebra [5].

3.3 General Correctness

In general correctness, terminating and non-terminating executions are treated independently. It therefore offers a finer distinction than either partial or total correctness [25,15]. For example, the program $1 + \mathsf{loop}$, which equals 1 in partial correctness and loop in total correctness, is neither 1 nor loop in general correctness; it has both a terminating and a non-terminating execution. The choice operation $+$ models erratic non-determinism according to [36].

However, a price must be paid for this additional precision: the natural order has to be replaced by the more complex Egli-Milner order to model recursion. In fact, the semantics of recursion is given by least fixpoints with respect to this order. The natural order is still used for refinement, as it is in partial and total correctness.

Theories of general correctness include those of [2,4,25,3], the commands of [34,32] and the prescriptions of [14].

As observed above, $\mathsf{loop}+a \notin \{a, \mathsf{loop}\}$ in general, and therefore $\mathsf{loop} \notin \{0, \top\}$. To obtain a representation for loop, we first recall that $\top \cdot 0 = 0 = \mathsf{loop}$ in partial correctness and $\top \cdot 0 = \top = \mathsf{loop}$ in total correctness. It is therefore manifest to try $\mathsf{loop} = \top \cdot 0$, and this attempt is confirmed by verifying it in concrete models of general correctness such as those of [34,14,32].

The element $\top \cdot 0$ occurs in several contexts, such as infinite computations and temporal logic [13,33,29]. For general correctness, it is important as the intended least element of the Egli-Milner order, hence we call this element $\mathsf{L} =_{\mathrm{def}} \top \cdot 0$. By itself, this definition does not impose any restrictions; this has to be done by adding further axioms to obtain general correctness.

Such axioms, about L and another element H corresponding to the command havoc of [34], are studied in our previous work [17]. There we propose the following four axioms:

$$\begin{array}{llll} (\mathsf{L1}) & \ulcorner a\mathsf{L} \le a\mathsf{L} & \qquad (\mathsf{H1}) & a \le b + \mathsf{L} \wedge a \le b + \mathsf{H} \Rightarrow a \le b \\ (\mathsf{L2}) & 1 \le \ulcorner \mathsf{L} & \qquad (\mathsf{H2}) & a \le a0 + \mathsf{H} \end{array}$$

Based on these axioms, a theory of general correctness is derived including correctness statements, a correctness calculus, specification constructs, a loop refinement rule, and a representation of least fixpoints with respect to the Egli-Milner order in terms of the natural order. By domain axioms, ($\mathsf{L1}$) is equivalent to $\ulcorner a\mathsf{L} = a\mathsf{L}$ and ($\mathsf{L2}$) is equivalent to $1 = \ulcorner \mathsf{L}$.

Although the above axioms are independent, our previous work does not answer which of them, or yet others, are essential to build a meaningful theory of general correctness. In the remainder of this section, we show that under reasonable assumptions we cannot do away with ($\mathsf{L1}$) and ($\mathsf{L2}$). The topic of Section 4 is to show that these two axioms indeed suffice to derive a useful theory.

We start by defining the relation $\sqsubseteq$ that is planned to become the Egli-Milner order, whence we call it the *Egli-Milner relation*:

$$a \sqsubseteq b \Leftrightarrow_{\mathrm{def}} a \le b + a0 \wedge b \le a + \ulcorner(a0)\top \ .$$

This definition is justified by verifying that it instantiates to the Egli-Milner order given in [34,32,16]; the calculation is similar to the one in [17]. The second condition $b \le a + \ulcorner(a0)\top$ is equivalent to $\neg\ulcorner(a0)b \le a$.

Remark. Intuitively, forming the product $a0$ cuts away all terminating executions of a, hence the term $\ulcorner(a0)$ represents those states from which non-terminating executions of a exist. Its complement $\neg\ulcorner(a0)$ represents the initial states from which termination is guaranteed. By restricting both inequalities of the Egli-Milner relation to these states, we obtain $\neg\ulcorner(a0)a = \neg\ulcorner(a0)b$, which means that a and b must have the same executions in states from which a certainly terminates. The intuition for the remaining states is that b may or may not terminate, but it must have at least the terminating executions of a.

The main result of this section precisely characterises the Egli-Milner order. To state it, we consider the following consequence of ($\mathsf{L1}$):

$$(\mathsf{L3}) \quad \ulcorner(a0)\mathsf{L} \le a$$

It is equivalent to $\ulcorner(a0)\mathsf{L} = a0$ and [17, Lemma 2] shows $(\mathsf{L1})\wedge(\mathsf{L2}) \Leftrightarrow (\mathsf{L3})\wedge(\mathsf{L2})$.

Theorem 1. *The relation $\sqsubseteq$ is a partial order if and only if* ($\mathsf{L3}$) *holds. It has the least element* L *if and only if* ($\mathsf{L2}$) *holds. It has no greatest element. The operations $+$ and $\cdot$ are isotone with respect to $\sqsubseteq$.*

Proof. Reflexivity follows immediately from the definition of $\sqsubseteq$.

We show that $\sqsubseteq$ is antisymmetric if and only if ($\mathsf{L3}$) holds. Assume ($\mathsf{L3}$) and $a \sqsubseteq b$ and $b \sqsubseteq a$. Then $b0 \le (a + \ulcorner(a0)\top)0 = a0 + \ulcorner(a0)\mathsf{L} = a0 + a0 = a0$, and symmetrically $a0 \le b0$. Thus $a \le b + a0 \le b + b0 = b$ and symmetrically $b \le a$, hence $a = b$.

For the converse implication assume that $\sqsubseteq$ is antisymmetric. Then ($\mathsf{L3}$) holds if we can show $\ulcorner(a0)\mathsf{L} \sqsubseteq a0$ and $a0 \sqsubseteq \ulcorner(a0)\mathsf{L}$. The latter follows immediately from the definition of $\sqsubseteq$ and the former follows by $\ulcorner(a0)\mathsf{L} = \ulcorner(a0)\mathsf{L}0 \le a0 + \ulcorner(a0)\mathsf{L}0$ and

$$a0 \le \ulcorner(a0)a0 \le \ulcorner(a0)\top 0 = \ulcorner(a0)\mathsf{L} \le \ulcorner(a0)\mathsf{L} + \ulcorner(\ulcorner(a0)\mathsf{L}0)\top$$

using a domain axiom in the first step.

We next show that ($\mathsf{L3}$) implies that $\sqsubseteq$ is transitive, which completes the proof of the first claim. Assume $a \sqsubseteq b$ and $b \sqsubseteq c$. Then $b0 \le a0$ as shown above, hence $a \le b + a0 \le c + b0 + a0 = c + a0$ and $c \le b + \ulcorner(b0)\top \le a + \ulcorner(a0)\top$. But this implies $a \sqsubseteq c$.

For the second claim, consider $\mathsf{L} \sqsubseteq b$. Since $\mathsf{L} \le b + \mathsf{L}0 = b + \mathsf{L}$ always holds, $\mathsf{L} \sqsubseteq b$ is equivalent to $b \le \mathsf{L} + \ulcorner(\mathsf{L}0)\top = \ulcorner\mathsf{L}\mathsf{L} + \ulcorner\mathsf{L}\top = \ulcorner\mathsf{L}\top$. This holds for all b if and only if $\top \le \ulcorner\mathsf{L}\top$. By characterisation of domain, this is equivalent to $1 \le \ulcorner\mathsf{L}$.

For the third claim, assume $0 \sqsubseteq b$ and $1 \sqsubseteq b$. The former implies that $b \le 0 + \ulcorner(0 \cdot 0)\top = \ulcorner 0\top = 0\top = 0$. Thus the latter implies $1 \le b + 1 \cdot 0 = 0 + 0 = 0$, a contradiction.

To see that $+$ and $\cdot$ are isotone, assume $a \sqsubseteq b$. Then $a + c \sqsubseteq b + c$, since $a+c \le b+a0+c = b+c+(a+c)0$ and $b+c \le a+\ulcorner(a0)\top+c \le a+c+\ulcorner((a+c)0)\top$. Moreover $ca \sqsubseteq cb$, since $ca \le c(b + a0) = cb + ca0$ and

$$cb \le c(a + \ulcorner(a0)\top) = ca + \ulcorner(c\ulcorner(a0))c\ulcorner(a0)\top \le ca + \ulcorner(ca0)\top\ .$$

Finally $ac \sqsubseteq bc$, since $ac \le (b + a0)c = bc + a0 \le bc + ac0$ holds as well as $bc \le (a + \ulcorner(a0)\top)c \le ac + \ulcorner(ac0)\top$. □

This means that axioms (L1) and (L2) are equivalent to the requirement that the Egli-Milner relation is a partial order with least element L. Let us discuss why this is a reasonable assumption. First, it holds in concrete models such as that of [34]. Second, if we do not assume a partial order, it is difficult to define least fixpoints, which are necessary for loops and recursion. Third, a characteristic of general correctness is that the endless loop L is the least fixpoint of the identity function with respect to the Egli-Milner order, hence its least element.

Remark. This leaves open the question, whether the Egli-Milner relation can be defined in another way, which requires less or different axioms. Besides the alternative definition of [17], which is based on (H1) and (H2), we have investigated the following candidates:

- $a \sqsubseteq_1 b \Leftrightarrow_{\text{def}} b \leq a + \ulcorner(a0)\top \wedge a \leq b + \mathsf{L}$
- $a \sqsubseteq_2 b \Leftrightarrow_{\text{def}} b \leq a + \ulcorner(a0)\top \wedge a \leq b + \ulcorner(a0)\mathsf{L}$
- $a \sqsubseteq_3 b \Leftrightarrow_{\text{def}} b \leq a + \ulcorner(a0)\top \wedge a \leq b + \neg\ulcorner(b0)\mathsf{L}$
- $a \sqsubseteq_4 b \Leftrightarrow_{\text{def}} b \leq a + \ulcorner(a0)\top \wedge a \leq b + \neg\ulcorner(b0)\ulcorner(a0)\mathsf{L}$

The relations $\sqsubseteq_{1,2}$ are preorders; they are orders if and only if (L3) holds; they have least element L if and only if (L2) holds. The relations $\sqsubseteq_{3,4}$ are orders; they have least element L if and only if both (L2) and (L3) hold. Assuming (L3), the orders $\sqsubseteq$ and $\sqsubseteq_2$ and $\sqsubseteq_4$ coincide, as do $\sqsubseteq_1$ and $\sqsubseteq_3$.

In all of the above cases, (L2) and (L3) are necessary to obtain a partial order with least element L. We have not found a way to replace the condition $b \leq a + \ulcorner(a0)\top$ to obtain a suitable, yet different relation.

An improvement over previous treatments such as [34,32] is that we abstract from the concrete definition of commands as pairs of termination and transition information.

An improvement over our previous work [17] is that we no longer require the axioms (H1) and (H2), and hence it is not necessary to introduce the new element H. This is beneficial, since with (H1) we omit an axiom which is a conditional equation, that is, an implication. In the basic case of a bounded domain semiring we thus have an equational axiomatisation, which can be handled significantly better by automated theorem provers [11,7]. In the presence of loops or recursion, however, the necessary fixpoints are introduced by conditional equations such as those of Kleene algebra.

4 General Correctness and Loops

In this section, we develop the theory of general correctness based on the axioms (L1) and (L2), and hence the Egli-Milner order $\sqsubseteq$ with least element L. The combined treatment with partial and total correctness is resumed in Section 5.

We treat full recursion, loops, correctness statements and correctness calculus, in this sequence. Our previous work [17] proves some of the following results based on the axioms (L1), (L2), (H1) and (H2); the contribution here is to show that (L1) and (L2) suffice. The combined iteration operator is new.

Throughout this section the underlying structure is a bounded domain semiring S satisfying (L1) and (L2). Additional axioms for fixpoints and loops are introduced as required. In Section 4.4 we give an outlook on pre-post specifications which require further axioms.

This section serves a twofold purpose. First, it shows that a useful theory of general correctness can be derived from very basic assumptions. Second, it is a precursor for Section 5 by deriving a semantics of while-loops.

4.1 Recursion

We first derive least fixpoints with respect to the Egli-Milner order $\sqsubseteq$ from fixpoints with respect to the natural order $\leq$. This is interesting, because $\leq$ is much simpler than $\sqsubseteq$ and hence it becomes much easier to compute the semantics of recursive programs. The results in this section require (L3) only.

Let $f : S \to S$ be the characteristic function of the recursion. We assume that f is isotone with respect to $\leq$ and $\sqsubseteq$. Moreover we assume that the least fixpoint μf and the greatest fixpoint νf of f with respect to $\leq$ exist. These assumptions are reasonable, since they are satisfied for typical programming constructs; they certainly apply for the derivation of the while-loop in Section 4.2. The least fixpoint of f with respect to the Egli-Milner order $\sqsubseteq$ is denoted by ξf.

The proof of our first result is very similar to that in [17], except that it accounts for the modified Egli-Milner order. We reproduce it to be self-contained.

Theorem 2. *Let $a \in S$, then $a = \xi f \Leftrightarrow \mu f \leq a \leq \nu f \wedge a \sqsubseteq \mu f \wedge a \sqsubseteq \nu f$.*

Proof. The forward implication is clear since ξf is the least fixpoint with respect to $\sqsubseteq$. For the backward implication, let $\mu f \leq a \leq \nu f$ and $a \sqsubseteq \mu f$ and $a \sqsubseteq \nu f$. By isotony of f we obtain $\mu f = f(\mu f) \leq f(a) \leq f(\nu f) = \nu f$ and $f(a) \sqsubseteq f(\mu f) = \mu f$ and $f(a) \sqsubseteq f(\nu f) = \nu f$. From these facts and the assumptions we obtain:

- $a \sqsubseteq f(a)$ since $a \leq \mu f + a0 \leq f(a) + a0$ and $f(a) \leq \nu f \leq a + \ulcorner(a0)\top$.
- $f(a) \sqsubseteq a$ since $f(a) \leq \mu f + f(a)0 \leq a + f(a)0$ and $a \leq \nu f \leq f(a) + \ulcorner(f(a)0)\top$.

Hence $a = f(a)$ by (L3) and Theorem 1. Let $b \in S$ such that $b = f(b)$, hence $\mu f \leq b \leq \nu f$. Then $a \sqsubseteq b$ since $a \leq \mu f + a0 \leq b + a0$ and $b \leq \nu f \leq a + \ulcorner(a0)\top$. □

Its important consequence is an explicit formula for ξf. This requires a new proof due to the modified Egli-Milner order and the limited set of axioms.

Corollary 3. *ξf exists $\Leftrightarrow \nu f \leq \mu f + \ulcorner(\nu f0)\top \Leftrightarrow \xi f = \nu f0 + \mu f$.*

Proof. Assuming $\nu f \leq \mu f + \ulcorner(\nu f0)\top$, we show $\xi f = \nu f0 + \mu f$ by Theorem 2. Since $\mu f \leq \nu f0 + \mu f \leq \nu f$ it suffices to show $\nu f0 + \mu f \sqsubseteq \mu f$ and $\nu f0 + \mu f \sqsubseteq \nu f$. But these follow since $\nu f0 + \mu f = \mu f + (\nu f0 + \mu f)0 \leq \nu f + (\nu f0 + \mu f)0$ and $\mu f \leq \nu f \leq \mu f + \ulcorner(\nu f0)\top = \mu f + \nu f0 + \ulcorner(\nu f0)\top = \nu f0 + \mu f + \ulcorner((\nu f0 + \mu f)0)\top$ using characterisation of domain in the third step.

If $\xi f = \nu f0 + \mu f$, then clearly ξf exists.

Finally assume that ξf exists. By Theorem 2 we obtain $\xi f \leq \mu f + \xi f0$ and $\nu f \leq \xi f + \ulcorner(\xi f0)\top$, hence $\nu f \leq \mu f + \xi f0 + \ulcorner(\xi f0)\top = \mu f + \ulcorner(\xi f0)\top \leq \mu f + \ulcorner(\nu f0)\top$ using characterisation of domain in the second step and $\xi f \leq \nu f$ in the third. □

4.2 While-Loops

We instantiate the results of Section 4.1 to obtain the semantics of the while-loop. To this end, let $f(x) = ax + b$. Then f is isotone with respect to $\leq$ and, by Theorem 1, also with respect to $\sqsubseteq$. To describe the extremal fixpoints with respect to $\leq$, we now assume that S is also a weak omega algebra. Then $\mu f = a^*b$ and $\nu f = a^\omega + a^*b$.

For the following results, we also need $(\mathsf{L2})$. An equivalent characterisation of $(\mathsf{L2})$ in a weak omega algebra is $a^\omega \leq \ulcorner(a^\omega 0)\top$ for all a. Indeed, instantiating $a = 1$ yields $\top \leq \ulcorner\mathsf{L}\top$ and hence $(\mathsf{L2})$, and the converse implication follows by characterisation of domain from $\ulcorner a^\omega = \ulcorner(a^\omega \ulcorner\mathsf{L}) = \ulcorner(a^\omega \top \mathsf{L}) = \ulcorner(a^\omega \top 0) = \ulcorner(a^\omega 0)$.

By $(\mathsf{L2})$ we thus obtain $\nu f = \mu f + a^\omega \leq \mu f + \ulcorner(a^\omega 0)\top \leq \mu f + \ulcorner(\nu f 0)\top$, hence ξf exists by Corollary 3, and $\xi f = \nu f 0 + \mu f = a^\omega 0 + a^* b$. This prompts us to introduce the notation $x^\circ =_{\mathrm{def}} x^\omega 0 + x^*$ combining infinite and finite iteration appropriately. In different axiomatic settings, we find the combinations $x^\omega + x^*$ called 'strong iteration' [37] and $x^\omega + x^* y$ [6].

As usual in general correctness [2,34,32,16], the semantics of the while-loop is given by $\mathsf{while}\ p\ \mathsf{do}\ a = \xi(\lambda x.pax + \overline{p})$, hence we have established the following result.

Corollary 4. *Let $p \in \mathrm{test}(S)$ and $a \in S$, then $\mathsf{while}\ p\ \mathsf{do}\ a = (pa)^\circ \overline{p}$.*

Remark. The test $\nabla x =_{\mathrm{def}} \ulcorner x^\omega$ represents the initial states of x from which x can be iterated infinitely. In [17] we show that in presence of $(\mathsf{L1})$ and $(\mathsf{L2})$, this complies with the axiomatisation of ∇ given in [8]. In particular, we obtain $x^\omega 0 = \nabla x \mathsf{L}$, and hence $\mathsf{while}\ p\ \mathsf{do}\ a = \nabla(pa)\mathsf{L} + (pa)^*\overline{p}$.

Theorem 1 shows that choice and composition are isotone with respect to $\sqsubseteq$. Hence also the conditional statement $\mathsf{if}\ p\ \mathsf{then}\ a\ \mathsf{else}\ b = pa + \overline{p}b$ is isotone in a and b. We complete this by showing that $\mathsf{while}\ p\ \mathsf{do}\ a$ is isotone in a, which follows from Theorem 6 below. It needs a few general results about our combined iteration operator. They supplement the sliding, unfold and decomposition laws for the star operation and will be useful in Section 5, too.

Lemma 5. *Let a and b be elements of a weak omega algebra. Then*

1. $a(ba)^\omega = (ab)^\omega$.
2. $a(ba)^\circ = (ab)^\circ a$.
3. $a^\circ = 1 + aa^\circ = 1 + a^\circ a$.
4. $(a+b)^\omega = (a^*b)^\omega + (a^*b)^* a^\omega$.
5. $(a+b)^\circ = (a^*b)^\circ a^\circ = (a^\circ b)^\circ a^\circ = a^\circ (ba^\circ)^\circ$.

Proof.

1. $a(ba)^\omega = aba(ba)^\omega$ by omega unfold, hence $a(ba)^\omega \leq (ab)^\omega$ by omega co-induction. By swapping a and b, this implies $(ab)^\omega = ab(ab)^\omega \leq a(ba)^\omega$.
2. $a(ba)^\circ = a((ba)^\omega 0 + (ba)^*) = a(ba)^\omega 0 + a(ba)^* = (ab)^\omega 0 + (ab)^* a = (ab)^\circ a$.
3. $1 + aa^\circ = 1 + a(a^\omega 0 + a^*) = 1 + aa^\omega 0 + aa^* = a^\omega 0 + a^* = a^\circ$. The second equality follows since $aa^\circ = a^\circ a$ by sliding with $b = 1$.

4. The part ($\le$) follows by omega co-induction from $(a+b)^\omega \le a^\omega + a^*b(a+b)^\omega$. But this follows from $(a+b)^\omega = (a+b)(a+b)^\omega = a(a+b)^\omega + b(a+b)^\omega$ again by omega co-induction. For the part ($\ge$) we observe

$$(a^*b)^*a^\omega \le (a^*b)^*a^*(a+b)^\omega = (a+b)^*(a+b)^\omega = (a+b)^\omega \ .$$

   The remaining $(a^*b)^\omega \le (a+b)^\omega$ follows by omega co-induction from

$$\begin{aligned}(a^*b)^\omega &= a^*b(a^*b)^\omega = (1+aa^*)b(a^*b)^\omega = b(a^*b)^\omega + aa^*b(a^*b)^\omega \\ &= b(a^*b)^\omega + a(a^*b)^\omega = (a+b)(a^*b)^\omega \ .\end{aligned}$$

5. The first equality follows since

$$\begin{aligned}(a+b)^\circ &= (a+b)^\omega 0 + (a+b)^* = ((a^*b)^\omega + (a^*b)^*a^\omega)0 + (a^*b)^*a^* \\ &= (a^*b)^\omega 0 + (a^*b)^*(a^\omega 0 + a^*) = (a^*b)^\circ a^\circ \ .\end{aligned}$$

   By isotony the second equality reduces to $(a^\circ b)^\circ a^\circ \le (a^*b)^\circ a^\circ$. But

$$\begin{aligned}(a^\circ b)^\omega 0 &= (a^*b + a^\omega 0)^\omega 0 = (((a^*b)^*a^\omega 0)^\omega + ((a^*b)^*a^\omega 0)^*(a^*b)^\omega)0 \\ &= (a^*b)^*a^\omega 0 + (1 + (a^*b)^*a^\omega 0)(a^*b)^\omega 0 = (a^*b)^*a^\omega 0 + (a^*b)^\omega 0 \\ &= (a^*b)^\circ a^\omega 0 \le (a^*b)^\circ a^\circ\end{aligned}$$

   and $(a^\circ b)^*a^\circ \le (a^*b)^\circ a^\circ$ follows by star induction from

$$a^\circ + a^\circ b(a^*b)^\circ a^\circ = a^\circ + a^\omega 0 + a^*b(a^*b)^\circ a^\circ = (1 + a^*b(a^*b)^\circ)a^\circ = (a^*b)^\circ a^\circ \ .$$

   The third equality now follows by sliding. □

Theorem 6. *Let $a, b \in S$ such that $a \sqsubseteq b$. Then $a^* \sqsubseteq b^*$ and $a^\circ \sqsubseteq b^\circ$.*

Proof. Assume $a \sqsubseteq b$, hence $a \le b + a0$ and $b \le a + \ulcorner(a0)\top$. Then $a^* \sqsubseteq b^*$ amounts to $a^* \le b^* + a^*0$ and $b^* \le a^* + \ulcorner(a^*0)\top$. The former follows by star induction from $1 + a(b^* + a^*0) \le 1 + (b + a0)b^* + aa^*0 = 1 + bb^* + a0 + aa^*0 \le b^* + a^*0$. The latter follows using star decomposition in

$$\begin{aligned}b^* &\le (a + \ulcorner(a0)\top)^* = a^*(\ulcorner(a0)\top a^*)^* = a^*(\ulcorner(a0)\top)^* = a^*(1 + \ulcorner(a0)\top(\ulcorner(a0)\top)^*) \\ &= a^* + a^*\ulcorner(a0)\top \le a^* + \ulcorner(a^*0)\top \ .\end{aligned}$$

The last step holds by characterisation of domain since $\ulcorner(a^*\ulcorner(a0)\top) = \ulcorner(a^*a0) \le \ulcorner(a^*0)$.

The second claim $a^\circ \sqsubseteq b^\circ$ amounts to $a^\circ \le b^\circ + a^\circ 0$ and $b^\circ \le a^\circ + \ulcorner(a^\circ 0)\top$. The former follows since $a^\omega 0 \le a^\circ 0$ and $a^* \le b^* + a^*0 \le b^\circ + a^\circ 0$ as above. The latter follows by replacing * with $^\circ$ in the calculation above, using Lemma 5. □

4.3 Correctness Statements and Calculus

Let $a \in S$ represent a program and three tests $p, q, r \in \text{test}(S)$ represent conditions on the state of a. As shown in [17], the following correctness claim is appropriate in a general correctness setting:

$$ra0 \le 0 \wedge pa\overline{q} \le \mathsf{L} \ .$$

In its first part, r describes the initial states from where termination of a has to be guaranteed. In the second part, the precondition p describes the initial states from where the postcondition q is established provided a terminates; a hypothetical transition from p to $\overline{q}$ would have to result in non-termination. Thus claims about terminating and non-terminating executions are distinguished.

The first part $ra0 \leq 0$ is the algebraic equivalent of the Hoare triple $r\,\{a\}\,1$ defined by [28,32]. It can be derived using existing Hoare calculi, with the additional triple $\neg\ulcorner(pa)^\omega\,\{\mathsf{while}\; p\; \mathsf{do}\; a\}\,1$. This triple is valid by Corollary 4, since $\neg\ulcorner(pa)^\omega(pa)^\circ\overline{p}0 = \neg\ulcorner(pa)^\omega(pa)^\circ 0 = \neg\ulcorner(pa)^\omega(pa)^\omega 0 = 0$, using $(pa)^*0 \leq (pa)^\omega 0$.

The second part $pa\overline{q} \leq \mathsf{L}$ has to be treated differently. It is implied by the 'weak correctness' claim $pa = paq$ of [37], but in contrast to our previous treatment, the two are no longer equivalent due to the restricted axioms. Hence we define the 'weaker correctness' claim $p\,[\![a]\!]\,q \Leftrightarrow_{\text{def}} pa\overline{q} \leq \mathsf{L}$. A calculus is given as follows; its correctness is proved in [17].

Theorem 7. *Let $a, b \in S$ and $p, q, r \in \text{test}(S)$. Then*

$$\begin{array}{llll} p\,[\![0]\!]\,q & p\,[\![\mathsf{L}]\!]\,q & q\,[\![1]\!]\,q & pr\,[\![1]\!]\,q \Rightarrow p\,[\![r]\!]\,q \\ \multicolumn{3}{l}{p\,[\![a]\!]\,q \wedge p\,[\![b]\!]\,q \Rightarrow p\,[\![a+b]\!]\,q} & p\,[\![a]\!]\,q \wedge q\,[\![b]\!]\,r \Rightarrow p\,[\![ab]\!]\,r \\ \multicolumn{3}{l}{rp\,[\![a]\!]\,q \wedge \overline{r}p\,[\![b]\!]\,q \Rightarrow p\,[\![\mathsf{if}\; r\; \mathsf{then}\; a\; \mathsf{else}\; b]\!]\,q} & pq\,[\![a]\!]\,q \Rightarrow q\,[\![\mathsf{while}\; p\; \mathsf{do}\; a]\!]\,\overline{p}q \end{array}$$

4.4 Pre-post Specifications

We finally discuss the extensions necessary to obtain specifications given by pre- and postconditions. An algebraic treatment in total correctness is given by [37]. In general correctness, we have to account for the termination precondition r, and hence obtain the specification $(r\,|\,p \rightsquigarrow q)$ with three components. As in Section 4.3, the test r describes the initial states from which execution must terminate. Moreover, if the precondition p holds in the initial state, the postcondition q must be established in the final states of terminating executions.

With tests p, q and r, the specification $(r\,|\,p \rightsquigarrow q)$ is axiomatised as the greatest element of S satisfying the general correctness claim:

$$x \leq (r\,|\,p \rightsquigarrow q) \Leftrightarrow rx0 = 0 \wedge px\overline{q} \leq \mathsf{L}\ .$$

We can then recover the element $\mathsf{H} =_{\text{def}} \sup\{\, x \mid x0 = 0 \,\}$ as the particular specification $\mathsf{H} = (1\,|\,1 \rightsquigarrow 1)$. While this allows us to prove several properties about H, such as $1 \leq \mathsf{H} = \mathsf{H}^2 = \mathsf{H}^*$ and $\mathsf{HL} = \mathsf{L} \neq \mathsf{H}$, we cannot derive ($\mathsf{H1}$) and ($\mathsf{H2}$). Hence the present axioms are still less restrictive than those of our previous work [17]. Further axioms about $(r\,|\,p \rightsquigarrow q)$, which also give an explicit characterisation of such specifications, are the subject of future work.

To show the usefulness of pre-post specifications, let us refine the specification $(r\,|\,q \rightsquigarrow q\overline{p})$ by the loop $\mathsf{while}\; p\; \mathsf{do}\; a$ as proposed in [16]. Algebraically, this means $\mathsf{while}\; p\; \mathsf{do}\; a \leq (r\,|\,q \rightsquigarrow q\overline{p})$, which is equivalent to $r\,\{\mathsf{while}\; p\; \mathsf{do}\; a\}\,1$ and $q\,[\![\mathsf{while}\; p\; \mathsf{do}\; a]\!]\,q\overline{p}$ by the axiom above. The former follows from $r \leq \neg\ulcorner(pa)^\omega$ as shown in Section 4.3; this means that pa cannot be iterated infinitely from states satisfying r. The latter follows from $pq\,[\![a]\!]\,q$ by Theorem 7; this is implied by $qpa = qpaq$, meaning that pa preserves the invariant q.

5 While-Programs

We now return to the combined treatment of partial, total and general correctness. The setting in this section is a weak omega algebra with tests S.

We first give a unified semantics of while-programs. Using this semantics, we state and prove the normal form theorem of [27,35] in our more general setting. This newly establishes that the result holds in general correctness. We finally extend it to while-programs with non-deterministic choice.

5.1 Unified Semantics

We first observe that the semantics of the while-loop derived for general correctness in Corollary 4 is adequate also for partial and total correctness:

- Recall from Section 3.1 that $a \cdot 0 = 0$ for all a in partial correctness. Hence we obtain $(pa)^\circ \overline{p} = (pa)^\omega 0 + (pa)^* \overline{p} = (pa)^* \overline{p} = \mu(\lambda x.pax + \overline{p})$. But this is precisely the semantics of the while-loop in this setting, since recursion is modelled by least fixpoints with respect to $\leq$.
- Recall from Section 3.2 that $\top \cdot 0 = \top$ in total correctness. It follows that $a^\omega 0 = a^\omega \top 0 = a^\omega \top = a^\omega$. Hence we obtain the proper semantics also in this setting by $(pa)^\circ \overline{p} = (pa)^\omega 0 + (pa)^* \overline{p} = (pa)^\omega + (pa)^* \overline{p} = \nu(\lambda x.pax + \overline{p})$, since recursion is modelled by greatest fixpoints with respect to $\leq$.

We can thus define a unified semantics of program constructs, including the while-loop, which is appropriate for partial, total and general correctness:

$$\begin{aligned} a \,;\, b &=_{\mathrm{def}} ab \\ \mathsf{if}\ p\ \mathsf{then}\ a\ \mathsf{else}\ b &=_{\mathrm{def}} pa + \overline{p}b \\ \mathsf{if}\ p\ \mathsf{then}\ a &=_{\mathrm{def}} pa + \overline{p} \\ \mathsf{while}\ p\ \mathsf{do}\ a &=_{\mathrm{def}} (pa)^\circ \overline{p} \end{aligned}$$

This gives the correct semantics in each particular correctness approach. Results proved using this semantics, applying only the axioms of Section 2, hold in partial, total and general correctness. We call programs composed from atomic programs by the above constructs *while-programs.*

Remark. Since the semantics of the while-loop has been derived as a special case of recursion, the question is raised whether the general correctness semantics of full recursion $\xi f = \nu f 0 + \mu f$ obtained in Corollary 3 also suits partial and total correctness.

For partial correctness, this is true, since $\nu f 0 + \mu f = 0 + \mu f = \mu f$ and μf is the appropriate fixpoint operator.

For total correctness the appropriate fixpoint operator is νf, but $\nu f 0 + \mu f \neq \nu f$ in general, as the following counterexample shows. It uses the meet operation inf, which is available in relational models such as [22]. Let $f(x) = \inf\{1, x\}$, hence $\nu f = f(\nu f) \leq 1$. Since $f(0) = 0$ and $f(1) = 1$, we have $\mu f = 0$ and $\nu f = 1$, but $\nu f 0 + \mu f = 1 \cdot 0 + 0 = 0 \neq 1 = \nu f$.

5.2 Normal Form Theorem

A while-program is in *normal form* if it has the form a ; while p do b where a and b are while-free. Using this definition, [27] goes on to prove that every while-program can be transformed into normal form; the statement is made more precise below. The given proof is valid in partial correctness only, since it uses both the axiom $\top \cdot 0 = 0$ and the least fixpoint $(pa)^*\overline{p}$ as the semantics of the loop while p do a.

Making this observation, [35] reproduces the result in total correctness, using the axiom $\top \cdot 0 = \top$ and the greatest fixpoint $(pa)^\omega + (pa)^*\overline{p}$ for the above loop. Actually, the setting is demonic refinement algebra [37], which is interdefinable with weak omega algebra extended by $\top \cdot 0 = \top$ as shown in [24].

Our goal in the following is to prove the normal form result using the unified semantics of Section 5.1 and only axioms of weak omega algebra with tests. This subsumes the theorems of [27,35] and newly establishes the result for general correctness. The structure of the proof is the same as for partial and total correctness: while-programs are successively transformed by moving inner while-loops to the outside.

We discuss the three kinds of program transformations necessary to perform these steps, and prove their correctness. Before that, let us introduce the necessary tools and formally state the theorem.

For $a \in S$ and $p \in \mathrm{test}(S)$ we say that *a preserves p* if $pa \leq ap$ and $\overline{p}a \leq a\overline{p}$. Observe that $pa \leq ap$ is equivalent to $pa = pap$. Moreover, tests preserve any test. The following two lemmas record general facts related to preservation and the import of tests into iterations. Some of these are known from [26,29]; we are particularly interested in the properties of our combined iteration $^\circ$.

Lemma 8. *Let a and b be elements of a weak omega algebra such that $ba \leq ab$. Then*

1. $ba^* \leq a^*b$ *and* $b^*a \leq ab^*$.
2. $ba^\omega \leq a^\omega$ *and* $(ab)^\omega \leq a^\omega$.
3. $ba^\circ \leq a^\circ b$.

Proof. Assume $ba \leq ab$.

1. $b + a^*ba \leq b + a^*ab = (1 + a^*a)b \leq a^*b$, hence $ba^* \leq a^*b$ by star induction. $a + bab^* \leq a + abb^* = a(1 + bb^*) \leq ab^*$, hence $b^*a \leq ab^*$ by star induction.
2. $ba^\omega = baa^\omega \leq aba^\omega$, hence $ba^\omega \leq a^\omega$ by omega co-induction. For the second claim we have $(ab)^\omega = a(ba)^\omega \leq a(ab)^\omega$ by Lemma 5, thus $(ab)^\omega \leq a^\omega$ by omega co-induction.
3. $ba^\circ = b(a^\omega 0 + a^*) = ba^\omega 0 + ba^* \leq a^\omega 0 + a^*b = (a^\omega 0 + a^*)b = a^\circ b$. □

Lemma 9. *Let $a \in S$ and $p \in \mathrm{test}(S)$ such that $pa \leq ap$. Then*

1. $pa^* = p(pa)^*$.
2. $pa^\omega = p(pa)^\omega = (pa)^\omega$.
3. $pa^\circ = p(pa)^\circ$.

Proof. Assume $pa \leq ap$.

1. Since $ppa \leq pap$ by the assumption, we have $p(pa)^* \leq (pa)^*p$ by Lemma 8, hence $p + p(pa)^*a \leq p + p(pa)^*pa = p(1 + (pa)^*pa) \leq p(pa)^*$, and therefore $pa^* \leq p(pa)^*$ by star induction. The converse inequality follows immediately.
2. $pa^\omega = paa^\omega = papa^\omega$, hence $pa^\omega \leq (pa)^\omega$ by omega co-induction, and moreover $(pa)^\omega = pa(pa)^\omega \leq paa^\omega = pa^\omega$. Thus $pa^\omega = ppa^\omega = p(pa)^\omega$.
3. $pa^\circ = p(a^\omega 0 + a^*) = pa^\omega 0 + pa^* = p(pa)^\omega 0 + p(pa)^* = p((pa)^\omega 0 + (pa)^*) = p(pa)^\circ$. □

For $s \in S$ and $p, q \in \text{test}(S)$ we say that s *assigns* p *to* q if $s = s(pq + \overline{p}\,\overline{q})$. Intuitively, s models a program that assigns the value of p to a new Boolean variable whose value is tested by q. The consequence of using a *new* variable is that programs can be augmented by the assigning subprogram s without essential changes and that q is preserved by components of the original program.

Theorem 10. *Every while-program, suitably augmented with assigning subprograms, is equivalent to a while-program in normal form under certain preservation assumptions.*

The following construction makes explicit where to add assigning subprograms and which preservation assumptions are required.

Remark. Both previous versions of the normal form theorem [27,35] talk about augmenting with 'subprograms of the form' s ; $(pq + \overline{p}\,\overline{q})$, and [27] adds that s is an 'uninterpreted atomic program symbol'. Let us therefore clarify our understanding of this. Syntactically, with each augmentation a new atomic subprogram is inserted, which we denote by s. It is semantically interpreted as an element $s \in S$ such that $s = s(pq + \overline{p}\,\overline{q})$. The program transformation is an equation in which s is *universally* quantified, over all elements satisfying this equation. For some values of s, such as $s = 0$ or $s = \top \cdot 0$, the claim is trivial; but there are also sensible choices as described above.

The individual program transformations, that move while-loops out of the three kinds of programming constructs, are stated and proved similarly to [27,35]. However, the different semantics of the while-loop must be taken into account; this is done by Lemmas 5, 8 and 9 about the $^\circ$ operator. Moreover, we take care to use only axioms of weak omega algebra with tests.

The first transformation moves two while-programs in normal form out of a conditional, and hence into normal form. Note that any while-free program a can be brought into normal form a ; while 0 do 1. This can be applied first if one of the branches is while-free as, for example, in the one-armed conditional. A similar remark applies to programs in sequential composition below.

Lemma 11. *Let s assign p to q and let a_1, a_2, b_1, b_2 preserve q. Then*

$$\begin{aligned} & s \text{ ; if } p \text{ then } (a_1 \text{ ; while } r_1 \text{ do } b_1) \text{ else } (a_2 \text{ ; while } r_2 \text{ do } b_2) \\ = {} & s \text{ ; (if } q \text{ then } a_1 \text{ else } a_2) \text{ ; while } qr_1 + \overline{q}r_2 \text{ do (if } q \text{ then } b_1 \text{ else } b_2) \,. \end{aligned}$$

Proof. Since $s = s(pq + \overline{p}\,\overline{q})$, it suffices to show

$$\begin{aligned} &(pq + \overline{p}\,\overline{q})(pa_1(r_1b_1)^\circ\overline{r_1} + \overline{p}a_2(r_2b_2)^\circ\overline{r_2}) \\ &= (pq + \overline{p}\,\overline{q})(qa_1 + \overline{q}a_2)((qr_1 + \overline{q}r_2)(qb_1 + \overline{q}b_2))^\circ\overline{qr_1 + \overline{q}r_2} \ . \end{aligned}$$

The right hand side of this equation is simplified by

– $(pq + \overline{p}\,\overline{q})(qa_1 + \overline{q}a_2) = pqqa_1 + pq\overline{q}a_2 + \overline{p}\,\overline{q}qa_1 + \overline{p}\,\overline{q}\,\overline{q}a_2 = pqa_1 + \overline{p}\,\overline{q}a_2$,
– $(qr_1 + \overline{q}r_2)(qb_1 + \overline{q}b_2) = qr_1qb_1 + qr_1\overline{q}b_2 + \overline{q}r_2qb_1 + \overline{q}r_2\overline{q}b_2 = qr_1b_1 + \overline{q}r_2b_2$,
– $\overline{qr_1 + \overline{q}r_2} = (\overline{q} + \overline{r_1})(q + \overline{r_2}) = \overline{q}\,\overline{r_2} + q\overline{r_1} + \overline{r_1}\,\overline{r_2} = q\overline{r_1} + \overline{q}\,\overline{r_2}$.

Similarly simplifying the left hand side, it suffices to show

$$pqa_1(r_1b_1)^\circ\overline{r_1} + \overline{p}\,\overline{q}a_2(r_2b_2)^\circ\overline{r_2} = (pqa_1 + \overline{p}\,\overline{q}a_2)(qr_1b_1 + \overline{q}r_2b_2)^\circ(q\overline{r_1} + \overline{q}\,\overline{r_2}) \ .$$

But this follows since

$$\begin{aligned} &pqa_1(qr_1b_1 + \overline{q}r_2b_2)^\circ(q\overline{r_1} + \overline{q}\,\overline{r_2}) = pqa_1q(qr_1b_1 + \overline{q}r_2b_2)^\circ(q\overline{r_1} + \overline{q}\,\overline{r_2}) \\ &= pqa_1q(qqr_1b_1 + q\overline{q}r_2b_2)^\circ(q\overline{r_1} + \overline{q}\,\overline{r_2}) = pqa_1q(qr_1b_1)^\circ(q\overline{r_1} + \overline{q}\,\overline{r_2}) \\ &= pqa_1q(r_1b_1)^\circ(q\overline{r_1} + \overline{q}\,\overline{r_2}) = pqa_1q(r_1b_1)^\circ q(q\overline{r_1} + \overline{q}\,\overline{r_2}) \\ &= pqa_1q(r_1b_1)^\circ q\overline{r_1} = pqa_1q(r_1b_1)^\circ\overline{r_1} = pqa_1(r_1b_1)^\circ\overline{r_1} \end{aligned}$$

and similarly $\overline{p}\,\overline{q}a_2(qr_1b_1 + \overline{q}r_2b_2)^\circ(q\overline{r_1} + \overline{q}\,\overline{r_2}) = \overline{p}\,\overline{q}a_2(r_2b_2)^\circ\overline{r_2}$ using Lemmas 8 and 9 because a_1, a_2, b_1, b_2 preserve q. □

The second transformation moves a while-program in normal form out of a while-loop, and hence into normal form by subsequent application of Lemma 11.

Lemma 12

$$\mathsf{while}\ p\ \mathsf{do}\ (a\ ;\ \mathsf{while}\ q\ \mathsf{do}\ b) = \mathsf{if}\ p\ \mathsf{then}\ (a\ ;\ \mathsf{while}\ p + q\ \mathsf{do}\ (\mathsf{if}\ q\ \mathsf{then}\ b\ \mathsf{else}\ a)) \ .$$

Proof. The claim follows since

$$\begin{aligned} &pa((p + q)(qb + \overline{q}a))^\circ\overline{p + q} + \overline{p} = pa(qb + \overline{q}pa)^\circ\overline{p}\,\overline{q} + \overline{p} \\ &= pa(qb)^\circ(\overline{q}pa(qb)^\circ)^\circ\overline{p}\,\overline{q} + \overline{p} = (pa(qb)^\circ(\overline{q}pa(qb)^\circ)^\circ\overline{q} + 1)\overline{p} \\ &= (pa(qb)^\circ\overline{q}(pa(qb)^\circ\overline{q})^\circ + 1)\overline{p} = (pa(qb)^\circ\overline{q})^\circ\overline{p} \ , \end{aligned}$$

using Lemma 5 several times. □

The third transformation moves two while-programs in normal form out of a sequential composition, and hence into normal form. Consider the composition a_1 ; (while p_1 do b_1) ; a_2 ; (while p_2 do b_2) of two programs in normal form. We first replace p_1 with a new test q that is preserved by the second program a_2 ; while p_2 do b_2. By Lemma 8 it suffices to assume that a_2 and b_2 preserve q.

Lemma 13. *Let s assign p to q. Then*

$$s\ ;\ (\mathsf{while}\ p\ \mathsf{do}\ (a\ ;\ s))\ ;\ b = s\ ;\ (\mathsf{while}\ q\ \mathsf{do}\ (a\ ;\ s))\ ;\ b \ .$$

Proof. Since $s = s(pq + \overline{p}\,\overline{q})$, it suffices to show, using $c = as$,

$$(pq + \overline{p}\,\overline{q})(pc(pq + \overline{p}\,\overline{q}))^{\circ}\overline{p} = (pq + \overline{p}\,\overline{q})(qc(pq + \overline{p}\,\overline{q}))^{\circ}\overline{q} \,.$$

But this follows from

$$((pq + \overline{p}\,\overline{q})pc)^{\circ}(pq + \overline{p}\,\overline{q})\overline{p} = (pqc)^{\circ}\overline{p}\,\overline{q} = ((pq + \overline{p}\,\overline{q})qc)^{\circ}(pq + \overline{p}\,\overline{q})\overline{q}$$

by sliding according to Lemma 5. □

The two occurrences of the assigning subprogram s can be absorbed into a_1 and b_1. We thus return to the composition a_1 ; $(\mathsf{while}\, p_1 \,\mathsf{do}\, b_1)$; a_2 ; $(\mathsf{while}\, p_2 \,\mathsf{do}\, b_2)$ and assume that p_1 is preserved by a_2 ; $\mathsf{while}\, p_2 \,\mathsf{do}\, b_2$ without losing generality. By the following lemma, we absorb this program into the first loop, introducing a copy for the case where the first loop is not executed.

Lemma 14. *Let b preserve p. Then*

$$(\mathsf{while}\, p \,\mathsf{do}\, a) \,;\, b = \mathsf{if}\, \overline{p} \,\mathsf{then}\, b \,\mathsf{else}\, (\mathsf{while}\, p \,\mathsf{do}\, (a \,;\, \mathsf{if}\, \overline{p} \,\mathsf{then}\, b)) \,.$$

Proof. If we can show $p(pa(\overline{p}b + p))^{\circ}\overline{p} = pa(pa)^{\circ}\overline{p}b$, the claim holds since

$$\overline{p}b + p(pa(\overline{p}b + p))^{\circ}\overline{p} = \overline{p}b + pa(pa)^{\circ}\overline{p}b = (1 + pa(pa)^{\circ})\overline{p}b = (pa)^{\circ}\overline{p}b$$

by Lemma 5. But the missing step follows because

$$\begin{aligned} &p(pa(\overline{p}b + p))^{\circ}\overline{p} = p(1 + pa(\overline{p}b + p)(pa(\overline{p}b + p))^{\circ})\overline{p} \\ &= pa(\overline{p}b + p)(pa(\overline{p}b + p))^{\circ}\overline{p} = pa((\overline{p}b + p)pa)^{\circ}(\overline{p}b + p)\overline{p} = pa(\overline{p}bpa + pa)^{\circ}\overline{p}b\overline{p} \\ &= pa(\overline{p}b0 + pa)^{\circ}\overline{p}b = pa(pa)^{\circ}(\overline{p}b0)^{\circ}\overline{p}b = pa(pa)^{\circ}\overline{p}b(0\overline{p}b)^{\circ} = pa(pa)^{\circ}\overline{p}b \,, \end{aligned}$$

using Lemma 5 several times. □

By applying this transformation, we obtain the program

$$\begin{aligned} &a_1 \,;\, \mathsf{if}\, \overline{p_1} \,\mathsf{then}\, (a_2 \,;\, \mathsf{while}\, p_2 \,\mathsf{do}\, b_2) \\ &\qquad \mathsf{else}\, (\mathsf{while}\, p_1 \,\mathsf{do}\, (b_1 \,;\, \mathsf{if}\, \overline{p_1} \,\mathsf{then}\, (a_2 \,;\, \mathsf{while}\, p_2 \,\mathsf{do}\, b_2))) \,. \end{aligned}$$

Afterwards, we proceed as in [27,35]. First, the else branch is normalised by moving the inner while-loop outside of the inner conditional and the outer while-loop using Lemmas 11 and 12. Second, the outer conditional is normalised again by Lemma 11. We thus obtain a program in normal form.

By successively applying the transformations of Lemmas 11–14 we can move while-loops from the inside to the outside of the program, until the whole program is in normal form. This proves Theorem 10.

5.3 Non-deterministic Programs

The choice operator + occurs in while-programs only in a restricted form, namely within the conditional $\mathsf{if}\, p \,\mathsf{then}\, a \,\mathsf{else}\, b = pa + \overline{p}b$. However, the normal form theorem does not assume that while-programs are deterministic; in fact the atomic subprograms might as well be non-deterministic. This observation helps us to turn an explicit non-deterministic choice into the restricted form of the conditional.

For $t \in S$ and $r \in \mathrm{test}(S)$ we say that t *tosses* r if $t = trt = t\overline{r}t$. Intuitively, t models a program that non-deterministically assigns true or false to a new Boolean variable whose value is tested by r. It is impossible to distinguish which value was assigned between two immediately following tosses.

Programs composed from the constructs allowed for while-programs and the non-deterministic choice

$$a \mathsf{~or~} b =_{\mathrm{def}} a + b$$

are called *non-deterministic while-programs.* We can extend Theorem 10 to such programs as follows.

Theorem 15. *Every* non-deterministic *while-program, suitably augmented with assigning* and tossing *subprograms, is equivalent to a while-program in normal form under certain preservation assumptions.*

In view of the transformations of Section 5.2, it remains to give one that deals with the non-deterministic choice between two while-programs in normal form. The following lemma eliminates the choice operation.

Lemma 16. *Let t toss r. Then*

$$t \mathbin{;} (a \mathsf{~or~} b) = t \mathbin{;} \mathsf{if~} r \mathsf{~then~} (t \mathbin{;} a) \mathsf{~else~} (t \mathbin{;} b) \ .$$

Proof. Since $t = trt = t\overline{r}t$, the claim follows by $t(a+b) = ta + tb = trta + t\overline{r}tb = t(rta + \overline{r}tb)$. □

Applying it to the program $t \mathbin{;} ((a_1 \mathbin{;} \mathsf{while~} p_1 \mathsf{~do~} b_1) \mathsf{~or~} (a_2 \mathbin{;} \mathsf{while~} p_2 \mathsf{~do~} b_2))$, where the choice is between programs in normal form, we obtain

$$t \mathbin{;} \mathsf{if~} r \mathsf{~then~} (t \mathbin{;} a_1 \mathbin{;} \mathsf{while~} p_1 \mathsf{~do~} b_1) \mathsf{~else~} (t \mathbin{;} a_2 \mathbin{;} \mathsf{while~} p_2 \mathsf{~do~} b_2) \ .$$

But this is brought into normal form using Lemma 11. Together with Lemmas 11–14 this proves Theorem 15.

Remark. We briefly discuss tossing elements. To this end, let t toss r. Then $tt = t(r + \overline{r})t = trt + t\overline{r}t = t + t = t$. To simplify the transformation above, the value of r is erased by another toss immediately after the conditional choice is made. Hence there is no way to determine the outcome of the first toss within the programs a and b of Lemma 16.

6 Conclusion

This work makes a point for using less axioms. It shows that many results of general correctness can be derived in a very basic setting, despite the complexity caused by the Egli-Milner order and the finer termination information. It also shows that by omitting characteristic axioms, a unified treatment of partial, total and general correctness is possible, with a common semantics of programs and common proofs of program transformations.

Future investigations concern axioms for pre-post specifications, refinement, and further operators for general correctness [16], as well as applications in the area of hybrid systems [23]. We also consider program transformations between symmetric linear and tail recursion, known from a total correctness setting [20].

Acknowledgement. I thank all anonymous referees for their valuable remarks and helpful suggestions.

References

1. Aarts, C.J.: Galois connections presented calculationally. Master's thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology (1992)
2. de Bakker, J.W.: Semantics and termination of nondeterministic recursive programs. In: Michaelson, S., Milner, R. (eds.) Automata, Languages and Programming: Third International Colloquium, pp. 435–477. Edinburgh University Press, Edinburgh (1976)
3. Berghammer, R., Zierer, H.: Relational algebraic semantics of deterministic and nondeterministic programs. Theoretical Computer Science 43, 123–147 (1986)
4. Broy, M., Gnatz, R., Wirsing, M.: Semantics of nondeterministic and noncontinuous constructs. In: Bauer, F.L., Broy, M. (eds.) Program Construction. LNCS, vol. 69, pp. 553–592. Springer, Heidelberg (1979)
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. Cohen, E.: Separation and reduction. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 45–59. Springer, Heidelberg (2000)
7. Dang, H.-H., Höfner, P.: First-order theorem prover evaluation w.r.t. relation- and Kleene algebra. In: Berghammer, R., Möller, B., Struth, G. (eds.) Relations and Kleene Algebra in Computer Science: PhD Programme at RelMiCS10/AKA5, Report 2008-04, pp. 48–52. Institut für Informatik, Universität Augsburg (April 2008)
8. Desharnais, J., Möller, B., Struth, G.: Algebraic notions of termination. Report 2006-23, Institut für Informatik, Universität Augsburg (October 2006)
9. Desharnais, J., Möller, B., Struth, G.: Kleene algebra with domain. ACM Transactions on Computational Logic 7(4), 798–833 (2006)
10. Desharnais, J., Struth, G.: Domain axioms for a family of near-semirings. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 330–345. Springer, Heidelberg (2008)
11. Desharnais, J., Struth, G.: Modal semirings revisited. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 360–387. Springer, Heidelberg (2008)
12. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Englewood Cliffs (1976)
13. Dijkstra, R.M.: Computation calculus bridging a formalization gap. Science of Computer Programming 37(1–3), 3–36 (2000)

14. Dunne, S.: Recasting Hoare and He's Unifying Theory of Programs in the context of general correctness. In: Butterfield, A., Strong, G., Pahl, C. (eds.) 5th Irish Workshop on Formal Methods, Electronic Workshops in Computing. The British Computer Society (July 2001)
15. Dunne, S., Galloway, A.: Lifting general correctness into partial correctness is *ok*. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 215–232. Springer, Heidelberg (2007)
16. Dunne, S., Hayes, I., Galloway, A.: Reasoning about loops in total and general correctness. In: Butterfield, A. (ed.) Second International Symposium on Unifying Theories of Programming. LNCS, vol. 5713, Springer, Heidelberg (to appear)
17. Guttmann, W.: General correctness algebra. In: Berghammer, R., Jaoua, A.M., Möller, B. (eds.) RelMiCS/AKA 2009. LNCS, vol. 5827, pp. 150–165. Springer, Heidelberg (2009)
18. Guttmann, W.: Lazy UTP. In: Butterfield, A. (ed.) Second International Symposium on Unifying Theories of Programming. LNCS, vol. 5713. Springer, Heidelberg (to appear)
19. Guttmann, W., Möller, B.: Modal design algebra. In: Dunne, S., Stoddart, W. (eds.) UTP 2006. LNCS, vol. 4010, pp. 236–256. Springer, Heidelberg (2006)
20. Guttmann, W., Möller, B.: Normal design algebra. Journal of Logic and Algebraic Programming 79(2), 144–173 (2010)
21. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580, 583 (1969)
22. Hoare, C.A.R., He, J.: Unifying theories of programming. Prentice-Hall Europe (1998)
23. Höfner, P., Möller, B.: An algebra of hybrid systems. Journal of Logic and Algebraic Programming 78(2), 74–97 (2009)
24. Höfner, P., Möller, B., Solin, K.: Omega algebra, demonic refinement algebra and commands. In: Schmidt, R.A. (ed.) RelMiCS/AKA 2006. LNCS, vol. 4136, pp. 222–234. Springer, Heidelberg (2006)
25. Jacobs, D., Gries, D.: General correctness: A unification of partial and total correctness. Acta Informatica 22(1), 67–83 (1985)
26. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Information and Computation 110(2), 366–390 (1994)
27. Kozen, D.: Kleene algebra with tests. ACM Transactions on Programming Languages and Systems 19(3), 427–443 (1997)
28. Kozen, D.: On Hoare logic and Kleene algebra with tests. ACM Transactions on Computational Logic 1(1), 60–76 (2000)
29. Möller, B.: Lazy Kleene algebra. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 252–273. Springer, Heidelberg (2004)
30. Möller, B.: The linear algebra of UTP. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 338–358. Springer, Heidelberg (2006)
31. Möller, B., Struth, G.: Algebras of modal operators and partial correctness. Theoretical Computer Science 351(2), 221–239 (2006)
32. Möller, B., Struth, G.: WP is WLP. In: MacCaull, W., Winter, M., Düntsch, I. (eds.) RelMiCS 2005. LNCS, vol. 3929, pp. 200–211. Springer, Heidelberg (2006)
33. Moszkowski, B.C.: A complete axiomatization of Interval Temporal Logic with infinite time. In: Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science, pp. 241–252. IEEE, Los Alamitos (2000)
34. Nelson, G.: A generalization of Dijkstra's calculus. ACM Transactions on Programming Languages and Systems 11(4), 517–561 (1989)

35. Solin, K.: A while program normal form theorem in total correctness. In: Berghammer, R., Jaoua, A.M., Möller, B. (eds.) RelMiCS/AKA 2009. LNCS, vol. 5827, pp. 322–336. Springer, Heidelberg (2009)
36. Søndergaard, H., Sestoft, P.: Non-determinism in functional languages. The Computer Journal 35(5), 514–523 (1992)
37. von Wright, J.: Towards a refinement algebra. Science of Computer Programming 51(1–2), 23–45 (2004)

Unifying Theories of Programming That Distinguish Nontermination and Abort

Ian J. Hayes[1], Steve E. Dunne[2], and Larissa Meinicke[3]

[1] The University of Queensland, Brisbane, 4072, Australia
[2] School of Computing, University of Teesside, Middlesbrough, TS1 3BA, UK
[3] Macquarie University, Sydney, Australia

Abstract. In this paper we focus on the relationship between a number of specification models. The models are formulated in the Unifying Theories of Programming of Hoare and He, but correspond to widely used specification models. We cover issues such as partial correctness, total correctness, and general correctness.

The properties we use to distinguish the models are these:

– whether they allow the specification of assumptions about the initial state outside of which no guarantees are given about the behaviour of the program, i.e., the program may "abort";
– whether a specification may allow or even require nontermination as a valid (non-aborting) outcome; and
– whether they allow the expression of *tests* or *enabling conditions*, outside of which the program has no possible behaviour.

When considering termination, we consider both an abstract model, which only distinguishes whether a program terminates or not, as well as models that include a notion of time: either abstract time representing a notion of progress or real-time.

1 Introduction

The aim of this paper is to better understand the relationships between a number of models of program specifications. We are interested in whether they can express properties such as total correctness, partial correctness, general correctness, timing properties, and reactive behaviour. As a framework to relate these models we use Hoare and He's Unifying Theories of Programming (UTP) [1], because this theory is general enough to do this succinctly.[1] Section 2 addresses UTP designs (or specifications), which support total-correctness specifications in the form of a precondition and a pre-post relation. These correspond to specifications in VDM [2], the refinement calculus [3,4,5,6], and B [7]. Section 3 examines Z specifications [8,9], which form the least expressive model considered here.

Section 4 introduces a new model that extends designs to distinguish abort and nontermination; this allows specification of both total- and partial-correctness properties.

[1] The UTP models that we consider are based on homogeneous relations between states. These are not rich enough to express both demonic and angelic choice simultaneously, which is possible in predicate transformer models, but such relational models are sufficient for the properties explored in this paper.

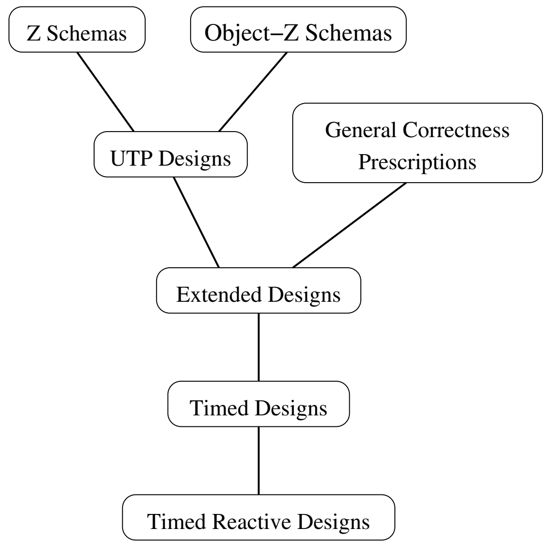


Fig. 1. Relationships between models: most general at the bottom

Extended designs can also express general-correctness properties. General correctness allows a termination set to be defined, but does not model a program aborting. General correctness is explored in Section 5, and Section 6 highlights the distinction between an assumption on the initial state and a termination set.

Section 7 generalises extended designs to allow timing properties to be expressed by using an observation of the time, τ, with $\tau' = \infty$ signifying nontermination; this approach is similar to that used by Hehner [10]. Section 8 generalises this further to allow the expression of reactive properties of traces of the program variables over time; this model corresponds to the real-time refinement calculus [11,12,13,14,15]. Figure 1 summarises the relationships between the models graphically.

Before getting into the details of the models, we warn the reader that the literature on the different models uses the term "precondition" in various different ways. It may be any of

- an *assumption*: a condition characterising those initial states from which the program is required to work properly or not be enabled, but outside of which it may "abort" (for example, by crashing), or terminate capriciously with an incorrect result, or fail to terminate at all by executing forever;
- a *termination set*: the initial states from which termination is required;
- an *enabling condition* (or *guard* or *test*): execution can only begin from initial states satisfying the condition (i.e., it is infeasible outside the enabling condition); or
- combinations of the above.

In all cases it is a predicate on the initial state. These different interpretations of the term “precondition” can lead to misunderstandings when moving from one model to another. One aim of this paper is to clarify these distinctions by making it clear how “precondition” is interpreted in each model.

Syntactic substitution. We use the notation $r\left[\frac{e}{v}\right]$ to stand for the relation r with every free occurrence of the variable name v replaced by the expression e. This can also be generalised so that v is a list of variable names and e a corresponding list of expressions.

2 UTP Designs

In Hoare and He's Unifying Theories of Programming (UTP), designs (or specifications) and programs are modelled via relations between the before and after program states [1]. A special boolean observation *okay* is used to model program termination, which in the model is indistinguishable from the program not aborting. Hoare and He [1] also use the term *stable*. A *design* has the syntax $(p \vdash w)$, where p is a single-state predicate on the before-values of the program variables and w characterises a relation between before- and after-values of the program variables.[2] For UTP designs the *precondition*, p, represents both an assumption on the initial state and the set of initial states from which termination is required. The semantics of a design is given by a relation characterised by the predicate

$$okay \wedge p \Rightarrow okay' \wedge w \;, \tag{1}$$

where unprimed variable names correspond to the initial values of the variables and primed names to their final values. If the program starts (i.e., *okay* holds) in an initial state in which p holds, the program will terminate (i.e., $okay'$ will hold) and relation w will hold between the initial and final states. Neither p nor w may refer to the observations *okay* and $okay'$.[3] Designs only model total correctness. To simplify the presentation below, we do not distinguish between a relation and the predicate characterising that relation.

A design $(p_0 \vdash w_0)$ *is refined by* another design $(p_1 \vdash w_1)$, written

$$(p_0 \vdash w_0) \sqsubseteq (p_1 \vdash w_1) \;,$$

provided the semantic relation defined by the latter implies (is included in) the semantic relation defined by the former, that is,

$$[(okay \wedge p_1 \Rightarrow okay' \wedge w_1) \Rightarrow (okay \wedge p_0 \Rightarrow okay' \wedge w_0)] \;, \tag{2}$$

where, as in Hoare and He [1], the notation $[P]$ stands for the universal quantification of P over all variables in the alphabet (including *okay* and $okay'$). Refinement condition (2) holds if and only if

$$[p_0 \Rightarrow (p_1 \wedge (w_1 \Rightarrow w_0))] \;. \tag{3}$$

[2] Hoare and He [1] allow p to be a relation in general, but then introduce a constraint (H3) that requires p to be single-state. The designs we describe here are thus their H3-designs.

[3] Allowing p and w to refer to *okay* and $okay'$ doesn't add anything because $(p \vdash w)$ is semantically equivalent to $(p\left[\frac{\mathtt{true}}{okay}\right] \vdash w\left[\frac{\mathtt{true},\mathtt{true}}{okay,okay'}\right])$.

There are three interesting extreme cases of designs:

$$\begin{aligned}
\textbf{abort}_{UTP} &\mathrel{\widehat{=}} (\text{false} \vdash \text{true}) \\
\textbf{terminates}_{UTP} &\mathrel{\widehat{=}} (\text{true} \vdash \text{true}) \\
\textbf{magic}_{UTP} &\mathrel{\widehat{=}} (\text{true} \vdash \text{false})
\end{aligned}$$

UTP designs form a complete lattice under the refinement ordering, with least element $\textbf{abort}_{UTP}$ and greatest element $\textbf{magic}_{UTP}$. The design $\textbf{terminates}_{UTP}$ is the specification that guarantees termination but nothing else. Note that the design $(\text{false} \vdash w)$ is semantically equivalent to $\textbf{abort}_{UTP}$ for any relation w. A design is *infeasible* in any state, v, for which the precondition p holds but the relation w doesn't relate v to any final state, i.e., the set of infeasible states are those satisfying $p \wedge \neg\, (\exists\, v' \bullet w)$. The design $\textbf{magic}_{UTP}$ is everywhere infeasible.

3 Z Specifications

Let S be a Z schema [8,9] representing the program state, then the Z schema

$$\begin{array}{|l}
\underline{\;Z\;\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\
S \\
S' \\
\hline
R \\
\hline
\end{array}$$

can be used as a specification. Here S' stands for S with all components decorated with a prime, and R is a predicate relating the components of S and S'. To save space we write the above schema in its equivalent *horizontal* form: $[S;\ S' \mid R]$.

In Z, the precondition of an operation is the predicate $(\exists\, S' \bullet R)$, i.e., any initial state for which there exists a corresponding final state. The precondition is the domain of the relation R. Under the conventional so-called *contract*, or *non-blocking*, interpretation of a Z operation schema [16], a precondition in Z represents, as for a UTP design, both an assumption on the initial state and the set of initial states on which termination is required. Thus the schema Z above *is refined by* a schema $[S;\ S' \mid Q]$, i.e., $[S;\ S' \mid R] \sqsubseteq [S;\ S' \mid Q]$, provided

$$[(\exists\, S' \bullet R) \Rightarrow ((\exists\, S' \bullet Q) \wedge (Q \Rightarrow R))]\ .$$

Relating Z schemas and designs. Any Z schema of this form can be uniquely mapped to a UTP design by the function ZD defined as follows.

$$ZD([S;\ S' \mid R]) \mathrel{\widehat{=}} ((\exists\, S' \bullet R) \vdash R)$$

This mapping preserves the refinement ordering, and the image of the mapping forms a subtheory of designs [1, Chap. 4].

There are two interesting extreme cases:

$$\begin{aligned} \textbf{abort}_Z &\mathrel{\widehat=} [S;\ S' \mid \text{false}] \\ \textbf{terminates}_Z &\mathrel{\widehat=} [S;\ S' \mid \text{true}] \end{aligned}$$

where $\textbf{abort}_Z$ is the least program in the refinement ordering. These schemas correspond to their equivalents using designs, i.e.,

$$\begin{aligned} \mathit{ZD}(\textbf{abort}_Z) &= \textbf{abort}_{\mathit{UTP}} \\ \mathit{ZD}(\textbf{terminates}_Z) &= \textbf{terminates}_{\mathit{UTP}} \end{aligned}$$

but note that one cannot represent magic by a Z specification. More generally, Z cannot represent infeasible specifications and hence the Z model is not as expressive as UTP designs (or VDM pre/post specifications, or B specifications).

In Object-Z [17,18] operations are always terminating and the domain of the relation R is treated as the operation's enabling condition[4]. Hence for Object-Z the schema Z above is mapped to a design as follows:

$$\mathit{OZD}([S;\ S' \mid R]) \mathrel{\widehat=} (\text{true} \vdash R)\ .$$

In the Object-Z model one can express infeasible specifications, although all specifications are terminating and non-aborting.

4 Distinguishing Nontermination and Abort

For reactive and real-time programs, it is often desirable to distinguish between abort and nontermination, because nontermination can be a desirable property that one would like to allow or even require in some circumstances. Even in the non-reactive case, one would like to be able to specify heuristic search problems, where if the program terminates, it returns a valid answer to a query, but the program is not guaranteed to terminate for all queries. It is not possible to specify such programs using designs.

An important distinction between nontermination and abortion is that, while the former may sometimes be a desirable property, the latter never is, so while a specification may sometimes tolerate abortion, it should never demand it. Even the specification $\textbf{abort}_{\mathit{UTP}}$, while everywhere admitting abortion, does not actually demand it. So while some of our new constructs in this paper allow a specification sometimes to demand nontermination, none of them provides a means of demanding abortion.

To allow such specifications to be expressed, we extend the model to allow nontermination and a program aborting to be distinguished. To do this, we use the boolean observation term' to model termination, and the boolean observation ok' to model the program not aborting. Informally $\mathit{ok}' \wedge \mathit{term}'$ in this model is equivalent to okay' for the UTP design model. As with the UTP design model, we also have the observations ok, indicating that the program starts in a stable state (i.e., the preceding program has not aborted), and term, indicating that the program starts at some finite time, (i.e., the

[4] This is known in the Z and Object-Z literature as the *blocking* interpretation [16].

preceding program terminated). We introduce a new form of design, an *extended design* $(p \vdash_X r)$, that if p holds initially, guarantees to deliver a post-state satisfying r with respect to the pre-state. The syntax of an extended design uses "$\vdash_X$" to distinguish it from a design, which uses just "$\vdash$". The assumption p holding initially does not guarantee termination, but r may explicitly refer to $term'$ to require termination. For example, for a relation w that does not refer to $term'$,

– $(p \vdash_X term' \wedge w)$ requires that if p holds initially, the program should terminate and satisfy w between its pre-state and post-state;
– $(p \vdash_X term' \Rightarrow w)$ requires that if both p holds initially and the program terminates, then it also satisfies w;
– $(p \vdash_X (q \Rightarrow term' \wedge w) \wedge (\neg\, q \Rightarrow \neg\, term'))$, where q is a single-state predicate on the pre-state, requires that if p holds initially, then both the following conditions hold: if q holds initially, the program should terminate and satisfy w; and if q does not hold initially, the program never terminates.

For (non)termination there are three possibilities for each initial state: termination is required, nontermination is required, or either termination or nontermination is possible.

The semantics of the extended design $(p \vdash_X r)$ as a relation is characterised by the following predicate:

$$(ok \wedge term \wedge p \Rightarrow ok' \wedge r) \wedge (\neg\, term' \Rightarrow ok') \wedge (term' \Rightarrow term)\ . \tag{4}$$

It says that if the program starts in a stable state at some finite time (i.e., $ok \wedge term$) in an initial state in which p holds, then the program will remain in a stable state (i.e., ok') and relation r will hold between the initial and final states. A nonterminating program is *ipso facto* non-aborting and therefore stable ($\neg\, term' \Rightarrow ok'$), and a program can only terminate if its predecessor terminated ($term' \Rightarrow term$). For an extended design, the single-state predicate p should not refer to ok or $term$, and the relation r may refer to $term'$ but not ok, ok' or $term$.[5] The relation $r(v, v', term')$ is in terms of the initial values of the program variables v, their final values v', and the final termination observation $term'$.

An extended design of the form

$$(\mathrm{true} \vdash_X \neg\, term' \wedge x' = 1) \tag{5}$$

is not sensible because it constrains the final value of x to be one, even though it is guaranteed to never terminate. Because one can never observe the final values of the program variables in the case of nontermination, for an extended design to be well formed, we require that r is such that it does not constrain the final values of the program variables in the case of nontermination, i.e.,

$$[p \wedge \neg\, term' \Rightarrow (r \Leftrightarrow (\forall\, v' \bullet r))]\ , \tag{6}$$

[5] Again, allowing p and r to refer to ok, ok', and $term$ doesn't add anything because $(p \vdash_X r)$ is semantically equivalent to $(p \left[\frac{\mathrm{true},\mathrm{true}}{ok,term}\right] \vdash_X r \left[\frac{\mathrm{true},\mathrm{true},\mathrm{true}}{ok,term,ok'}\right])$.

where v' is the set of final-state program variables. Note that the program variables (v) do not include the observations *ok* and *term*. The example (5) does not satisfy this requirement because the following does not hold for all values of $term'$ and x' :

$$\begin{aligned} &[\neg\, term' \Rightarrow (\neg\, term' \wedge x' = 1 \Leftrightarrow (\forall x' \bullet \neg\, term' \wedge x' = 1))] \\ \equiv\; &[\neg\, term' \Rightarrow (x' = 1 \Leftrightarrow \text{false})] \\ \equiv\; &[\neg\, term' \Rightarrow (x' \neq 1)]\ . \end{aligned}$$

An extended design $(p_0 \vdash_X r_0)$ *is refined by* another extended design $(p_1 \vdash_X r_1)$ provided the semantic relation defined by the latter implies the semantic relation defined by the former, that is,

$$\begin{aligned} &[(ok \wedge term \wedge p_1 \Rightarrow ok' \wedge r_1) \wedge (\neg\, term' \Rightarrow ok') \wedge (term' \Rightarrow term) \Rightarrow \\ &\ (ok \wedge term \wedge p_0 \Rightarrow ok' \wedge r_0) \wedge (\neg\, term' \Rightarrow ok') \wedge (term' \Rightarrow term)]\ , \end{aligned}$$

which holds if and only if

$$[p_0 \Rightarrow (p_1 \wedge (r_1 \Rightarrow r_0))]\ . \tag{7}$$

This is similar to the condition for refining UTP designs (3), except that r_0 and r_1 may refer to $term'$ to specify termination behaviour.

Relating designs and extended designs. To see that an extended design generalises a design, we show that we can map any design into a unique extended design. For any design, $(p \vdash w)$, we have

$$DX(p \vdash w) \mathrel{\widehat=} (p \vdash_X term' \wedge w)\ .$$

It is straightforward to show that this mapping preserves the refinement ordering, and that the image of this mapping is a subtheory of extended designs [1, Chap. 4].

For extended designs, we have the following interesting extreme cases:

$$\begin{aligned} \mathbf{abort}_X &\mathrel{\widehat=} (\text{false} \vdash_X \text{true}) \\ \mathbf{chaos}_X &\mathrel{\widehat=} (\text{true} \vdash_X \text{true}) \\ \mathbf{terminates}_X &\mathrel{\widehat=} (\text{true} \vdash_X term') \\ \mathbf{forever}_X &\mathrel{\widehat=} (\text{true} \vdash_X \neg\, term') \\ \mathbf{magic}_X &\mathrel{\widehat=} (\text{true} \vdash_X \text{false}) \end{aligned}$$

where $\mathbf{abort}_X$, $\mathbf{terminates}_X$, and $\mathbf{magic}_X$ correspond to their UTP design equivalents (via the mapping DX). The other two commands do not have equivalent UTP designs: $\mathbf{chaos}_X$ does not abort but it may or may not terminate, and if it terminates then any final state is possible; and $\mathbf{forever}_X$ does not abort but also never terminates. The extended design $\mathbf{chaos}_X$ is refined by both $\mathbf{terminates}_X$ and $\mathbf{forever}_X$. Extended designs form a complete lattice under the refinement ordering, with least element $\mathbf{abort}_X$ and greatest element $\mathbf{magic}_X$.

Total and partial correctness. To show that a program s is totally correct with respect to the precondition p (interpreted as both an assumption on the initial states and a termination set) and relation w, we must show

$$(p \vdash_X \mathit{term}' \land w) \sqsubseteq s$$

and to show partial correctness with respect to the same precondition (this time interpreted as just an assumption on the initial state) and relation, we must show

$$(p \vdash_X \mathit{term}' \Rightarrow w) \sqsubseteq s \ .$$

5 General Correctness

Parnas [19,20] introduced the notion of a limited domain (LD) relation to describe termination sets and pre-post relations (of a control structure that generalised Dijkstra's guarded command control structures [21]). Jacobs and Gries [22] introduced a similar idea called *general correctness*, which has been further explored by Nelson [23] and Dijkstra and Scholten [24]. Dunne has studied general correctness [25,26] and incorporated general correctness into a UTP setting [27]. He makes use of a *prescription* of the form $(p \Vdash w)$, which is guaranteed to terminate from initial states in which p holds, and if it does terminate (whether or not it was guaranteed to do so) then relation w holds on termination. Note that the syntax of a prescription uses a "$\Vdash$" in place of a "$\vdash$" to distinguish it. We can model the semantics of the prescription $(p \Vdash w)$ as a relation by making use of the observation *term*, which represents termination[6]:

$$(\mathit{term} \land p \Rightarrow \mathit{term}') \land (\mathit{term}' \Rightarrow w \land \mathit{term}) \ . \tag{8}$$

The following examples illustrate the expressive versatility of prescriptions:

- $(\text{true} \Vdash w)$ guarantees termination from any state and that w holds;
- $(p \Vdash p \Rightarrow w)$ requires that in any initial state in which p holds, the program terminates and satisfies w, and if p does not hold initially, there is no guarantee of termination and no guarantee about the final state (although it never aborts — see Section 6 for further explanation);
- $(\text{false} \Vdash w)$ corresponds to a partial correctness specification — although no guarantee of termination is given, if it does terminate, w holds; and
- $(\text{false} \Vdash \text{false})$ guarantees to never terminate.

A prescription $(p_0 \Vdash w_0)$ *is refined by* another prescription $(p_1 \Vdash w_1)$ provided the semantic relation defined by the latter implies (is included in) the semantic relation defined by the former, that is,

$$[(\mathit{term} \land p_1 \Rightarrow \mathit{term}') \land (\mathit{term}' \Rightarrow w_1 \land \mathit{term}) \Rightarrow$$
$$(\mathit{term} \land p_0 \Rightarrow \mathit{term}') \land (\mathit{term}' \Rightarrow w_0 \land \mathit{term})] \ ,$$

which holds if and only if $[p_0 \Rightarrow p_1] \land [w_1 \Rightarrow w_0]$.

[6] We use the observation name "*term*" to be consistent with the terminology in the rest of this paper, although the name "*ok*" is used by Dunne [27].

Relating prescriptions and extended designs. To see that an extended design generalises a prescription, we show that we can map any prescription into a unique extended design. For any prescription $(p \Vdash w)$ we have

$$PX(p \Vdash w) \mathrel{\widehat{=}} (\text{true} \vdash_X (p \Rightarrow \mathit{term}') \wedge (\mathit{term}' \Rightarrow w)) \ .$$

It is straightforward to show that this mapping preserves the refinement ordering, and that the image of this mapping is a subtheory of extended designs.

For prescriptions we have the following interesting extreme cases:

$$\begin{aligned} \mathbf{chaos}_P &\mathrel{\widehat{=}} (\text{false} \Vdash \text{true}) \\ \mathbf{terminates}_P &\mathrel{\widehat{=}} (\text{true} \Vdash \text{true}) \\ \mathbf{forever}_P &\mathrel{\widehat{=}} (\text{false} \Vdash \text{false}) \\ \mathbf{magic}_P &\mathrel{\widehat{=}} (\text{true} \Vdash \text{false}) \end{aligned}$$

where these all correspond to their extended design equivalents, but note that there is no equivalent of $\mathbf{abort}_X$. Prescriptions form a complete lattice under the refinement ordering, with least element $\mathbf{chaos}_P$ and greatest element $\mathbf{magic}_P$. We expand on the distinction between general correctness and extended designs in the next section.

6 Assumptions on the Initial State versus Termination Sets

In both the UTP design, $(p \vdash w)$, and the extended design, $(p \vdash_X r)$, the predicate p acts as an assumption the implementor can make about the initial state. If p doesn't hold initially then the implementation is free to do anything, even abort. The UTP design has the requirement that the program must also terminate whenever p holds initially. Hence for a UTP design, p is both an assumption on the initial state and a termination set. For extended designs, p is only an assumption on the initial state. Modulo p, the termination set is specified within r. This is because any behaviour is allowable if p does not hold initially, so termination is only guaranteed from those initial states where both p holds and r requires termination.

In the general correctness prescription $(p \Vdash w)$, the predicate p specifies the termination set. There is no way to specify an assumption on the initial state (in the above sense) in general correctness, because general correctness has no notion of abortion. One can get close with a prescription of the form $(p \Vdash q \Rightarrow w)$, where q is a single-state predicate on the initial state. If q does not hold initially, then any non-aborting behaviour is allowed. However, this has a subtle difference in behaviour when prescriptions are sequentially composed. In all our models, sequential composition is defined as the relational composition of the semantic relations of the two commands. For general correctness we have that

$$(p \Vdash \text{true});\ (\text{true} \Vdash x' = 1) \tag{9}$$

guarantees that, even if p does not hold initially, if the first prescription terminates, then the whole terminates and the final value of x will be one. Hence, if the whole terminates, then x is guaranteed to be one. If we replace the prescriptions in (9) with UTP designs or extended designs of the same form, no such guarantee about the final value of x is given if p does not hold initially. With the UTP design $(p \vdash \text{true})$, if p doesn't hold initially, then its semantic relation allows $okay'$ to be false, in which case $okay$ may be false for the second command $(\text{true} \vdash x' = 1)$ and hence it can do anything, and no guarantee can be given about the final value of x. However, for the prescription $(p \Vdash \text{true})$, if p doesn't hold initially, this prescription isn't required to terminate, but if it does terminate, $(\text{true} \Vdash x' = 1)$ is then required to terminate and set x to one.

Note that for extended designs, we have the law

$$\textbf{abort}_X;\ s = \textbf{abort}_X\ ,$$

but the following law does not hold in general

$$\textbf{chaos}_X;\ s = \textbf{chaos}_X\ .$$

In summary, the implementor can rely on the assumption, p, on the initial state holding. Nothing can be assumed about an implementation, I, when it is executed from an initial state not satisfying p, and furthermore nothing can be assumed about the behaviour of any component executing after I if the execution of I happens to terminate. In contrast (non)termination represents an allowed or required behaviour of any implementation. The reason these are often confused is that both assumptions on the initial state and termination sets are defined in terms of a condition on the initial state.

7 Timed Designs

To discuss timing issues one can introduce an observation representing the current time, as done by Hehner [10,28,29] and Abadi and Lamport [30]. An extended design can be generalised to a timed design by replacing the observations *term* and *term'* by the observations τ and τ', representing the initial and final times, respectively. The two most interesting choices for representing time are the natural numbers and the non-negative reals, in both cases augmented with the value ∞ to represent nontermination. For our discussion here either representation is valid. Hehner [10,28] uses natural numbers to represent *abstract time*, that is, they represent a notion of progress rather than real time. The real-time refinement calculus [13] uses real numbers to represent real time.

A *timed design*, $(q \vdash_T r)$, has a semantics given by the following relation:

$$(ok \wedge \tau \neq \infty \wedge q \Rightarrow ok' \wedge r) \wedge (\tau' = \infty \Rightarrow ok') \wedge \tau \leq \tau'\ . \tag{10}$$

The form is similar to that for an extended design (4), except that we require that time does not go backwards, i.e., $\tau \leq \tau'$. We allow q to refer to the before-values of the program variables as well as τ and τ', and r can to refer to both the before- and after-values of the program variables as well as τ and τ', but neither q nor r can refer to ok or ok'. Because we allow it to refer to τ' the precondition q – which specifies the states in which the program is guaranteed not to abort – is no longer a condition on the initial state only, unlike the preconditions of each of our previous designs. To emphasise this we have used the name q rather than p, which we reserve for predicates on a single state.

We allow q to refer to τ' so that we may express constraints regarding *when* the program may abort. But note that q may not refer to v' because, unlike τ', the final values v' of the program variables cannot be constrained in the event of the program aborting.

The inclusion of a time variable makes it possible to express, not just if the program terminates, but when it terminates. Such execution time constraints may be included in r, e.g., $\tau' - \tau \leq 1$ requires execution to take at most one time unit. Execution time constraints may be used to define a deadline command [31], that requires that the time is at most D when the deadline command is reached, as the timed design $(\mathrm{true} \vdash_T \tau = \tau' \leq D \wedge \mathrm{id})$, where id is the identity relation on program variables. The deadline command is a specification construct; it cannot be directly implemented.

As already mentioned, as well as being used to specify termination time constraints, time may also be used to specify when the program may abort. Program abortion time constraints can be included in q. If q is taken to be false, as in the extreme program

$$\mathbf{abort}_T \mathrel{\widehat{=}} (\mathrm{false} \vdash_T \mathrm{true}) \ ,$$

we have that the program may become unstable immediately at the initial time τ. Since q is able to reference the final time, τ', it is also possible to specify that a program may abort at least t time units *after* the start time τ. For example, design

$$(\tau' - \tau < 10 \vdash_T r)$$

may either terminate within 10 time units satisfying r, or it may do anything as long as the τ' is greater than or equal to $\tau + 10$. A special case of this is the timed design

$$(\tau' - \tau < 10 \vdash_T \mathrm{false})$$

which guarantees to run for 10 time units, after which it may become unstable. It cannot terminate within 10 time units because in doing so it would incur the impossible obligation of satisfying false. This program may also be expressed as the sequential composition

$$(\mathrm{true} \vdash_T \tau' - \tau \geq 10);\ \mathbf{abort}_T$$

but note that this sequential composition could not be expressed as a single timed design if we did not allow the precondition to refer to τ'.

We consider a timed design such as $(\tau' - \tau > t \vdash_T r)$ for some non-negative time t not to be reasonable since it would put an upper bound on the time at which the program may abort. Since we would like to specify that a program that may abort at time t may be implemented by one which aborts at some later time (that is, a program that delays the occurrence of a catastrophic event), we impose a condition on the assumption q of a timed design $(q \vdash_T r)$ that $\neg\, q$ must not impose an upper bound on τ', i.e.,

$$\left[\neg\, q \Rightarrow \left(\forall\, \tau'' \bullet \tau' < \tau'' \Rightarrow \neg\, q\left[\frac{\tau''}{\tau'}\right]\right)\right] \ . \tag{11}$$

We also need a timed-design version of condition (6) ensuring that the final values of the program variables are not constrained under nontermination:

$$[\tau \neq \infty \wedge q \wedge \tau' = \infty \Rightarrow (r \Leftrightarrow (\forall\, v' \bullet r))] \ . \tag{12}$$

Refinement of timed designs,

$$(q_0 \vdash_T r_0) \sqsubseteq (q_1 \vdash_T r_1) ,$$

is defined in terms of reverse implication of the equivalent semantic relations, and hence holds provided

$$[\tau \neq \infty \wedge \tau \leq \tau' \wedge q_0 \Rrightarrow ((\tau' \neq \infty \Rrightarrow q_1) \wedge ((q_1 \Rrightarrow r_1) \Rrightarrow r_0))] . \tag{13}$$

This condition is similar to that for UTP designs (3) and extended designs (7), except that it adds the implicit precondition that the start time is finite, and the healthiness condition that no command can allow time to go backwards. The consequent is also expressed differently because q_1 may refer to the finish time τ'. In the common special case that q_1 is independent of τ', the consequent simplifies to $(q_1 \wedge (r_1 \Rrightarrow r_0))$. In the more general case, satisfaction of the antecedent $\tau \neq \infty \wedge \tau \leq \tau' \wedge q_0$ need only imply that q_1 holds when τ' is finite, since programs may not abort at time infinity, however it must always guarantee that $((q_1 \Rrightarrow r_1) \Rrightarrow r_0)$.

7.1 Relating Extended Designs and Timed Designs

Timed designs are richer than extended designs and hence we can simulate an extended design $(p \vdash_X r)$ by the timed design in which within r the observation *term'*, representing termination, is replaced by the observation that the final time is finite, i.e., $\tau' \neq \infty$. Hence we can map any extended design into a unique timed design. For any extended design $(p \vdash_X r)$ we have

$$XT(p \vdash_X r) \widehat{=} (p \vdash_T r \left[\frac{\tau' \neq \infty}{term'}\right]) .$$

It is straightforward to show that *XT* preserves the refinement ordering, and that the image of this mapping is a subtheory of timed designs.

One can define extreme cases in a similar fashion to those for extended designs, except that **terminates**$_T$ and **forever**$_T$ make use of τ' rather than *term'*. We only give the definition of these two:

$$\begin{aligned} \textbf{terminates}_T &\mathrel{\widehat{=}} (\text{true} \vdash_T \tau' \neq \infty) \\ \textbf{forever}_T &\mathrel{\widehat{=}} (\text{true} \vdash_T \tau' = \infty) . \end{aligned}$$

Timed designs form a complete lattice under the refinement ordering, with least element **abort**$_T$ and greatest element **magic**$_T$.

8 Timed Reactive Designs

To model the interactions of a real-time program with its environment, one can use a trace of the values of the program variables over time, i.e., a mapping, σ, from times to the values of the program variables at those times. As with timed designs, time can either be natural numbers or real numbers, in both cases extended with infinity. The domain of a trace, $\mathrm{dom}(\sigma)$, never includes the time ∞. A program relation then relates an initial trace, σ, to an extension of that trace σ'. For a *timed reactive design*, $(q \vdash_R r)$, both q and r are relations between the initial trace, σ, of the values of the program variables up to the start time of the command, and the final trace σ'. The start time τ is then an abbreviation for $\sup(\mathrm{dom}(\sigma))$ and the final time τ' is an abbreviation for $\sup(\mathrm{dom}(\sigma'))$, where sup stands for supremum, i.e., least upper bound. For a nonterminating computation, the domain of the final trace σ' has no finite bound, and hence $\sup(\mathrm{dom}(\sigma')) = \infty$. A timed reactive design $(q \vdash_R r)$ has a semantics given by the following relation:

$$(ok \wedge \tau \neq \infty \wedge q \Rightarrow ok' \wedge r) \wedge (\tau' = \infty \Rightarrow ok') \wedge \sigma \subseteq \sigma' \; . \tag{14}$$

The significant change from the timed design semantics (10) is that $\tau \leq \tau'$ is replaced by the stronger requirement that σ is a prefix of σ', i.e., $\sigma \subseteq \sigma'$, which implies $\sup(\mathrm{dom}(\sigma)) \leq \sup(\mathrm{dom}(\sigma'))$, i.e., $\tau \leq \tau'$. The initial state of a timed reactive design corresponds to $\sigma(\tau)$ and the final state (if there is one) to $\sigma'(\tau')$. Note that if $\tau' = \infty$, $\mathrm{dom}(\sigma')$ is the complete range of all finite times, but does not include infinity. Hence we don't need a version of condition (12) in this case.

Another change from the timed design semantics is that precondition q – which specifies the conditions under which the program is guaranteed to not abort – may refer to the final trace σ'. To illustrate, consider the following timed reactive design interpreted using abstract time (i.e., the domain of σ is natural numbers):

$$(\sigma' \neq \sigma \frown \langle x \rangle \vdash_R \text{false}) \; .$$

This may become unstable immediately after it has set the program state to the value x at time $\tau + 1$.

For the reactive timed design $(q \vdash_R r)$ we impose a condition on q that is analogous to (11) for timed designs:

$$\left[\neg\, q \Rightarrow \left(\forall \sigma'' \bullet \sigma' \subset \sigma'' \Rightarrow \neg\, q\left[\frac{\sigma''}{\sigma'}\right]\right)\right] \; . \tag{15}$$

It requires that a reactive design that may abort after behaving like trace σ', i.e., if q is false for σ', may be implemented by one that aborts at some later time, i.e., q with σ' replaced by σ'' is false for all traces σ'' that are extensions of σ'. Note that with $\tau' = \sup(\mathrm{dom}(\sigma'))$ and $\tau'' = \sup(\mathrm{dom}(\sigma''))$, (15) implies (11). The reactive design

$$(\neg\, (\exists \sigma'' \bullet x \notin \mathrm{ran}(\sigma'') \wedge \sigma' = \sigma \frown \sigma'') \vdash_R \text{false}) \; ,$$

for instance, does *not* satisfy (15), since it may abort at time τ, but it may not delay the abortion time to time $\tau + 1$ and extend the final trace with a state that takes the value x.

Refinement of timed reactive designs,

$$(q_0 \vdash_R r_0) \sqsubseteq (q_1 \vdash_R r_1) \; ,$$

is defined in terms of reverse implication of the equivalent semantic relations, and hence holds provided

$$[\tau \neq \infty \wedge \sigma \subseteq \sigma' \wedge q_0 \Rightarrow ((\tau \neq \infty \Rightarrow q_1) \wedge ((q_1 \Rightarrow r_1) \Rightarrow r_0))] \; .$$

This condition is similar to that for timed designs (13), except that the predicates now refer to the initial and final traces, σ and σ', and the healthiness constraint $\tau \leq \tau'$ is strengthened to ensure that the initial trace is a prefix of the final trace, i.e., $\sigma \subseteq \sigma'$.

For abstract time (natural numbers) the timed reactive model corresponds closely to models based on sequences of states as used in, for example, action systems [32] and TLA [33], while for real-time (real numbers) a timed reactive design corresponds closely to a *real-time specification* as used in the real-time refinement calculus [34,35,15,13]. Hoare and He [1, Chap. 8] introduce reactive processes, which consider traces of events, *tr*. Their processes satisfy the property that a process only ever extends a trace, i.e., $tr \leq tr'$, similar to our constraint on traces of states.

Relating timed designs and timed reactive designs. Timed reactive designs generalise timed designs. Each timed design $(q \vdash_T t)$ can be mapped to a unique timed reactive design.

$$TR(q \vdash_T t) \mathrel{\widehat{=}} \left(\begin{pmatrix} \exists\, \tau, \tau', v \bullet q \;\wedge \\ \quad \tau = \sup(\mathrm{dom}(\sigma)) \;\wedge \\ \quad \tau' = \sup(\mathrm{dom}(\sigma')) \;\wedge \\ \quad v = \sigma(\tau) \end{pmatrix} \vdash_R \begin{pmatrix} \exists\, \tau, \tau', v, v' \bullet t \;\wedge \\ \quad \tau = \sup(\mathrm{dom}(\sigma)) \;\wedge \\ \quad \tau' = \sup(\mathrm{dom}(\sigma')) \;\wedge \\ \quad v = \sigma(\tau) \;\wedge \\ \quad (\tau' \neq \infty \Rightarrow \\ \qquad v' = \sigma'(\tau')) \end{pmatrix} \right)$$

This mapping preserves the refinement ordering, and the image of this mapping is a subtheory of timed reactive designs.

One can define extreme cases in a similar fashion to those for extended designs and timed designs, except that $\mathbf{terminates}_R$ and $\mathbf{forever}_R$ make use of σ' rather than *term'* or τ'. We only give the definition of these two:

$$\begin{aligned} \mathbf{terminates}_R &\mathrel{\widehat{=}} (\mathrm{true} \vdash_R \sup(\mathrm{dom}(\sigma')) \neq \infty) \\ \mathbf{forever}_R &\mathrel{\widehat{=}} (\mathrm{true} \vdash_R \sup(\mathrm{dom}(\sigma')) = \infty)\ . \end{aligned}$$

Timed reactive designs form a complete lattice under the refinement ordering, with least element $\mathbf{abort}_R$ and greatest element $\mathbf{magic}_R$.

9 Conclusions

The purpose of this paper has been to help formalise the relationships between a number of different relational models of programs. These relationships are summarised graphically in Figure 1. We introduced extended designs to allow nontermination and abort to be distinguished. This allows both partial and total correctness concerns to be modelled, as well as allowing requirements like "a program must not terminate from certain initial states" to be specified. Z specifications [8,9], UTP designs [1], VDM pre-post specifications [2], refinement calculus specifications [3,4], B specifications [7], and general-correctness prescriptions [27] can be seen as special cases of extended designs. Extended designs allow the distinction between an assumption on the initial state and a termination set to be made.

One interesting consequence of formalising the relationships between these models is that it has highlighted the fact that these different approaches to specification use the word "precondition" to mean different things: it can mean an assumption, a termination set, an enabling condition (or guard or test), or combinations of these (as described in Section 1). By embedding all these approaches in the more general extended design model, we can separate out these concepts and hence determine which of them applies in each case. We hope that this better understanding of the relationships between the models and the different interpretations of the meaning of "precondition" will lead to less confusion when comparing or switching between these different models.

Extended designs can be seen as an abstraction of timed designs. In a timed design one can place specific requirements on the final time τ', whereas in an extended design one can only refer to termination (*term*$'$), which effectively abstracts all finite restrictions of τ' in a timed design simply to $\tau' \neq \infty$. Timed designs making use of abstract time are closely related to Hehner's timed models [10,28,29].

Timed reactive designs provide a richer model than timed designs, in which initial and final states are replaced by initial and final traces, σ and σ', where σ is a prefix of σ'. With abstract time this model corresponds to those used for action systems [32] and TLA [33], and with real time to the real-time refinement calculus [13,15].

In developing the real-time refinement calculus, it was observed that one needed to distinguish abort and nontermination, unlike in existing pre-post specifications in UTP designs, VDM, the refinement calculus, and B. It was in order to reconcile these models that the extended-design model was invented. It generalises the existing pre-post specification models, while simultaneously being a specialisation of both the timed and timed reactive models.

The relationship between the various models is given by the mappings between models. Each downward link in Figure 1 corresponds to a mapping from a sparser model to a richer one. In addition, one can compose these mappings to create a mapping from a model to any richer one that can be reached by following downward links. For example, one can compose the mapping *DX* from UTP designs to extended designs with the mapping *XT* from extended designs to timed designs to get a mapping $XT \circ DX$ from designs to timed designs.

Because the mappings between models embed one model as a subtheory of another, this allows properties proved in the richer model, that apply to the elements of the subtheory, to be used in the simpler model. Investigation of these uses of the mappings and extending the mappings to program constructs other than designs are avenues for future research.

Acknowledgements. This research was supported, in part, by the EPSRC-funded *Trustworthy Ambient Systems (TrAmS)* Platform Project and Australian Research Council (ARC) Discovery Grants DP0987452 and DP0879529. We would like to thank Brijesh Dongol for feedback on earlier drafts of this paper.

References

1. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall, Englewood Cliffs (1998)
2. Jones, C.B.: Systematic Software Development using VDM. Prentice-Hall, Englewood Cliffs (1986)

3. Back, R.J.R.: On correct refinement of programs. Journal of Computer and System Sciences 23(1), 49–68 (1981)
4. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, Heidelberg (1998)
5. Morgan, C.C.: The specification statement. ACM Trans. on Prog. Lang. and Sys. 10(3) (July 1988)
6. Morgan, C.C.: Programming from Specifications, 2nd edn. Prentice-Hall, Englewood Cliffs (1994)
7. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
8. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall, London (1989)
9. Hayes, I.J. (ed.): Specification Case Studies, 2nd edn. Prentice Hall, Englewood Cliffs (1993)
10. Hehner, E.C.R.: Termination is timing. In: van de Snepscheut, J.L.A. (ed.) MPC 1989. LNCS, vol. 375, pp. 36–47. Springer, Heidelberg (1989)
11. Utting, M., Fidge, C.J.: A real-time refinement calculus that changes only time. In: He, J. (ed.) Proc. 7th BCS/FACS Refinement Workshop. Electronic Workshops in Computing. Springer, Heidelberg (1996)
12. Hayes, I.J., Utting, M.: Coercing real-time refinement: A transmitter. In: Duke, D.J., Evans, A.S. (eds.) BCS-FACS Northern Formal Methods Workshop (NFMW'96). Electronic Workshops in Computing. Springer, Heidelberg (1997)
13. Hayes, I.J., Utting, M.: A sequential real-time refinement calculus. Acta Informatica 37(6), 385–448 (2001)
14. Hayes, I.J.: A predicative semantics for real-time refinement. In: McIver, A., Morgan, C.C. (eds.) Programming Methodology, pp. 109–133. Springer, Heidelberg (2003)
15. Hayes, I.J.: Reasoning about real-time repetitions: Terminating and nonterminating. Science of Computer Programming 43(2-3), 161–192 (2002)
16. Derrick, J., Boiten, E.: Refinement in Z and Object-Z. Springer, Heidelberg (2001)
17. Duke, R., Rose, G., Smith, G.: Object-Z: A specification language advocated for the description of standards. Computer Standards and Interfaces 17 (1995)
18. Smith, G.: The Object-Z Specification Language. Kluwer Academic Publishers, Dordrecht (2000)
19. Parnas, D.L.: A generalized control structure and its formal definition. Commun. ACM 26(8), 572–581 (1983)
20. Parnas, D.L., Madey, J.: Functional documents for computer systems. Sci. Comput. Program. 25(1), 41–61 (1995)
21. Dijkstra, E.W.: Guarded commands, nondeterminacy, and a formal derivation of programs. CACM 18, 453–458 (1975)
22. Jacobs, D., Gries, D.: General correctness: a unification of partial and total correctness. Acta Informatica 22, 67–83 (1985)
23. Nelson, G.: A generalisation of Dijkstra's calculus. ACM Trans. on Prog. Lang. and Sys. 11(4) (1989)
24. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer, Heidelberg (1990)
25. Dunne, S.E., Stoddart, W.J., Galloway, A.J.: Specification and refinement in general correctness. In: Evans, A., Duke, D., Clark, A. (eds.) Proceedings of the 3rd Northern Formal Methods Workshop, BCS Electronic Workshops in Computing (1998)
26. Dunne, S.E.: Abstract commands: a uniform notation for specifications and implementations. In: Fidge, C. (ed.) Computing: The Australasian Theory Symposium (CATS 2001). Electronic Notes in Theoretical Computer Science, vol. 42, pp. 104–123. Elsevier Science BV, Amsterdam (2001)

27. Dunne, S.E.: Recasting Hoare and He's unifying theory of programs in the context of general correctness. In: Butterfield, A., Strong, G., Pahl, C. (eds.) Proceedings of the 5th Irish Workshop in Formal Methods, IWFM 2001. Workshops in Computing, British Computer Society (2001)
28. Hehner, E.C.R.: Abstractions of time. In: Roscoe, A. (ed.) A Classical Mind, pp. 191–210. Prentice Hall, Englewood Cliffs (1994)
29. Hehner, E.C.R.: Retrospective and prospective for unifying theories of programming. In: Dunne, S., Stoddart, B. (eds.) UTP 2006. LNCS, vol. 4010, pp. 1–17. Springer, Heidelberg (2006)
30. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM Trans. on Prog. Lang. and Sys. 16(5), 1543–1571 (1994)
31. Fidge, C.J., Hayes, I.J., Watson, G.: The deadline command. IEE Proceedings—Software 146(2), 104–111 (1999)
32. Back, R.J., von Wright, J.: Trace refinement of action systems. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 367–384. Springer, Heidelberg (1994)
33. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison Wesley, Reading (2003)
34. Hayes, I.J.: Procedures and parameters in the real-time program refinement calculus. Science of Computer Programming 64(3), 286–311 (2007)
35. Hayes, I.J.: Termination of real-time programs: Definitely, definitely not, or maybe. In: Dunne, S., Stoddart, B. (eds.) UTP 2006. LNCS, vol. 4010, pp. 141–154. Springer, Heidelberg (2006)

Adjoint Folds and Unfolds
Or: Scything through the Thicket of Morphisms

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. Folds and unfolds are at the heart of the algebra of programming. They allow the cognoscenti to derive and manipulate programs rigorously and effectively. Fundamental laws such as fusion codify basic optimisation principles. However, most, if not all, programs require some tweaking to be given the form of an (un-) fold, and thus make them amenable to formal manipulation. In this paper, we remedy the situation by introducing adjoint folds and unfolds. We demonstrate that most programs are already of the required form and thus are directly amenable to manipulation. Central to the development is the categorical notion of an adjunction, which links adjoint (un-) folds to standard (un-) folds. We discuss a number of adjunctions and show that they are directly relevant to programming.

Keywords: initial algebra, fold, final coalgebra, unfold, adjunction.

1 Introduction

One Ring to rule them all, One Ring to find them,
One Ring to bring them all and in the darkness bind them
The Lord of the Rings—J. R. R. Tolkien.

Effective calculations are likely to be based on a few fundamental principles. The theory of initial datatypes aspires to play that rôle when it comes to calculating programs. And indeed, a single combining form and a single proof principle rule them all: programs are expressed as folds, program calculations are based on the universal property of folds. In a nutshell, the universal property formalises that a fold is the unique solution of its defining equation. It implies computation rules and optimisation rules such as fusion. The economy of reasoning is further enhanced by the principle of duality: initial algebras dualise to final coalgebras, and alongside folds dualise to unfolds. Two theories for the price of one.

However, all that glitters is not gold. Most if not all programs require some tweaking to be given the form of a fold or an unfold, and thus make them amenable to formal manipulation. Somewhat ironically, this is in particular true of the "Hello, world!" programs of functional programming: factorial, the Fibonacci function and append. For instance, append does not have the form of a fold as it takes a second argument that is later used in the base case.

We offer a solution to the problem in the form of adjoint folds and unfolds. The central idea is to gain flexibility by allowing the argument of a fold or the result of an unfold to be wrapped up in a functor application. In the case of append, the functor is essentially pairing. Not every functor is admissible though: to preserve the salient properties of folds and unfolds, we require the functor to have a right adjoint and, dually, a left adjoint for unfolds. Like folds, adjoint folds are then the unique solutions of their defining equations and, as to be expected, this dualises to unfolds. I cannot claim originality for the idea: Bird and Paterson [5] used the approach to demonstrate that their generalised folds are uniquely defined. The purpose of the present paper is to show that the idea is more profound and more far-reaching. In a sense, we turn a proof technique into a definitional principle and explore the consequences and opportunities of doing this. Specifically, the main contributions of this paper are the following:

- we introduce folds and unfolds as solutions of so-called Mendler-style equations (Mendler-style folds have been studied before [38], but we believe that they deserve to be better known);
- we argue that termination and productivity can be captured semantically using naturality;
- we show that by choosing suitable base categories mutually recursive types and parametric types are subsumed by the framework;
- we generalise Mendler-style equations to adjoint equations and demonstrate that many programs are of the required form;
- we conduct a systematic study of adjunctions and show their relevance to programming.

We largely follow a deductive approach: simple (co-) recursive programs are naturally captured as solutions of Mendler-style equations; adjoint equations generalise them in a straightforward way. Furthermore, we emphasise duality throughout by developing adjoint folds and unfolds in tandem.

Prerequisites. A basic knowledge of category theory is assumed, along the lines of the categorical trinity: categories, functors and natural transformations. I have made some effort to keep the paper sufficiently self-contained, explaining the more advanced concepts as we go along. Some knowledge of the functional programming language Haskell [32] is useful, as the formal development is paralleled by a series of programming examples.

Outline. The rest of the paper is structured as follows. Section 2 introduces some notation, serving mainly as a handy reference. Section 3 reviews conventional folds and unfolds. We take a somewhat non-standard approach and introduce them as solutions of Mendler-style equations. Section 4 generalises these equations to adjoint equations and demonstrates that many, if not most, Haskell functions fall under this umbrella. Finally, Section 5 reviews related work and Section 6 concludes.

2 Notation

We let $\mathbb{C}$, $\mathbb{D}$ etc. range over categories. By abuse of notation $\mathbb{C}$ also denotes the class of objects: we write $A \in \mathbb{C}$ to express that A is an object of $\mathbb{C}$. The class of arrows from $A \in \mathbb{C}$ to $B \in \mathbb{C}$ is denoted $\mathbb{C}(A, B)$. If $\mathbb{C}$ is obvious from the context, we abbreviate $f \in \mathbb{C}(A, B)$ by $f : A \to B$. The latter notation is used in particular for total functions (arrows in **Set**) and functors (arrows in **Cat**). Furthermore, we let A, B etc. range over objects, F, G, $\mathfrak{F}$, $\mathfrak{G}$ etc. over functors, and α, β, ϕ, Ψ etc. over natural transformations. Let $\mathsf{F}, \mathsf{G} : \mathbb{C} \to \mathbb{D}$ be two parallel functors. The class of natural transformations from F to G is denoted $\mathbb{D}^{\mathbb{C}}(\mathsf{F}, \mathsf{G})$. If $\mathbb{C}$ and $\mathbb{D}$ are obvious from the context, we abbreviate $\alpha \in \mathbb{D}^{\mathbb{C}}(\mathsf{F}, \mathsf{G})$ by $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$. We also write $\alpha : \forall A \,.\, \mathsf{F}\,A \to \mathsf{G}\,A$ and furthermore $\alpha : \forall A \,.\, \mathsf{F}\,A \cong \mathsf{G}\,A$, if α is a natural isomorphism. The inverse of an isomorphism is denoted α°.

Partial applications of functions and operators are often written using 'categorical dummies', where $-$ marks the first and $=$ the optional second argument. As an example, $- * 2$ denotes the doubling function and $- * =$ multiplication. Another example is the so-called *hom-functor* $\mathbb{C}(-, =) : \mathbb{C}^{\mathsf{op}} \times \mathbb{C} \to \mathbf{Set}$, whose action on arrows is given by $\mathbb{C}(f, g)\,h = g \cdot h \cdot f$.

The formal development is complemented by a series of Haskell programs. Unfortunately, Haskell's lexical and syntactic conventions deviate somewhat from standard mathematical practise. In Haskell, type variables start with a lower-case letter (identifiers with an initial upper-case letter are reserved for type and data constructors). Lambda expressions such as $\lambda\,x \,.\, e$ are written $\lambda x \to e$. In the Haskell code, the conventions of the language are adhered to, with one notable exception: I have taken the liberty to typeset '::' as ':' — in Haskell, '::' is used to provide a type signature, while ':' is syntax for consing an element to a list, an operator I do not use in this paper.

3 Fixed-Point Equations

To iterate is human, to recurse divine.
L. Peter Deutsch

In this section we review the semantics of datatypes and introduce folds and unfolds, albeit with a slight twist. The following two Haskell programs serve as running examples.

Haskell example 1. The datatype *Stack* models stacks of natural numbers.

$$\mathbf{data}\,\mathit{Stack} = \mathit{Empty} \mid \mathit{Push}\,(\mathit{Nat}, \mathit{Stack})$$

The type (A, B) is Haskell syntax for the cartesian product $A \times B$.

The function *total* computes the sum of a stack of natural numbers.

$$\begin{aligned} &\mathit{total} : \mathit{Stack} && \to \mathit{Nat} \\ &\mathit{total}\ \ \mathit{Empty} && = 0 \\ &\mathit{total}\ \ (\mathit{Push}\,(n, s)) && = n + \mathit{total}\,s \end{aligned}$$

The function is a typical example of a *fold*, a function that *consumes* data. □

Haskell example 2. The datatype *Sequ* captures infinite sequences of natural numbers.

$$\mathbf{data}\, \mathit{Sequ} = \mathit{Next}\,(\mathit{Nat}, \mathit{Sequ})$$

The function *from* constructs the infinite sequence of naturals, from the given argument onwards.

$$\begin{aligned}
&\mathit{from} : \mathit{Nat} \to \mathit{Sequ}\\
&\mathit{from}\quad n \quad = \mathit{Next}\,(n, \mathit{from}\,(n+1))
\end{aligned}$$

The function is a typical example of an *unfold*, a function that *produces* data. □

Both the types, *Stack* and *Sequ*, and the functions, *total* and *from*, are given by recursion equations. At the outset, it is not at all clear that these equations have solutions and if so whether the solutions are unique. It is customary to rephrase this problem as a fixed-point problem: A recursion equation of the form $x = \Psi\, x$ implicitly defines a function Ψ in the unknown x, the so-called *base function* of x. A fixed-point of the base function is then a solution of the recursion equation and vice versa.

Consider the type equation defining *Stack*. The base function, or rather, *base functor* of *Stack* is given by

$$\begin{aligned}
&\mathbf{data}\, \mathfrak{Stack}\, \mathit{stack} = \mathfrak{Empty} \mid \mathfrak{Push}\,(\mathit{Nat}, \mathit{stack})\\
&\mathbf{instance}\, \mathit{Functor}\, \mathfrak{Stack}\, \mathbf{where}\\
&\quad \mathit{fmap}\, f\, \mathfrak{Empty} \qquad\quad = \mathfrak{Empty}\\
&\quad \mathit{fmap}\, f\, (\mathfrak{Push}\,(n, s)) = \mathfrak{Push}\,(n, f\, s)\ .
\end{aligned}$$

The type argument of $\mathfrak{Stack}$ marks the recursive component.

All the functors underlying datatype declarations (sums of products) have two extremal fixed points: the *initial* F-*algebra* $\langle \mu\mathsf{F},\, \mathit{in}\rangle$ and the *final* F-*coalgebra* $\langle \nu\mathsf{F},\, \mathit{out}\rangle$, where $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ is the functor in question. (The proof that these fixed points exist is beyond the scope of this paper.) Very briefly, an F-algebra is a pair $\langle A, f\rangle$ consisting of an object $A \in \mathbb{C}$ and an arrow $f \in \mathbb{C}(\mathsf{F}\,A, A)$. Likewise, an F-coalgebra is a pair $\langle A, f\rangle$ consisting of an object $A \in \mathbb{C}$ and an arrow $f \in \mathbb{C}(A, \mathsf{F}\,A)$. (By abuse of language, we shall use the term (co-) algebra also for the components of the pair.) The objects $\mu\mathsf{F}$ and $\nu\mathsf{F}$ are the actual fixed points of the functor F: we have $\mathsf{F}\,(\mu\mathsf{F}) \cong \mu\mathsf{F}$ and $\mathsf{F}\,(\nu\mathsf{F}) \cong \nu\mathsf{F}$. The isomorphisms are witnessed by the arrows $\mathit{in} : \mathsf{F}\,(\mu\mathsf{F}) \cong \mu\mathsf{F}$ and $\mathit{out} : \nu\mathsf{F} \cong \mathsf{F}\,(\nu\mathsf{F})$.

Some languages such as Charity [7] or Coq [35] allow the user to choose between initial and final solutions — the datatype declarations are flagged as *inductive* or *coinductive*. Haskell is not one of them. Since Haskell's underlying category is $\mathbf{Cpo}_{\perp}$, the category of complete partial orders and strict continuous functions, initial algebras and final coalgebras actually coincide [16,11]. By contrast, in **Set** elements of an inductive type are finite, whereas elements of a co-inductive type are potentially infinite. Operationally, an element of an inductive type is constructed in a finite number of steps, whereas an element of a coinductive type is deconstructed in a finite number of steps.

Turning to our running examples, we view *Stack* as an initial algebra — though inductive and coinductive stacks are both equally useful. For sequences only the coinductive reading makes sense, since the initial algebra of *Sequ*'s base functor is the empty set in **Set**.

Haskell definition 3. In Haskell, initial algebras and final coalgebras can be defined as follows.

$$\begin{aligned}&\mathbf{newtype}\,\mu f = \mathit{In} \quad \{\mathit{in}^{\circ} : f\,(\mu f)\}\\ &\mathbf{newtype}\,\nu f = \mathit{Out}^{\circ}\,\{\mathit{out} : f\,(\nu f)\}\end{aligned}$$

The definitions use Haskell's record syntax to introduce the deconstructors in° and *out* in addition to the constructors *In* and Out°. The **newtype** declaration guarantees that μf and $f\,(\mu f)$ share the same representation at run-time, and likewise for νf and $f\,(\nu f)$. In other words, the constructors and deconstructors are no-ops. Of course, since initial algebras and final coalgebras coincide in Haskell, they could be defined by a single **newtype** definition. However, for emphasis we keep them separate. □

Working towards a semantics for *total*, let us first adapt its definition to the new 'two-level type' $\mu\mathfrak{Stack}$. (The term is due to Sheard [34]; one level describes the structure of the data, the other level ties the recursive knot.)

$$\begin{aligned}&\mathit{total} : \mu\mathfrak{Stack} &&\to \mathit{Nat}\\ &\mathit{total}\;\;(\mathit{In}\,\mathfrak{Empty}) &&= 0\\ &\mathit{total}\;\;(\mathit{In}\,(\mathfrak{Push}\,(n,s))) &&= n + \mathit{total}\,s\end{aligned}$$

Now, if we abstract away from the recursive call, we obtain a non-recursive base function of type $(\mu\mathfrak{Stack} \to \mathit{Nat}) \to (\mu\mathfrak{Stack} \to \mathit{Nat})$. Functions of this type possibly have many fixed points — consider as an extreme example the identity function, which has an infinite number of fixed points. Interestingly, the problem disappears into thin air, if we additionally remove the constructor *In*.

$$\begin{aligned}&\mathfrak{total} : \forall x\,.\,(x \to \mathit{Nat}) \to (\mathfrak{Stack}\,x &&\to \mathit{Nat})\\ &\mathfrak{total}\quad \mathit{total}\qquad\quad (\mathfrak{Empty}) &&= 0\\ &\mathfrak{total}\quad \mathit{total}\qquad\quad (\mathfrak{Push}\,(n,s)) &&= n + \mathit{total}\,s\end{aligned}$$

The type of the base function has become polymorphic in the argument of the recursive call. We shall show in the next section that this type guarantees that the recursive definition of *total*

$$\begin{aligned}&\mathit{total} : \mu\mathfrak{Stack} \to \mathit{Nat}\\ &\mathit{total}\;\;(\mathit{In}\,l)\;\;= \mathfrak{total}\,\mathit{total}\,l\end{aligned}$$

is well-defined and furthermore that the equation has exactly one solution.

Applying the same transformation to the type *Sequ* and the function *from* we obtain

$$\mathbf{data}\,\mathfrak{Sequ}\,\mathit{sequ} = \mathfrak{Next}\,(\mathit{Nat}, \mathit{sequ})$$

$$\begin{aligned}&\mathfrak{from} : \forall x\,.\,(\mathit{Nat} \to x) \to (\mathit{Nat} \to \mathfrak{Sequ}\,x)\\ &\mathfrak{from}\quad \mathit{from}\qquad\quad n\quad = \mathfrak{Next}\,(n, \mathit{from}\,(n+1))\end{aligned}$$

$$\begin{aligned}&\mathit{from} : \mathit{Nat} \to \nu\mathfrak{Sequ}\\ &\mathit{from}\;\;n\;\;= \mathit{Out}^{\circ}\,(\mathfrak{from}\,\mathit{from}\,n)\;\;.\end{aligned}$$

Again, the base function enjoys a polymorphic type that guarantees that the recursive function is well-defined.

Abstracting away from the particulars of the syntax, the examples suggest to consider fixed-point equations of the form

$$x \cdot \mathit{in} = \Psi\, x, \qquad \text{and dually} \qquad \mathit{out} \cdot x = \Psi\, x \ \ , \tag{1}$$

where the unknown x has type $\mathbb{C}(\mu\mathsf{F}, A)$ on the left and $\mathbb{C}(A, \nu\mathsf{F})$ on the right. Arrows defined by equations of this form are known as *Mendler-style folds and unfolds* [28]. We shall henceforth drop the qualifier and call the solutions simply folds and unfolds. In fact, the abuse of language is justified as each Mendler-style equation is equivalent to the defining equation of an (un-) fold. This is what we show next, considering folds first.

3.1 Initial Fixed-Point Equations

Let $\mathbb{C}$ be some base category and let $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ be some endofunctor. An *initial fixed-point equation* in the unknown $x \in \mathbb{C}(\mu\mathsf{F}, A)$ has the syntactic form

$$x \cdot \mathit{in} = \Psi\, x \ \ , \tag{2}$$

where the base function Ψ has type

$$\Psi : \forall X \ . \ \mathbb{C}(X, A) \to \mathbb{C}(\mathsf{F}\, X, A) \ \ .$$

Informally speaking, the naturality of Ψ ensures *termination*: the first argument of Ψ, the recursive call of x, can only be applied to proper sub-terms of x's argument — recall that the type argument of F marks the recursive components. The naturality condition can be seen as the semantic counterpart of the *guarded-by-deconstructors* condition [15]. This becomes more visible, if we move the isomorphism $\mathit{in} : \mathsf{F}\,(\mu\mathsf{F}) \cong \mu\mathsf{F}$ to the right-hand side: $x = \Psi\, x \cdot \mathit{in}^{\circ}$. Here in° is the deconstructor that guards the recursive calls.

Termination is an operational notion; how the notion translates to a denotational setting depends on the underlying category. Our primary goal is to show that Equation 2 has a *unique solution*. When working in **Set** this result implies that the equation admits a solution that is indeed a total function. On the other hand, if the underlying category is $\mathbf{Cpo}_{\perp}$, then the solution is a continuous function that does not necessarily terminate for all its inputs, since initial algebras in $\mathbf{Cpo}_{\perp}$ possibly contain infinite elements.

While the definition of *total* fits nicely into the framework above, the following program does not.

Haskell example 4. The naturality condition is sufficient but not necessary as the example of factorial demonstrates.

$$\begin{aligned}
&\mathbf{data}\, \mathit{Nat} = Z \mid S\ \mathit{Nat}\\
&\mathit{fac} : \mathit{Nat} \ \ \to \mathit{Nat}\\
&\mathit{fac}\ \ \ Z \qquad = 1\\
&\mathit{fac}\ \ (S\, n) = S\, n * \mathit{fac}\, n
\end{aligned}$$

Like for *total*, we split the underlying datatype into two levels.

$$
\begin{aligned}
&\mathbf{type}\, \mathit{Nat} = \mu \mathfrak{Nat} \\
&\mathbf{data}\, \mathfrak{Nat}\, \mathit{nat} = \mathfrak{Z} \mid \mathfrak{S}\, \mathit{nat} \\
&\mathbf{instance}\, \mathit{Functor}\, \mathfrak{Nat}\, \mathbf{where} \\
&\quad \mathit{fmap}\, f\, \mathfrak{Z} \qquad = \mathfrak{Z} \\
&\quad \mathit{fmap}\, f\, (\mathfrak{S}\, n) = \mathfrak{S}\, (f\, n)
\end{aligned}
$$

The implementation of factorial is clearly terminating. However, the associated base function

$$
\begin{aligned}
&\mathfrak{fac} : (\mathit{Nat} \to \mathit{Nat}) \to (\mathfrak{Nat}\, \mathit{Nat} \to \mathit{Nat}) \\
&\mathfrak{fac}\;\; \mathit{fac} \qquad\qquad (\mathfrak{Z}) \qquad = 1 \\
&\mathfrak{fac}\;\; \mathit{fac} \qquad\qquad (\mathfrak{S}\, n) \quad = \mathit{In}\, (\mathfrak{S}\, n) * \mathit{fac}\, n
\end{aligned}
$$

lacks naturality. In a sense, $\mathfrak{fac}$'s type is too concrete, as it reveals that the recursive call takes a natural number. An adversary can make use of this information turning the terminating program into a non-terminating one:

$$
\begin{aligned}
&\mathfrak{bogus} : (\mathit{Nat} \to \mathit{Nat}) \to (\mathfrak{Nat}\, \mathit{Nat} \to \mathit{Nat}) \\
&\mathfrak{bogus}\;\; \mathit{fac} \qquad\qquad (\mathfrak{Z}) \qquad = 1 \\
&\mathfrak{bogus}\;\; \mathit{fac} \qquad\qquad (\mathfrak{S}\, n) \quad = n * \mathit{fac}\, (\mathit{In}\, (\mathfrak{S}\, n)) \;\; .
\end{aligned}
$$

We will get back to this example in Section 4.5. □

Turning to the proof of uniqueness, let us first spell out the naturality property underlying Ψ's type: if $h \in \mathbb{C}(X_1, X_2)$, then $\mathbb{C}(\mathsf{F}\, h, id) \cdot \Psi = \Psi \cdot \mathbb{C}(h, id)$. Recalling that $\mathbb{C}(f, g)\, h = g \cdot h \cdot f$, this unfolds to

$$\Psi\, (f \cdot h) = \Psi\, f \cdot \mathsf{F}\, h \;\; , \tag{3}$$

for all arrows $f \in \mathbb{C}(X_2, A)$. This property implies, in particular, that Ψ is completely determined by its image of id as $\Psi\, h = \Psi\, id \cdot \mathsf{F}\, h$. Moreover, the type of Ψ is isomorphic to $\mathbb{C}(\mathsf{F}\, A, A)$, the type of F-algebras.

With hindsight, we generalise the isomorphism slightly. Let $\mathsf{F} : \mathbb{D} \to \mathbb{C}$ be an arbitrary functor, then

$$\phi : \forall A\, B \,.\, \mathbb{C}(\mathsf{F}\, A, B) \cong (\forall X{:}\mathbb{D} \,.\, \mathbb{D}(X, A) \to \mathbb{C}(\mathsf{F}\, X, B)) \;\; . \tag{4}$$

Readers versed in category theory will notice that this bijection is an instance of the *Yoneda lemma*. Let $\mathsf{H} = \mathbb{C}(\mathsf{F}\, -, B)$ be the contravariant functor $\mathsf{H} : \mathbb{D}^{\mathsf{op}} \to \mathbf{Set}$ that maps an object $A \in \mathbb{D}^{\mathsf{op}}$ to the set of arrows $\mathbb{C}(\mathsf{F}\, A, B) \in \mathbf{Set}$. The Yoneda lemma states that this set is isomorphic to a set of natural transformations:

$$\forall \mathsf{H}\, A \,.\, \mathsf{H}\, A \cong (\mathbb{D}^{\mathsf{op}}(A, -) \mathrel{\dot{\to}} \mathsf{H}) \;\; ,$$

which is (4) in abstract clothing. Let us explicate the proof of (4). The functions witnessing the isomorphism are

$$\phi\, f = \lambda\, \kappa \,.\, f \cdot \mathsf{F}\, \kappa \qquad \text{and} \qquad \phi^{\circ}\, \Psi = \Psi\, id \;\; .$$

It is easy to see that ϕ° is the left-inverse of ϕ.

$$\begin{aligned}
&\phi^{\circ}\,(\phi\, f)\\
=\;&\quad \{\text{ definition of } \phi \text{ and definition of } \phi^{\circ}\ \}\\
&f \cdot \mathsf{F}\, \mathit{id}\\
=\;&\quad \{\ \mathsf{F} \text{ functor and identity }\}\\
&f
\end{aligned}$$

For the opposite direction, we have to make use of the naturality property (3). (The naturality property is the same for the more general setting.)

$$\begin{aligned}
&\phi\,(\phi^{\circ}\,\Psi)\\
=\;&\quad \{\text{ definition of } \phi^{\circ} \text{ and definition of } \phi\ \}\\
&\lambda\,\kappa\,.\,\Psi\,\mathit{id} \cdot \mathsf{F}\,\kappa\\
=\;&\quad \{\text{ naturality of } \Psi\ \}\\
&\lambda\,\kappa\,.\,\Psi\,(\mathit{id} \cdot \kappa)\\
=\;&\quad \{\text{ identity and extensionality }\}\\
&\Psi
\end{aligned}$$

We are finally in a position to prove that Equation (2) has a *unique* solution: we show that x is a solution if and only if x is a standard fold, denoted $(\!|-|\!)$.

$$\begin{aligned}
&x \cdot \mathit{in} = \Psi\, x\\
\iff\;&\quad \{\text{ isomorphism }\}\\
&x \cdot \mathit{in} = \phi\,(\phi^{\circ}\,\Psi)\, x\\
\iff\;&\quad \{\text{ definition of } \phi \text{ and definition of } \phi^{\circ}\ \}\\
&x \cdot \mathit{in} = \Psi\,\mathit{id} \cdot \mathsf{F}\, x\\
\iff\;&\quad \{\text{ initial algebras }\}\\
&x = (\!|\Psi\,\mathit{id}|\!)
\end{aligned}$$

The proof only requires that the initial F-algebra exists in $\mathbb{C}$.

3.2 Final Fixed-Point Equations

The development of the previous section dualises to final coalgebras. For reference, let us spell out the details.

A *final fixed-point equation* in the unknown $x \in \mathbb{C}(A, \nu\mathsf{F})$ has the syntactic form

$$\mathit{out} \cdot x = \Psi\, x\ , \tag{5}$$

where the base function Ψ has type

$$\Psi : \forall X\,.\,\mathbb{C}(A, X) \to \mathbb{C}(A, \mathsf{F}\, X)\ .$$

Informally speaking, the naturality of Ψ ensures *productivity*: every recursive call is guarded by a constructor. The naturality condition captures the *guarded-by-constructors* condition [15]. This can be seen more clearly, if we move the isomorphism $out : \nu\mathsf{F} \cong \mathsf{F}\,(\nu\mathsf{F})$ to the right-hand side: $x = out^{\circ} \cdot \Psi\,x$. Here out° is the constructor that guards the recursive calls.

The type of Ψ is isomorphic to $\mathbb{C}(A, \mathsf{F}\,A)$, the type of F-coalgebras. More generally, let $\mathsf{F} : \mathbb{D} \to \mathbb{C}$, then

$$\phi : \forall A\,B\;.\;\mathbb{C}(A, \mathsf{F}\,B) \cong (\forall X{:}\mathbb{D}\;.\;\mathbb{D}(B, X) \to \mathbb{C}(A, \mathsf{F}\,X))\;\;. \tag{6}$$

Again, this is an instance of the Yoneda lemma: now $\mathsf{H} = \mathbb{C}(A, \mathsf{F}\,-)$ is a covariant functor $\mathsf{H} : \mathbb{C} \to \mathbf{Set}$ and

$$\forall \mathsf{H}\,B\;.\;\mathsf{H}\,B \cong (\mathbb{D}(B, -) \dot{\to} \mathsf{H})\;\;.$$

Finally, the functions witnessing the isomorphism are

$$\phi\,f = \lambda\,\kappa\;.\;\mathsf{F}\,\kappa \cdot f \qquad \text{and} \qquad \phi^{\circ}\,\Psi = \Psi\,id\;\;.$$

In the following two sections we show that fixed-point equations are quite general. More functions fit under this umbrella than one might initially think.

3.3 Mutual Type Recursion: $\mathbb{C} \times \mathbb{D}$

In Haskell, datatypes can be defined by mutual recursion.

Haskell example 5. The type of multiway trees, also known as rose trees, is defined by mutual type recursion.

$$\mathbf{data}\;\mathit{Tree} = \mathit{Node}\;\mathit{Nat}\;\mathit{Trees}$$
$$\mathbf{data}\;\mathit{Trees} = \mathit{Nil} \mid \mathit{Cons}\,(\mathit{Tree}, \mathit{Trees})$$

Functions that consume a tree or a list of trees are typically defined by mutual value recursion.

$$\begin{aligned}
&\mathit{flattena} : \mathit{Tree} &&\to \mathit{Stack}\\
&\mathit{flattena}\;\;(\mathit{Node}\;n\;\mathit{ts}) &&= \mathit{Push}\,(n, \mathit{flattens}\;\mathit{ts})
\end{aligned}$$

$$\begin{aligned}
&\mathit{flattens} : \mathit{Trees} &&\to \mathit{Stack}\\
&\mathit{flattens}\;\;(\mathit{Nil}) &&= \mathit{Empty}\\
&\mathit{flattens}\;\;(\mathit{Cons}\,(t, \mathit{ts})) &&= \mathit{stack}\,(\mathit{flattena}\;t, \mathit{flattens}\;\mathit{ts})
\end{aligned}$$

The helper function *stack* defined

$$\begin{aligned}
&\mathit{stack} : (\mathit{Stack}, &&\mathit{Stack}) \to \mathit{Stack}\\
&\mathit{stack}\;\;(\mathit{Empty}, &&\mathit{bs}) \;\;= \mathit{bs}\\
&\mathit{stack}\;\;(\mathit{Push}\,(a, \mathit{as}), &&\mathit{bs}) \;\;= \mathit{Push}\,(a, \mathit{stack}\,(\mathit{as}, \mathit{bs}))
\end{aligned}$$

concatenates two stacks, see also Example 14. □

Can we fit the above definitions into the framework of the previous section? Yes, we only have to choose a suitable base category, in this case, a product category.

Given two categories $\mathbb{C}_1$ and $\mathbb{C}_2$, the *product category* $\mathbb{C}_1 \times \mathbb{C}_2$ is constructed as follows: an object of $\mathbb{C}_1 \times \mathbb{C}_2$ is a pair $\langle A_1, A_2 \rangle$ of objects $A_1 \in \mathbb{C}_1$ and $A_2 \in \mathbb{C}_2$; an arrow of $(\mathbb{C}_1 \times \mathbb{C}_2)(\langle A_1, A_2 \rangle, \langle B_1, B_2 \rangle)$ is a pair $\langle f_1, f_2 \rangle$ of arrows $f_1 \in \mathbb{C}_1(A_1, B_1)$ and $f_2 \in \mathbb{C}_2(A_2, B_2)$. Identity and composition are defined component-wise:

$$id = \langle id, id \rangle \qquad \text{and} \qquad \langle f_1, f_2 \rangle \cdot \langle g_1, g_2 \rangle = \langle f_1 \cdot g_1, f_2 \cdot g_2 \rangle \ . \tag{7}$$

The functor $\mathit{Outl} : \mathbb{C}_1 \times \mathbb{C}_2 \to \mathbb{C}_1$, which projects onto the first category, is defined by $\mathit{Outl}\,\langle A_1, A_2 \rangle = A_1$ and $\mathit{Outl}\,\langle f_1, f_2 \rangle = f_1$, and, likewise, $\mathit{Outr} : \mathbb{C}_1 \times \mathbb{C}_2 \to \mathbb{C}_2$. (As an aside, $\mathbb{C}_1 \times \mathbb{C}_2$ is the product in **Cat**.)

Returning to Example 5, the base functor underlying *Tree* and *Trees* can be seen as an endofunctor over a product category:

$$\mathsf{F}\,\langle A, B \rangle = \langle \mathit{Nat} \times B, 1 + A \times B \rangle \ .$$

The Haskell types are given by projections: $\mathit{Tree} = \mathit{Outl}\,(\mu\mathsf{F})$ and $\mathit{Trees} = \mathit{Outr}\,(\mu\mathsf{F})$. The functions *flattena* and *flattens* are handled accordingly, we bundle them to an arrow

$$\mathit{flatten} \in (\mathbb{C} \times \mathbb{C})(\mu\mathsf{F}, \langle \mathit{Stack}, \mathit{Stack} \rangle) \ ,$$

The Haskell functions are then given by projections: $\mathit{flattena} = \mathit{Outl}\,\mathit{flatten}$ and $\mathit{flattens} = \mathit{Outr}\,\mathit{flatten}$.

The following calculation makes explicit that an initial fixed-point equation in $\mathbb{C} \times \mathbb{D}$ corresponds to two equations, one in $\mathbb{C}$ and one in $\mathbb{D}$.

$$\begin{aligned}
& x \cdot in = \Psi\,x \\
\iff & \quad \{\text{ surjective pairing: } f = \langle \mathit{Outl}\,f, \mathit{Outr}\,f \rangle \ \} \\
& \langle \mathit{Outl}\,x, \mathit{Outr}\,x \rangle \cdot \langle \mathit{Outl}\,in, \mathit{Outr}\,in \rangle = \Psi\,\langle \mathit{Outl}\,x, \mathit{Outr}\,x \rangle \\
\iff & \quad \{\text{ set } x_1 = \mathit{Outl}\,x,\ x_2 = \mathit{Outr}\,x \text{ and } in_1 = \mathit{Outl}\,in,\ in_2 = \mathit{Outr}\,in \ \} \\
& \langle x_1, x_2 \rangle \cdot \langle in_1, in_2 \rangle = \Psi\,\langle x_1, x_2 \rangle \\
\iff & \quad \{\text{ definition of composition }\} \\
& \langle x_1 \cdot in_1, x_2 \cdot in_2 \rangle = \Psi\,\langle x_1, x_2 \rangle \\
\iff & \quad \{\text{ surjective pairing: } f = \langle \mathit{Outl}\,f, \mathit{Outr}\,f \rangle \ \} \\
& \langle x_1 \cdot in_1, x_2 \cdot in_2 \rangle = \langle \mathit{Outl}\,(\Psi\,\langle x_1, x_2 \rangle), \mathit{Outr}\,(\Psi\,\langle x_1, x_2 \rangle) \rangle \\
\iff & \quad \{\text{ equality of functions }\} \\
& x_1 \cdot in_1 = (\mathit{Outl} \cdot \Psi)\,\langle x_1, x_2 \rangle \quad \text{and} \quad x_2 \cdot in_2 = (\mathit{Outr} \cdot \Psi)\,\langle x_1, x_2 \rangle \\
\iff & \quad \{\text{ set } \Psi_1 = \mathit{Outl} \cdot \Psi \text{ and } \Psi_2 = \mathit{Outr} \cdot \Psi \ \} \\
& x_1 \cdot in_1 = \Psi_1\,\langle x_1, x_2 \rangle \quad \text{and} \quad x_2 \cdot in_2 = \Psi_2\,\langle x_1, x_2 \rangle
\end{aligned}$$

The base functions Ψ_1 and Ψ_2 are parametrised both with x_1 and x_2. Other than that, the syntactic form is identical to a standard fixed-point equation.

It is a simple exercise to bring the equations of Example 5 into this form.

Haskell definition 6. Mutually recursive datatypes can be modelled as follows.

$$\begin{aligned}
&\mathbf{newtype}\,\mu_1\,f_1\,f_2 = \mathit{In}_1\,\{\,in_1^\circ : f_1\,(\mu_1\,f_1\,f_2)\,(\mu_2\,f_1\,f_2)\,\} \\
&\mathbf{newtype}\,\mu_2\,f_1\,f_2 = \mathit{In}_2\,\{\,in_2^\circ : f_2\,(\mu_1\,f_1\,f_2)\,(\mu_2\,f_1\,f_2)\,\}
\end{aligned}$$

Since Haskell has no concept of pairs on the type level, that is, no product kinds, we have to curry the type constructors: $\mu_1\, f_1\, f_2 = Outl\,(\mu\langle f_1, f_2\rangle)$ and $\mu_2\, f_1\, f_2 = Outr\,(\mu\langle f_1, f_2\rangle)$. □

Haskell example 7. The base functors of *Tree* and *Trees* are

$$\mathbf{data}\,\mathfrak{Tree}\; tree\; trees = \mathfrak{Node}\; Nat\; trees$$
$$\mathbf{data}\,\mathfrak{Trees}\; tree\; trees = \mathfrak{Nil} \mid \mathfrak{Cons}\; tree\; trees\ .$$

Since all functions in Haskell live in the same category, we have to represent arrows in $\mathbb{C} \times \mathbb{C}$ by pairs of arrows in $\mathbb{C}$.

$$\begin{aligned}
&\mathfrak{flattena} : \forall x_1\, x_2\, . \\
&\qquad (x_1 \to Stack, x_2 \to Stack) \to (\mathfrak{Tree}\, x_1\, x_2 \to Stack) \\
&\mathfrak{flattena}\ \ (\mathit{flattena}, \ \mathit{flattens})\ \ (\mathfrak{Node}\, n\, ts) = Push\,(n, \mathit{flattens}\; ts) \\
&\mathfrak{flattens} : \forall x_1\, x_2\, . \\
&\qquad (x_1 \to Stack, x_2 \to Stack) \to (\mathfrak{Trees}\, x_1\, x_2 \to Stack) \\
&\mathfrak{flattens}\ \ (\mathit{flattena}, \ \mathit{flattens})\ \ (\mathfrak{Nil}) = Empty \\
&\mathfrak{flattens}\ \ (\mathit{flattena}, \ \mathit{flattens})\ \ (\mathfrak{Cons}\, t\, ts) = stack\,(\mathit{flattena}\; t, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{flattens}\; ts)
\end{aligned}$$

The definitions of *flattena* and *flattens* match exactly the scheme above.

$$\begin{aligned}
&\mathit{flattena} : \mu_1\, \mathfrak{Tree}\, \mathfrak{Trees} \to Stack \\
&\mathit{flattena}\ \ (In_1\; t) = \mathfrak{flattena}\,(\mathit{flattena}, \mathit{flattens})\; t \\
&\mathit{flattens} : \mu_2\, \mathfrak{Tree}\, \mathfrak{Trees} \to Stack \\
&\mathit{flattens}\ \ (In_2\; ts) = \mathfrak{flattens}\,(\mathit{flattena}, \mathit{flattens})\; ts
\end{aligned}$$

Since the two equations are equivalent to an initial fixed-point equation in $\mathbb{C} \times \mathbb{C}$, they indeed have unique solutions. □

No new theory is needed to deal with mutually recursive datatypes and mutually recursive functions over them.

By duality, the same is true for final coalgebras. For final fixed-point equations we have the following correspondence.

$$out \cdot x = \Psi\, x \quad \iff \quad out_1 \cdot x_1 = \Psi_1\,\langle x_1, x_2\rangle \text{ and } out_2 \cdot x_2 = \Psi_2\,\langle x_1, x_2\rangle$$

3.4 Type Functors: $\mathbb{D}^{\mathbb{C}}$

In Haskell, datatypes can be parametrised by types.

Haskell example 8. The type of perfectly balanced, binary leaf trees, perfect trees for short, is given by

$$\begin{aligned}
&\mathbf{data}\; \mathsf{Perfect}\; a = Zero\; a \mid Succ\,(\mathsf{Perfect}\,(a, a)) \\
&\mathbf{instance}\; Functor\; \mathsf{Perfect}\; \mathbf{where} \\
&\quad fmap\, f\, (Zero\; a) = Zero\,(f\ a) \\
&\quad fmap\, f\, (Succ\, p) = Succ\,(fmap\,(f \times f)\, p) \\
&(f \times g)\,(a, b) = (f\ a, g\ b)\ .
\end{aligned}$$

The type $\mathsf{Perfect}$ is a so-called *nested datatype* [4] as the type argument is changed in the recursive call. The constructors represent the height of the tree: a perfect tree of height 0 is a leaf; a perfect tree of height $n+1$ is a perfect tree of height n that contains pairs of elements.

$$\begin{aligned}
&\mathit{size} : \forall a\ .\ \mathsf{Perfect}\ a \to \mathit{Nat}\\
&\mathit{size}\quad (\mathit{Zero}\ a) = 1\\
&\mathit{size}\quad (\mathit{Succ}\ p) = 2 * \mathit{size}\ p
\end{aligned}$$

The function *size* calculates the size of a perfect tree, making good use of the balance condition. The definition requires *polymorphic recursion* [29], as the recursive call has type $\mathsf{Perfect}\,(a, a) \to \mathit{Nat}$, which is a substitution instance of the declared type. □

Can we fit the definitions above into the framework of Section 3.1? Again, the answer is yes. We only have to choose a suitable base category, this time, a functor category.

Given two categories $\mathbb{C}$ and $\mathbb{D}$, the *functor category* $\mathbb{D}^{\mathbb{C}}$ is constructed as follows: an object of $\mathbb{D}^{\mathbb{C}}$ is a functor $\mathsf{F} : \mathbb{C} \to \mathbb{D}$; an arrow of $\mathbb{D}^{\mathbb{C}}(\mathsf{F}, \mathsf{G})$ is a natural transformation $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$. (As an aside, $\mathbb{D}^{\mathbb{C}}$ is the exponential in **Cat**.)

Now, the base functor underlying $\mathsf{Perfect}$ is an endofunctor over a functor category:

$$\mathsf{F}\,P = \Lambda\,A\ .\ A + P\,(A \times A)\ .$$

Here we use Λ-notation to define a functor [14]. The second-order functor F sends a functor to a functor. Since its fixed point $\mathsf{Perfect} = \mu\mathsf{F}$ lives in a functor category, folds over perfect trees are necessarily natural transformations. The function *size* is a natural transformation, as we can assign it the type

$$\mathit{size} : \mu\mathsf{F} \dot{\to} \mathsf{K}\,\mathit{Nat}\ ,$$

where $\mathsf{K} : \mathbb{D} \to \mathbb{D}^{\mathbb{C}}$ is the constant functor $\mathsf{K}\,A = \Lambda\,B\ .\ A$. Again, we can replay the development in Haskell.

Haskell definition 9. The definition of second-order initial algebras and final coalgebras is identical to that of Definition 3, except for an additional type argument.

$$\begin{aligned}
&\mathbf{newtype}\,\mu f\ a = \mathit{In}\quad \{\mathit{in}^{\circ} : f\,(\mu f)\,a\}\\
&\mathbf{newtype}\,\nu f\ a = \mathit{Out}^{\circ}\,\{\mathit{out} : f\,(\nu f)\,a\}
\end{aligned}$$

To capture the fact that μf and νf are functors whenever f is a second-order functor, we need an extension of the Haskell 98 class system.

$$\begin{aligned}
&\mathbf{instance}\,(\forall x\ .\ (\mathit{Functor}\ x) \Rightarrow \mathit{Functor}\,(f\ x)) \Rightarrow \mathit{Functor}\,(\mu f)\,\mathbf{where}\\
&\quad \mathit{fmap}\,f\,(\mathit{In}\quad s) = \mathit{In}\quad (\mathit{fmap}\,f\ s)\\
&\mathbf{instance}\,(\forall x\ .\ (\mathit{Functor}\ x) \Rightarrow \mathit{Functor}\,(f\ x)) \Rightarrow \mathit{Functor}\,(\nu f)\,\mathbf{where}\\
&\quad \mathit{fmap}\,f\,(\mathit{Out}^{\circ}\ s) = \mathit{Out}^{\circ}\,(\mathit{fmap}\,f\ s)
\end{aligned}$$

The declarations use a so-called *polymorphic predicate* [20], which precisely captures the requirement that f sends functors to functors. Unfortunately, the extension has not been implemented yet. It can be simulated within Haskell 98 [36], but the resulting code is somewhat clumsy. $\square$

Haskell example 10. Continuing Example 8, the base functor of Perfect maps functors to functors: it has kind $(\star \to \star) \to (\star \to \star)$.

$$\mathbf{data}\ \mathfrak{Perfect}\ \mathit{perfect}\ a = \mathfrak{Zero}\ a \mid \mathfrak{Succ}\ (\mathit{perfect}\ (a, a))$$

$$\begin{aligned}
&\mathbf{instance}\ (\mathit{Functor}\ x) \Rightarrow \mathit{Functor}\ (\mathfrak{Perfect}\ x)\ \mathbf{where}\\
&\quad \mathit{fmap}\ f\ (\mathfrak{Zero}\ a) \ = \mathfrak{Zero}\ (f\ a)\\
&\quad \mathit{fmap}\ f\ (\mathfrak{Succ}\ p) = \mathfrak{Succ}\ (\mathit{fmap}\ (f \times f)\ p)
\end{aligned}$$

Accordingly, the base function of *size* is a second-order natural transformation that takes natural transformations to natural transformations.

$$\begin{array}{llll}
\mathfrak{size} : \forall x\ .\ (\forall a\ .\ x\ a \to \mathit{Nat}) \to (\forall a\ .\ \mathfrak{Perfect}\ x\ a \to \mathit{Nat}) & & & \\
\mathfrak{size} & \mathit{size} & (\mathfrak{Zero}\ a) & = 1\\
\mathfrak{size} & \mathit{size} & (\mathfrak{Succ}\ p) & = 2 * \mathit{size}\ p
\end{array}$$

$$\begin{array}{lll}
\mathit{size} : \forall a\ .\ \mu\mathfrak{Perfect}\ a \to \mathit{Nat} & & \\
\mathit{size} & (\mathit{In}\ p) & = \mathfrak{size}\ \mathit{size}\ p
\end{array}$$

The resulting equation fits the pattern of an initial fixed-point equation. Consequently, it has a unique solution. $\square$

The bottom line is that no new theory is needed to deal with parametric datatypes and polymorphic functions over them.

Table 1 summarises our findings so far.

4 Adjoint Fixed-Point Equations

⟨. . .⟩, good general theory does not search for the maximum generality, but for the right generality.

Categories for the Working Mathematician—Saunders Mac Lane

We have seen in the previous section that initial and final fixed-point equations are quite general. However, there are obviously a lot of definitions that do not fit the pattern. We have mentioned list concatenation in the introduction. Here is another example along those lines.

Haskell example 11. The function *shunt* pushes the elements of the first onto the second stack.

$$\begin{array}{lll}
\mathit{shunt} : (\mu\mathfrak{Stack}, & \mathit{Stack}) \to \mathit{Stack} & \\
\mathit{shunt}\ \ (\mathit{In}\ \mathfrak{Empty}, & \mathit{bs}) & = \mathit{bs}\\
\mathit{shunt}\ \ (\mathit{In}\ (\mathfrak{Push}\ (a, \mathit{as})), & \mathit{bs}) & = \mathit{shunt}\ (\mathit{as}, \mathit{In}\ (\mathfrak{Push}\ (a, \mathit{bs})))
\end{array}$$

The definition does not fit the pattern of an initial fixed-point equation as it takes two arguments and recurses only over the first one. $\square$

Table 1. Initial algebras and final coalgebras in different categories

category	initial fixed-point equation $x \cdot in = \Psi\, x$	final fixed-point equation $out \cdot x = \Psi\, x$
Set	inductive type standard fold	coinductive type standard unfold
Cpo	—	continuous coalgebra (domain) continuous unfold (F locally continuous in $\mathbf{Cpo}_{\perp}$)
$\mathbf{Cpo}_{\perp}$	continuous algebra (domain) strict continuous fold (F locally continuous in $\mathbf{Cpo}_{\perp}$, $\mu\mathsf{F} \cong \nu\mathsf{F}$)	continuous coalgebra (domain) strict continuous unfold (F locally continuous in $\mathbf{Cpo}_{\perp}$, $\mu\mathsf{F} \cong \nu\mathsf{F}$)
$\mathbb{C} \times \mathbb{D}$	mutually recursive inductive types mutually recursive folds	mutually recursive coinductive types mutually recursive unfolds
$\mathbb{D}^{\mathbb{C}}$	inductive type functor higher-order fold	coinductive type functor higher-order unfold

Haskell example 12. The functions *nats* and *squares* generate the sequence of natural numbers interleaved with the sequence of squares.

$$\begin{aligned} &nats : Nat \to \nu\mathfrak{Sequ} \\ &nats\ \ n \quad = Out^{\circ}\,(\mathfrak{Next}\,(n, squares\; n)) \\ &squares : Nat \to \nu\mathfrak{Sequ} \\ &squares\ \ n \quad = Out^{\circ}\,(\mathfrak{Next}\,(n * n, nats\,(n+1))) \end{aligned}$$

The two definitions are not instances of final fixed-point equations, because while the functions are mutually recursive, the datatype is not. □

In Example 11 the element of the initial algebra is embedded in a context. The central idea of this paper is to model this context by a functor, generalising fixed-point equations to

$$x \cdot \mathsf{L}\, in = \Psi\, x, \qquad \text{and dually} \qquad \mathsf{R}\, out \cdot x = \Psi\, x \ , \tag{8}$$

where the unknown x has type $\mathbb{C}(\mathsf{L}\,(\mu\mathsf{F}), A)$ on the left and $\mathbb{C}(A, \mathsf{R}\,(\nu\mathsf{F}))$ on the right. The functor L models the context of $\mu\mathsf{F}$, in the case of *shunt*, $\mathsf{L} = - \times Stack$. Dually, R allows x to return an element of $\nu\mathsf{F}$ embedded in a context. Section 4.5 discusses a suitable choice for R in Example 12. Of course, we cannot use any plain, old functors for L and R; for reasons to become clear later on, we require them to be adjoint: $\mathsf{L} \dashv \mathsf{R}$. (For a calculational introduction to adjunctions, we refer the interested reader to the paper "Adjunctions" [9].)

Let $\mathbb{C}$ and $\mathbb{D}$ be categories. The functors L and R are *adjoint*

$$\begin{tikzcd} \mathbb{C} \arrow[r, bend right, "\mathsf{R}"'] \arrow[r, phantom, "\perp"] & \mathbb{D} \arrow[l, bend right, "\mathsf{L}"'] \end{tikzcd}$$

if and only if there is a bijection

$$\phi : \forall A\, B\ .\ \mathbb{C}(\mathsf{L}\, A, B) \cong \mathbb{D}(A, \mathsf{R}\, B) \ ,$$

that is natural both in A and B. The isomorphism ϕ is called the *adjoint transposition* or *left adjunct*.

The adjoint transposition allows us to trade L in the source for R in the target of an arrow, which is the key for showing that generalised fixed-point equations (8) have unique solutions. This is what we do next.

4.1 Adjoint Initial Fixed-Point Equations

One Size Fits All
Frank Zappa and The Mothers of Invention

Let $\mathbb{C}$ and $\mathbb{D}$ be categories, let $\mathsf{L} \dashv \mathsf{R}$ be an adjoint pair of functors $\mathsf{L} : \mathbb{D} \to \mathbb{C}$ and $\mathsf{R} : \mathbb{C} \to \mathbb{D}$ and let $\mathsf{F} : \mathbb{D} \to \mathbb{D}$ be some endofunctor. An *adjoint initial fixed-point equation* in the unknown $x \in \mathbb{C}(\mathsf{L}\,(\mu\mathsf{F}), A)$ has the syntactic form

$$x \cdot \mathsf{L}\, in = \Psi\, x \ , \tag{9}$$

where the base function Ψ has type

$$\Psi : \forall X{:}\mathbb{D} \,.\, \mathbb{C}(\mathsf{L}\, X, A) \to \mathbb{C}(\mathsf{L}\,(\mathsf{F}\, X), A) \ .$$

The unique solution of (9) is called an *adjoint fold*.

The proof of uniqueness makes essential use of the fact that the adjoint transposition ϕ is natural in A: $\mathbb{D}(h, id) \cdot \phi = \phi \cdot \mathbb{C}(\mathsf{L}\, h, id)$, which translates to

$$\phi\,(f \cdot \mathsf{L}\, h) = \phi\, f \cdot h \ . \tag{10}$$

We reason as follows.

$$\begin{aligned}
& x \cdot \mathsf{L}\, in = \Psi\, x \\
\iff & \quad \{ \text{ adjunction } \} \\
& \phi\,(x \cdot \mathsf{L}\, in) = \phi\,(\Psi\, x) \\
\iff & \quad \{ \text{ naturality of } \phi \} \\
& \phi\, x \cdot in = \phi\,(\Psi\, x) \\
\iff & \quad \{ \text{ adjunction } \} \\
& \phi\, x \cdot in = (\phi \cdot \Psi \cdot \phi^{\circ})\,(\phi\, x) \\
\iff & \quad \{ \text{ Section 3.1 } \} \\
& \phi\, x = (\!|(\phi \cdot \Psi \cdot \phi^{\circ})\; id|\!) \\
\iff & \quad \{ \text{ adjunction } \} \\
& x = \phi^{\circ}\, (\!|(\phi \cdot \Psi \cdot \phi^{\circ})\; id|\!)
\end{aligned}$$

In three simple steps we have transformed the adjoint fold $x \in \mathbb{C}(\mathsf{L}\,(\mu\mathsf{F}), A)$ into the standard fold $\phi\, x \in \mathbb{D}(\mu\mathsf{F}, \mathsf{R}\, A)$ and, furthermore, the adjoint base function $\Psi : \forall X \,.\, \mathbb{C}(\mathsf{L}\, X, A) \to \mathbb{C}(\mathsf{L}\,(\mathsf{F}\, X), A)$ into the standard base function $(\phi \cdot \Psi \cdot \phi^{\circ}) : \forall X \,.\, \mathbb{D}(X, \mathsf{R}\, A) \to \mathbb{D}(\mathsf{F}\, X, \mathsf{R}\, A)$. We have shown in Section 3.1 that the resulting equation has a unique solution. The arrow $\phi\, x$ is called the *transpose* of x.

4.2 Adjoint Final Fixed-Point Equations

Buy one get one free!
A common form of sales promotion (BOGOF).

Dually, an *adjoint final fixed-point equation* in the unknown $x \in \mathbb{D}(A, \mathsf{R}\,(\nu\mathsf{F}))$ has the syntactic form

$$\mathsf{R}\, out \cdot x = \Psi\, x \ , \tag{11}$$

where the base function Ψ has type

$$\Psi : \forall X{:}\mathbb{C}\ .\ \mathbb{D}(A, \mathsf{R}\, X) \to \mathbb{D}(A, \mathsf{R}\,(\mathsf{F}\, X)) \ .$$

The unique solution of (11) is called an *adjoint unfold.*

The proof of uniqueness relies on the fact that the inverse ϕ° of the adjoint transposition is natural in B: $\mathbb{C}(id, h) \cdot \phi^{\circ} = \phi^{\circ} \cdot \mathbb{D}(id, \mathsf{R}\, h)$, that is,

$$\phi^{\circ}\,(\mathsf{R}\, h \cdot f) = h \cdot \phi^{\circ}\, f \ . \tag{12}$$

We leave it to the reader to fill in the details.

4.3 Identity: $\mathsf{Id} \dashv \mathsf{Id}$

The simplest example of an adjunction is $\mathsf{Id} \dashv \mathsf{Id}$, which shows that adjoint fixed-point equations (8) subsume fixed-point equations (1).

In the following sections we explore more interesting examples of adjunctions. Each section is structured as follows: we introduce an adjunction, specialise Equations (8) to the adjoint functors, and then provide some Haskell examples that fit the pattern.

4.4 Currying: $- \times X \dashv -^{X}$

The best-known example of an adjunction is perhaps currying. In **Set**, a function of two arguments can be treated as a function of the first argument whose values are functions of the second argument.

$$\phi : \forall A\, B\ .\ (A \times X \to B) \cong (A \to B^{X})$$

The object B^{X} is the *exponential* of X and B. In **Set**, B^{X} is the set of total functions from X to B. That this adjunction exists is one of the requirements for cartesian closure. In the case of **Set**, the isomorphisms are given by

$$\phi\, f = \lambda\, a\ .\ \lambda\, x\ .\ f\,(a, x) \qquad \text{and} \qquad \phi^{\circ}\, g = \lambda\,(a, x)\ .\ g\, a\, x \ .$$

Let us specialise the adjoint equations to $\mathsf{L} = - \times X$ and $\mathsf{R} = -^{X}$ in **Set**.

$$\begin{array}{ll}
 & x \cdot \mathsf{L}\, in = \Psi\, x \\
\iff & \{\ \text{definition of } \mathsf{L}\ \} \\
 & x \cdot (in \times id) = \Psi\, x \\
\iff & \{\ \text{pointwise}\ \} \\
 & x\,(in\ a, c) = \Psi\, x\,(a, c)
\end{array}
\qquad
\begin{array}{ll}
 & \mathsf{R}\, out \cdot x = \Psi\, x \\
\iff & \{\ \text{definition of } \mathsf{R}\ \} \\
 & (out\cdot) \cdot x = \Psi\, x \\
\iff & \{\ \text{pointwise}\ \} \\
 & out\,(x\ a\ c) = \Psi\, x\ a\ c
\end{array}$$

The adjoint fold takes two arguments, an element of an initial algebra and a second argument (often an accumulator), both of which are available on the right-hand side. The transposed fold is then a higher-order function that yields a function. Dually, a curried unfold is transformed into an uncurried unfold.

Haskell example 13. To turn the definition of *shunt* into the form of an adjoint equation, we follow the same steps as in Section 3. First, we determine the base function abstracting away from the recursive call, additionally removing *in*, and then we tie the recursive knot. The adjoint functors are $\mathsf{L} = - \times \mathit{Stack}$ and $\mathsf{R} = -^{\mathit{Stack}}$.

$$\begin{array}{llll}
\mathfrak{shunt} : \forall x\,. & & & \\
\quad (\mathsf{L}\, x \to \mathit{Stack}) \to (\mathsf{L}\,(\mathfrak{Stack}\, x) & & \to \mathit{Stack}) & \\
\mathfrak{shunt}\ \ \mathit{shunt} & (\mathfrak{Empty}, & \mathit{bs}) = \mathit{bs} & \\
\mathfrak{shunt}\ \ \mathit{shunt} & (\mathfrak{Push}\,(a, \mathit{as}), & \mathit{bs}) = \mathit{shunt}\,(\mathit{as}, \mathit{In}\,(\mathfrak{Push}\,(a, \mathit{bs}))) &
\end{array}$$

$$\begin{array}{l}
\mathit{shunt} : \mathsf{L}\,(\mu\mathfrak{Stack}) \to \mathit{Stack} \\
\mathit{shunt}\ \ (\mathit{In}\ \mathit{as}, \mathit{bs}) = \mathfrak{shunt}\ \mathit{shunt}\,(\mathit{as}, \mathit{bs})
\end{array}$$

The definition of *shunt* matches exactly the scheme for adjoint initial fixed-point equations. The transposed fold, $\phi\, \mathit{shunt}$,

$$\begin{array}{lll}
\mathit{shunt}' : \mu\mathfrak{Stack} & \to & \mathsf{R}\,\mathit{Stack} \\
\mathit{shunt}'\ \ (\mathit{In}\ \mathfrak{Empty}) & = & \lambda \mathit{bs} \to \mathit{bs} \\
\mathit{shunt}'\ \ (\mathit{In}\,(\mathfrak{Push}\,(a, \mathit{as}))) & = & \lambda \mathit{bs} \to \mathit{shunt}'\ \mathit{as}\,(\mathit{In}\,(\mathfrak{Push}\,(a, \mathit{bs})))
\end{array}$$

is the curried variant of *shunt*. □

Lists are parametric in Haskell. Can we adopt the above reasoning to parametric types and polymorphic functions?

Haskell example 14. The type of lists is given as the initial algebra of a higher-order base functor of kind $(\star \to \star) \to (\star \to \star)$.

$$\begin{array}{l}
\mathbf{data}\, \mathfrak{List}\ \mathit{list}\ a = \mathfrak{Nil} \mid \mathfrak{Cons}\,(a, \mathit{list}\ a) \\
\mathbf{instance}\ (\mathit{Functor}\ \mathit{list}) \Rightarrow \mathit{Functor}\ (\mathfrak{List}\ \mathit{list})\ \mathbf{where} \\
\quad \mathit{fmap}\ f\ \mathfrak{Nil} = \mathfrak{Nil} \\
\quad \mathit{fmap}\ f\ (\mathfrak{Cons}\,(a, \mathit{as})) = \mathfrak{Cons}\,(f\ a, \mathit{fmap}\ f\ \mathit{as})
\end{array}$$

Lists generalise stacks, sequences of natural numbers, to an arbitrary element type. The function *append* concatenates two lists.

$$\begin{array}{llll}
\mathit{append} : \forall a\,. & (\mu\mathfrak{List}\, a, & \mathsf{List}\, a) & \to \mathsf{List}\, a \\
\mathit{append} & (\mathit{In}\, \mathfrak{Nil}, & \mathit{bs}) & = \mathit{bs} \\
\mathit{append} & (\mathit{In}\,(\mathfrak{Cons}\,(a, \mathit{as})), & \mathit{bs}) & = \mathit{In}\,(\mathfrak{Cons}\,(a, \mathit{append}\,(\mathit{as}, \mathit{bs})))
\end{array}$$

Concatenation generalises the function *stack* (see Example 5) to sequences of an arbitrary element type. □

If we lift products pointwise to functors, $(\mathsf{F} \mathbin{\dot{\times}} \mathsf{G})\, A = \mathsf{F}\, A \times \mathsf{G}\, A$, we can view *append* as a natural transformation:

$$\mathit{append} : \mathsf{List} \mathbin{\dot{\times}} \mathsf{List} \mathbin{\dot{\to}} \mathsf{List}\ .$$

All that is left to do is to find the right adjoint of the lifted product $- \mathbin{\dot\times} \mathsf{H}$. (One could be led to think that $\mathsf{F} \mathbin{\dot\times} \mathsf{H} \mathbin{\dot\to} \mathsf{G} \cong \mathsf{F} \mathbin{\dot\to} (\mathsf{H} \mathbin{\dot\to} \mathsf{G})$, but this does not make any sense as $\mathsf{H} \mathbin{\dot\to} \mathsf{G}$ is not a functor. Also, lifting exponentials pointwise $\mathsf{G}^{\mathsf{H}}\, A = (\mathsf{G}\, A)^{\mathsf{H}\, A}$ does not work, because the data does not define a functor as the exponential is contravariant in its first argument.) For simplicity, let us assume that the functor category is $\mathbf{Set}^{\mathbb{C}}$ so that $\mathsf{G}^{\mathsf{H}} : \mathbb{C} \to \mathbf{Set}$. We reason as follows:

$$
\begin{aligned}
& \mathsf{G}^{\mathsf{H}}\, A \\
\cong\ & \quad \{\ \text{Yoneda lemma}\ \} \\
& \mathbb{C}(A, -) \mathbin{\dot\to} \mathsf{G}^{\mathsf{H}} \\
\cong\ & \quad \{\ \text{requirement: } - \mathbin{\dot\times} \mathsf{H} \dashv -^{\mathsf{H}}\ \} \\
& \mathbb{C}(A, -) \mathbin{\dot\times} \mathsf{H} \mathbin{\dot\to} \mathsf{G} \\
\cong\ & \quad \{\ \text{natural transformation}\ \} \\
& \forall X : \mathbb{C} \,.\, \mathbb{C}(A, X) \times \mathsf{H}\, X \to \mathsf{G}\, X \\
\cong\ & \quad \{\ - \times X \dashv -^{X}\ \} \\
& \forall X : \mathbb{C} \,.\, \mathbb{C}(A, X) \to (\mathsf{G}\, X)^{\mathsf{H}\, X} \quad .
\end{aligned}
$$

If we set $\mathsf{G}^{\mathsf{H}}\, A = \forall X : \mathbb{C} \,.\, \mathbb{C}(A, X) \to (\mathsf{G}\, X)^{\mathsf{H}\, X}$ and $\mathsf{G}^{\mathsf{H}}\, f = \Lambda X \,.\, \mathbb{C}(f, id) \to id$, then $- \mathbin{\dot\times} \mathsf{H} \dashv -^{\mathsf{H}}$.

Haskell definition 15. The definition of exponentials goes beyond Haskell 98, as it requires rank-2 types (the data constructor *Exp* has a rank-2 type).

$$
\begin{aligned}
& \mathbf{newtype}\ \mathsf{Exp}\ h\ g\ a = \mathit{Exp}\ \{\ \mathit{exp}^{\circ} : \forall x \,.\, (a \to x) \to (h\, x \to g\, x)\ \} \\
& \mathbf{instance}\ \mathit{Functor}\ (\mathsf{Exp}\ h\ g)\ \mathbf{where} \\
& \quad \mathit{fmap}\ f\ (\mathit{Exp}\ h) = \mathit{Exp}\ (\lambda\kappa \to h\ (\kappa \cdot f))
\end{aligned}
$$

Morally, h and g are functors, as well. However, their mapping functions are not needed to define the $\mathsf{Exp}\, h\, g$ instance of *Functor*. The transpositions are defined

$$
\begin{aligned}
& \phi_{\mathsf{Exp}} : (\mathit{Functor}\ f) \Rightarrow (\forall x \,.\, (f\ x, h\ x) \to g\ x) \to (\forall x \,.\, f\ x \to \mathsf{Exp}\ h\ g\ x) \\
& \phi_{\mathsf{Exp}}\ \sigma = \lambda s \to \mathit{Exp}\ (\lambda\kappa \to \lambda t \to \sigma\ (\mathit{fmap}\ \kappa\ s, t)) \\
& \phi^{\circ}_{\mathsf{Exp}} : (\forall x \,.\, f\ x \to \mathsf{Exp}\ h\ g\ x) \to (\forall x \,.\, (f\ x, h\ x) \to g\ x) \\
& \phi^{\circ}_{\mathsf{Exp}}\ \tau = \lambda(s, t) \to \mathit{exp}^{\circ}\ (\tau\ s)\ \mathit{id}\ t \quad .
\end{aligned}
$$

Again, most of the functor instances are not needed. □

Haskell example 16. Returning to Example 14, we may conclude that the defining equation of *append* has a unique solution. Its transpose of type $\mathsf{List} \mathbin{\dot\to} \mathsf{List}^{\mathsf{List}}$ is interesting as it combines *append* with *fmap*:

$$
\begin{aligned}
& \mathit{append}' : \forall a \,.\, \mathsf{List}\ a \to \forall x \,.\, (a \to x) \to (\mathsf{List}\ x \to \mathsf{List}\ x) \\
& \mathit{append}'\ \ \mathit{as} = \lambda f \to \lambda \mathit{bs} \to \mathit{append}\ (\mathit{fmap}\ f\ \mathit{as}, \mathit{bs}) \quad .
\end{aligned}
$$

For clarity, we have inlined the definition of $\mathsf{Exp}\ \mathsf{List}\ \mathsf{List}$. □

4.5 Mutual Value Recursion: $(+) \dashv \Delta \dashv (\times)$

The functions *nats* and *squares* introduced in Example 12 are defined by mutual recursion. The program is similar to Example 5, which defines *flattena* and *flattens*, with the notable difference that only one datatype is involved, rather than a pair of mutually recursive ones. Nonetheless, the correspondence suggests to view *nats* and *squares* as a single arrow in a product category.

$$\mathit{numbers} : \langle \mathit{Nat}, \mathit{Nat} \rangle \to \Delta(\nu \mathfrak{Sequ})$$

Here $\Delta : \mathbb{C} \to \mathbb{C} \times \mathbb{C}$ is the *diagonal functor* defined by $\Delta A = \langle A, A \rangle$ and $\Delta f = \langle f, f \rangle$. According to the type, *numbers* is an adjoint unfold, provided the diagonal functor has a left adjoint. It turns out that Δ has both a left and a right adjoint. We discuss the left one first.

The left adjoint of the diagonal functor is the *coproduct*.

$$\phi : \forall A\, B \,.\, \mathbb{C}((+)\, A, B) \cong (\mathbb{C} \times \mathbb{C})(A, \Delta B)$$

Note that B is an object of $\mathbb{C}$ and A is an object of $\mathbb{C} \times \mathbb{C}$, that is, a pair of objects. Unrolling the definition of arrows in $\mathbb{C} \times \mathbb{C}$ we have

$$\phi : \forall A\, B \,.\, (A_1 + A_2 \to B) \cong (A_1 \to B) \times (A_2 \to B) \;.$$

The adjunction captures the observation that we can represent a pair of functions to the same codomain by a single function from the coproduct of the domains. The adjoint transpositions are given by

$$\phi f = \langle f \cdot \mathit{inl}, f \cdot \mathit{inr} \rangle \qquad \text{and} \qquad \phi^{\circ} \langle f_1, f_2 \rangle = f_1 \triangledown f_2 \;.$$

The reader is invited to verify that the two functions are inverses.

Using a similar reasoning as in Section 3.3, we unfold the adjoint final fixed-point equation specialised to the diagonal functor.

$$\begin{aligned}
& \Delta \mathit{out} \cdot x = \Psi\, x \\
\iff & \quad \{ \text{ definition of } \Delta \text{ } \} \\
& \langle \mathit{out}, \mathit{out} \rangle \cdot x = \Psi\, x \\
\iff & \quad \{ \text{ surjective pairing: } f = \langle \mathit{Outl}\, f, \mathit{Outr}\, f \rangle \text{ } \} \\
& \langle \mathit{out}, \mathit{out} \rangle \cdot \langle \mathit{Outl}\, x, \mathit{Outr}\, x \rangle = \Psi \langle \mathit{Outl}\, x, \mathit{Outr}\, x \rangle \\
\iff & \quad \{ \text{ set } x_1 = \mathit{Outl}\, x \text{ and } x_2 = \mathit{Outr}\, x \text{ } \} \\
& \langle \mathit{out}, \mathit{out} \rangle \cdot \langle x_1, x_2 \rangle = \Psi \langle x_1, x_2 \rangle \\
\iff & \quad \{ \text{ definition of composition } \} \\
& \langle \mathit{out} \cdot x_1, \mathit{out} \cdot x_2 \rangle = \Psi \langle x_1, x_2 \rangle \\
\iff & \quad \{ \text{ surjective pairing: } f = \langle \mathit{Outl}\, f, \mathit{Outr}\, f \rangle \text{ } \} \\
& \langle \mathit{out} \cdot x_1, \mathit{out} \cdot x_2 \rangle = \langle \mathit{Outl}\, (\Psi \langle x_1, x_2 \rangle), \mathit{Outr}\, (\Psi \langle x_1, x_2 \rangle) \rangle \\
\iff & \quad \{ \text{ equality of functions } \} \\
& \mathit{out} \cdot x_1 = (\mathit{Outl} \cdot \Psi) \langle x_1, x_2 \rangle \quad \text{and} \quad \mathit{out} \cdot x_2 = (\mathit{Outr} \cdot \Psi) \langle x_1, x_2 \rangle \\
\iff & \quad \{ \text{ set } \Psi_1 = \mathit{Outl} \cdot \Psi \text{ and } \Psi_2 = \mathit{Outr} \cdot \Psi \text{ } \} \\
& \mathit{out} \cdot x_1 = \Psi_1 \langle x_1, x_2 \rangle \quad \text{and} \quad \mathit{out} \cdot x_2 = \Psi_2 \langle x_1, x_2 \rangle
\end{aligned}$$

The resulting equations are similar to those of Section 3.3, except that now the deconstructor *out* is the same in both equations.

Haskell example 17. Continuing Haskell Example 12, the base functions of *nats* and *squares* are given by

$$\begin{aligned}
&\mathfrak{nats} : (Nat \to x, Nat \to x) \to (Nat \to \mathfrak{Sequ}\, x)\\
&\mathfrak{nats}\ \ (nats, \quad squares) \quad n \quad = \mathfrak{Next}\,(n, squares\, n)\\
&\mathfrak{squares} : (Nat \to x, Nat \to x) \to (Nat \to \mathfrak{Sequ}\, x)\\
&\mathfrak{squares}\ \ (nats, \quad squares) \quad n \quad = \mathfrak{Next}\,(n * n, nats\,(n+1))\ .
\end{aligned}$$

The recursion equations

$$\begin{aligned}
&nats : Nat \to \nu\mathfrak{Sequ}\\
&nats\ \ n \quad = Out^{\circ}\,(\mathfrak{nats}\,(nats, squares)\, n)\\
&squares : Nat \to \nu\mathfrak{Sequ}\\
&squares\ \ n \quad = Out^{\circ}\,(\mathfrak{squares}\,(nats, squares)\, n)
\end{aligned}$$

exactly fit the pattern above (if we move Out° to the left-hand side). Hence, both functions are indeed uniquely defined. Their transpose, $\phi^{\circ}\,\langle nats,\, squares\rangle$, combines the two functions into a single one using a coproduct.

$$\begin{aligned}
&numbers : \mathsf{Either}\, Nat\, Nat \to \nu\mathfrak{Sequ}\\
&numbers\ \ (Left\ \ n) \quad = Out^{\circ}\,(\mathfrak{Next}\,(n, numbers\,(Right\, n)))\\
&numbers\ \ (Right\, n) \quad = Out^{\circ}\,(\mathfrak{Next}\,(n * n, numbers\,(Left\,(n+1))))
\end{aligned}$$

The datatype Either defined $\mathbf{data}\,\mathsf{Either}\, a\, b = Left\ a \mid Right\ b$ is Haskell's coproduct. □

Turning to the dual case, to handle folds defined by mutual recursion we need the right adjoint of the diagonal functor, which is the *product.*

$$\phi : \forall A\, B\ .\ (\mathbb{C} \times \mathbb{C})(\Delta A, B) \cong \mathbb{C}(A, (\times)\, B)$$

Unrolling the definition of $\mathbb{C} \times \mathbb{C}$, we have

$$\phi : \forall A\, B\ .\ (A \to B_1) \times (A \to B_2) \cong (A \to B_1 \times B_2)\ .$$

We can represent a pair of functions with the same domain by a single function to the product of the codomains. The bijection is witnessed by

$$\phi\,\langle f_1, f_2\rangle = f_1 \vartriangle f_2 \qquad \text{and} \qquad \phi^{\circ}\, f = \langle outl \cdot f,\ outr \cdot f\rangle\ .$$

Specialising the adjoint initial fixed-point equation yields

$$x \cdot \Delta in = \Psi\, x \quad \iff \quad x_1 \cdot in = \Psi_1\,\langle x_1,\, x_2\rangle \quad \text{and} \quad x_2 \cdot in = \Psi_2\,\langle x_1,\, x_2\rangle\ .$$

If we instantiate the base function to $\Psi\, x = f \cdot \Delta(\mathsf{F}\,(\phi\, x))$ for some suitable pair of arrows f, we obtain Fokkinga's *mutomorphisms* [10]. Fokkinga observes that *paramorphisms* can be seen as a special case of mutomorphisms.

Haskell example 18. We can use mutual value recursion to fit the definition of factorial, see Example 4, into the framework. The definition of *fac* has the form of a *paramorphism* [26], as the argument that drives the recursion is not only used in the recursive call. The idea is to 'guard' the other occurrence by the identity function and to pretend that both functions are defined by mutual recursion.

$$\begin{aligned}
&\mathit{fac} : \mu\mathfrak{Nat} && \to \mathit{Nat} \\
&\mathit{fac}\ \ (\mathit{In}\,\mathfrak{Z}) && = 1 \\
&\mathit{fac}\ \ (\mathit{In}\,(\mathfrak{S}\,n)) && = \mathit{In}\,(\mathfrak{S}\,(\mathit{id}\,n)) * \mathit{fac}\,n \\
&\mathit{id} : \mu\mathfrak{Nat} && \to \mathit{Nat} \\
&\mathit{id}\ \ (\mathit{In}\,\mathfrak{Z}) && = \mathit{In}\,\mathfrak{Z} \\
&\mathit{id}\ \ (\mathit{In}\,(\mathfrak{S}\,n)) && = \mathit{In}\,(\mathfrak{S}\,(\mathit{id}\,n))
\end{aligned}$$

If we abstract away from the recursive calls, we find that the two base functions have indeed the required polymorphic types.

$$\begin{aligned}
&\mathfrak{fac} : \forall x\,.\,(x \to \mathit{Nat}, x \to \mathit{Nat}) \to (\mathfrak{Nat}\,x \to \mathit{Nat}) \\
&\mathfrak{fac}\qquad (\mathit{fac},\qquad \mathit{id})\qquad\qquad (\mathfrak{Z})\qquad = 1 \\
&\mathfrak{fac}\qquad (\mathit{fac},\qquad \mathit{id})\qquad\qquad (\mathfrak{S}\,n)\quad = \mathit{In}\,(\mathfrak{S}\,(\mathit{id}\,n)) * \mathit{fac}\,n \\
&\mathfrak{id} : \forall x\,.\,(x \to \mathit{Nat}, x \to \mathit{Nat}) \to (\mathfrak{Nat}\,x \to \mathit{Nat}) \\
&\mathfrak{id}\qquad (\mathit{fac},\qquad \mathit{id})\qquad\qquad (\mathfrak{Z})\qquad = \mathit{In}\,\mathfrak{Z} \\
&\mathfrak{id}\qquad (\mathit{fac},\qquad \mathit{id})\qquad\qquad (\mathfrak{S}\,n)\quad = \mathit{In}\,(\mathfrak{S}\,(\mathit{id}\,n))
\end{aligned}$$

The transposed fold has type $\mu\mathfrak{Nat} \to \mathit{Nat} \times \mathit{Nat}$ and corresponds to the usual encoding of paramorphisms as folds (using tupling).

As an aside, the trick does not work for the 'base function' $\mathfrak{bogus}$, as the resulting function still lacks naturality. □

Haskell example 19. Incidentally, we can employ a similar approach to also fit the Fibonacci function into the framework.

$$\begin{aligned}
&\mathit{fib} : \mathit{Nat} && \to \mathit{Nat} \\
&\mathit{fib}\ \ Z && = Z \\
&\mathit{fib}\ \ (S\,Z) && = S\,Z \\
&\mathit{fib}\ \ (S\,(S\,n)) && = \mathit{fib}\,n + \mathit{fib}\,(S\,n)
\end{aligned}$$

The definition is sometimes characterised as a *histomorphism* [37] because in the third equation *fib* depends on two previous values, rather than only one. Now, setting $\mathit{fib}'\,n = \mathit{fib}\,(S\,n)$, we can transform the nested recursion into a mutual recursion. (Indeed, this is the usual approach taken when defining the stream of Fibonacci numbers, see, for example, [19].)

$$\begin{aligned}
&\mathit{fib} : \mathit{Nat} \ \to \mathit{Nat} &\qquad\qquad& \mathit{fib}' : \mathit{Nat} \ \to \mathit{Nat} \\
&\mathit{fib}\ \ Z \quad\ = Z && \mathit{fib}'\ \ Z \quad\ = S\,Z \\
&\mathit{fib}\ \ (S\,n) = \mathit{fib}'\,n && \mathit{fib}'\ \ (S\,n) = \mathit{fib}\,n + \mathit{fib}'\,n
\end{aligned}$$

We leave the details to the reader and only remark that the transposed fold corresponds to the usual linear-time implementation of Fibonacci, called *twofib* in [2]. □

The diagram below illustrates the double adjunction $(+) \dashv \Delta \dashv (\times)$.

$$\begin{tikzcd}
\mathbb{C} \arrow[r, bend right, "\Delta"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \times \mathbb{C} \arrow[l, bend right, "+"'] \arrow[r, bend right, "\times"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, bend right, "\Delta"']
\end{tikzcd}$$

Each double adjunction actually gives rise to four different schemes and transformations: two for initial and two for final fixed-point equations. We have discussed $(+) \dashv \Delta$ for unfolds and $\Delta \dashv (\times)$ for folds. Their 'inverses' are less useful: using $(+) \dashv \Delta$ we can transform an adjoint *fold* that works on a coproduct of mutually recursive datatypes into a standard fold over a product category (see Section 3.3). Dually, $\Delta \dashv (\times)$ enables us to transform an adjoint *unfold* that yields a product of mutually recursive datatypes into a standard unfold over a product category.

4.6 Mutual Value Recursion: $\sum i \in \mathbb{I} \dashv \Delta \dashv \prod i \in \mathbb{I}$

In the previous section we have considered *two* functions defined by mutual recursion. It is straightforward to generalise the development to n mutually recursive functions (or, indeed, to an infinite number of functions). Central to the previous undertaking was the notion of a product category. Now, the product category $\mathbb{C} \times \mathbb{C}$ can be regarded as a simple functor category: $\mathbb{C}^2$, where 2 is some two-element set. To be able to deal with an arbitrary number of functions we simply generalise from 2 to an arbitrary index set.

A set forms a so-called *discrete category*: the objects are the elements of the set and the only arrows are the identities. A functor from a discrete category is uniquely defined by its action on objects. The *category of indexed objects and arrows* $\mathbb{C}^{\mathbb{I}}$, where $\mathbb{I}$ is some arbitrary index set, is a functor category from a discrete category: $A \in \mathbb{C}^{\mathbb{I}}$ if and only if $\forall i \in \mathbb{I} \,.\, A_i \in \mathbb{C}$ and $f \in \mathbb{C}^{\mathbb{I}}(A, B)$ if and only if $\forall i \in \mathbb{I} \,.\, f_i \in \mathbb{C}(A_i, B_i)$.

The diagonal functor $\Delta : \mathbb{C} \to \mathbb{C}^{\mathbb{I}}$ now sends each index to the same object: $(\Delta A)_i = A$. Left and right adjoints of the diagonal functor generalise the constructions of the previous section. The left adjoint of the diagonal functor is (a simple form of) a *dependent sum* (also called a dependent product).

$$\forall A\, B \,.\, \mathbb{C}(\textstyle\sum i \in \mathbb{I} \,.\, A_i, B) \cong \mathbb{C}^{\mathbb{I}}(A, \Delta B)$$

Its right adjoint is a *dependent product* (also called a dependent function space).

$$\forall A\, B \,.\, \mathbb{C}^{\mathbb{I}}(\Delta A, B) \cong \mathbb{C}(A, \textstyle\prod i \in \mathbb{I} \,.\, B_i)$$

The following diagram summarises the type information.

$$\begin{tikzcd}
\mathbb{C} \arrow[r, bend right, "\Delta"'] \arrow[r, phantom, "\perp"] & \mathbb{C}^{\mathbb{I}} \arrow[l, bend right, "\sum i \in \mathbb{I}"'] \arrow[r, bend right, "\prod i \in \mathbb{I}"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, bend right, "\Delta"']
\end{tikzcd}$$

It is worth singling out a special case of the construction that we shall need later on. First of all, note that

$$\mathbb{C}^{\mathbb{I}}(\Delta X, \Delta Y) \cong \mathbb{I} \to \mathbb{C}(X, Y)$$

Consequently, if the summands of the sum and the factors of the product are the same, $A = \Delta X$ and $B = \Delta Y$, we obtain another adjoint situation:

$$\forall X\ Y\ .\ \mathbb{C}(\textstyle\sum \mathbb{I}\ .\ X, Y) \cong \mathbb{I} \to \mathbb{C}(X, Y) \cong \mathbb{C}(X, \prod \mathbb{I}\ .\ Y)\ . \tag{13}$$

The degenerated sum $\sum \mathbb{I}\ .\ A$ is also called a *copower* (sometimes written $\mathbb{I} \bullet A$); the degenerated product $\prod \mathbb{I}\ .\ A$ is also called a *power* (sometimes written $A^{\mathbb{I}}$). In **Set**, we have $\sum \mathbb{I}\ .\ A = \mathbb{I} \times A$ and $\prod \mathbb{I}\ .\ A = \mathbb{I} \to A$. (Hence, $\sum \mathbb{I} \dashv \prod \mathbb{I}$ is essentially a variant of currying).

4.7 Type Application: $\mathsf{Lsh}_X \dashv (-\ X) \dashv \mathsf{Rsh}_X$

Folds of higher-order initial algebras are necessarily natural transformations, as they live in a functor category. However, many Haskell functions that recurse over a parametric datatype are actually monomorphic.

Haskell example 20. The function *sum* defined

$$\begin{array}{lll} sum : \mu\mathfrak{List}\,Nat & \to Nat \\ sum\ \ (In\,\mathfrak{Nil}) & = 0 \\ sum\ \ (In\,(\mathfrak{Cons}\,(a, as))) & = a + sum\ as \end{array}$$

sums a list of natural numbers. □

The definition of *sum* looks suspiciously like a fold, but it is not, as it does not have the right type. The corresponding function on perfect trees does not even resemble a fold.

Haskell example 21. The function *sump* sums a perfect tree of natural numbers.

$$\begin{array}{lll} sump : \mu\mathfrak{Perfect}\,Nat & \to Nat \\ sump\ \ (In\,(\mathfrak{Zero}\,n)) & = n \\ sump\ \ (In\,(\mathfrak{Succ}\,p)) & = sump\,(fmap\ plus\ p) \end{array}$$

Here, *plus* is the uncurried variant of addition: $plus\,(a, b) = a + b$. Note that the recursive call is not applied to a subterm of $\mathfrak{Succ}\,p$. In fact, it cannot, as p has type $\mathsf{Perfect}\,(Nat, Nat)$. (As an aside, this definition requires the functor instance for μ, see Definition 9.) □

Perhaps surprisingly, the definitions above fit into the framework of adjoint fixed-point equations. We simply have to view type application as a functor: given $X \in \mathbb{D}$ define $\mathsf{App}_X : \mathbb{C}^{\mathbb{D}} \to \mathbb{C}$ by $\mathsf{App}_X\,\mathsf{F} = \mathsf{F}\,X$ and $\mathsf{App}_X\,\alpha = \alpha\,X$. (The natural transformation α is applied to the object X. In Haskell this type application is invisible, which is why we cannot see that *sum* is not a standard fold.) It is easy to show that this data defines a functor: $\mathsf{App}_X\,id = id\,X = id_X$ and $\mathsf{App}_X\,(\alpha \cdot \beta) = (\alpha \cdot \beta)\,X = \alpha\,X \cdot \beta\,X = \mathsf{App}_X\,\alpha \cdot \mathsf{App}_X\,\beta$. Using App_X we can assign *sum* the type $\mathsf{App}_{Nat}\,(\mu\mathfrak{List}) \to Nat$. All that is left to do is to check whether App_X is part of an adjunction. It turns out that App_X has, in fact, both a left and a right adjoint. We choose to derive the left adjoint.

$$\begin{aligned}
& \mathbb{C}(A, \mathsf{App}_X\, B) \\
\cong\;& \{\text{ definition of } \mathsf{App}_X \} \\
& \mathbb{C}(A, B\,X) \\
\cong\;& \{\text{ Yoneda (6) }\} \\
& \forall Y{:}\mathbb{D}\,.\,\mathbb{D}(X, Y) \to \mathbb{C}(A, B\,Y) \\
\cong\;& \{\text{ definition of a copower: } \mathbb{I} \to \mathbb{C}(X, Y) \cong \mathbb{C}(\textstyle\sum \mathbb{I}\,.\,X, Y) \} \\
& \forall Y{:}\mathbb{D}\,.\,\mathbb{C}(\textstyle\sum \mathbb{D}(X, Y)\,.\,A, B\,Y) \\
\cong\;& \{\text{ define } \mathsf{Lsh}_X\, A = \Lambda\, Y{:}\mathbb{D}\,.\,\textstyle\sum \mathbb{D}(X, Y)\,.\,A \} \\
& \forall Y{:}\mathbb{D}\,.\,\mathbb{C}(\mathsf{Lsh}_X\, A\, Y, B\, Y) \\
\cong\;& \{\text{ natural transformation }\} \\
& \mathsf{Lsh}_X\, A \mathrel{\dot{\to}} B
\end{aligned}$$

We call Lsh_X the *left shift* of X, for want of a better name. Dually, the right adjoint is $\mathsf{Rsh}_X\, B = \Lambda\, Y{:}\mathbb{D}\,.\,\prod \mathbb{D}(Y, X)\,.\,B$, the *right shift* of X. The following diagram summarises the type information.

$$\begin{tikzcd}
\mathbb{C}^{\mathbb{D}} \arrow[r, bend right, "\mathsf{App}_X"'] \arrow[r, phantom, "\perp"] & \mathbb{C} \arrow[l, bend right, "\mathsf{Lsh}_X"'] \arrow[r, bend right, "\mathsf{Rsh}_X"'] \arrow[r, phantom, "\perp"] & \mathbb{C}^{\mathbb{D}} \arrow[l, bend right, "\mathsf{App}_X"']
\end{tikzcd}$$

Recall that in **Set**, the copower $\sum \mathbb{I}\,.\,A$ is the cartesian product $\mathbb{I} \times A$ and the power $\prod \mathbb{I}\,.\,A$ is the set of functions $\mathbb{I} \to A$. This correspondence suggests the Haskell implementation below. However, it is important to note that $\mathbb{I}$ is a set, not an object.

Haskell definition 22. The functors Lsh and Rsh can be defined as follows.

```
newtype Lsh_x a y = Lsh (x → y, a)
instance Functor (Lsh_x a) where
  fmap f (Lsh (κ, a)) = Lsh (f · κ, a)
newtype Rsh_x b y = Rsh { rsh° : (y → x) → b }
instance Functor (Rsh_x b) where
  fmap f (Rsh g)   = Rsh (λκ → g (κ · f))
```

The functor $\mathsf{Rsh}_x\, b$ implements a continuation type — often, but not necessarily the types x and b are identical. The transpositions are defined

$$\begin{aligned}
&\phi_{\mathsf{Lsh}} : (\forall y\,.\,\mathsf{Lsh}_x\, a\, y \to b\, y) \to (a \to b\, x) \\
&\phi_{\mathsf{Lsh}}\,\alpha = \lambda s \to \alpha\,(Lsh\,(id, s)) \\
&\phi^{\circ}_{\mathsf{Lsh}} : (Functor\ b) \Rightarrow (a \to b\, x) \to (\forall y\,.\,\mathsf{Lsh}_x\, a\, y \to b\, y) \\
&\phi^{\circ}_{\mathsf{Lsh}}\, g = \lambda(Lsh\,(\kappa, s)) \to fmap\,\kappa\,(g\, s)
\end{aligned}$$

$$\begin{aligned}
&\phi_{\mathsf{Rsh}} : (\mathit{Functor}\ a) \Rightarrow (a\ x \to b) \to (\forall y\ .\ a\ y \to \mathsf{Rsh}_x\ b\ y) \\
&\phi_{\mathsf{Rsh}}\ f = \lambda s \to \mathit{Rsh}\ (\lambda\kappa \to f\ (\mathit{fmap}\ \kappa\ s)) \\
&\phi^{\circ}_{\mathsf{Rsh}} : (\forall y\ .\ a\ y \to \mathsf{Rsh}_x\ b\ y) \to (a\ x \to b) \\
&\phi^{\circ}_{\mathsf{Rsh}}\ \beta = \lambda s \to \mathit{rsh}^{\circ}\ (\beta\ s)\ \mathit{id}\ \ .
\end{aligned}$$

The type variables x, a and b are implicitly universally quantified. □

As usual, let us specialise the adjoint equations.

$$\begin{array}{llll}
 & x \cdot \mathsf{App}_X\ \mathit{in} = \Psi\ x & & \mathsf{App}_X\ \mathit{out} \cdot x = \Psi\ x \\
\Longleftrightarrow & \{\ \text{definition of } \mathsf{App}_X\ \} & \Longleftrightarrow & \{\ \text{definition of } \mathsf{App}_X\ \} \\
 & x \cdot \mathit{in}\ X = \Psi\ x & & \mathit{out}\ X \cdot x = \Psi\ x
\end{array}$$

Since both type abstraction and type application are invisible in Haskell, adjoint equations are, in fact, indistinguishable from standard fixed-point equations.

Haskell example 23. The base function of *sump* is given by

$$\begin{array}{llll}
\mathfrak{sump} : \forall x\ .\ (\mathit{Functor}\ x) \Rightarrow & & & \\
\quad (x\ \mathit{Nat} \to \mathit{Nat}) \to (\mathfrak{Perfect}\ x\ \mathit{Nat} \to \mathit{Nat}) & & & \\
\mathfrak{sump}\ \ \mathit{sump} & (\mathfrak{Zero}\ n) & = & n \\
\mathfrak{sump}\ \ \mathit{sump} & (\mathfrak{Succ}\ p) & = & \mathit{sump}\ (\mathit{fmap}\ \mathit{plus}\ p)\ \ .
\end{array}$$

The definition requires the $\mathfrak{Perfect}$ functor instance, which in turn induces the *Functor x* context. The transpose of *sump* is a fold that returns a higher-order function.

$$\begin{array}{llll}
\mathit{sump}' : \forall x\ .\ \mathsf{Perfect}\ x \to (x \to \mathit{Nat}) \to \mathit{Nat} & & & \\
\mathit{sump}' & (\mathit{Zero}\ n) & = \lambda\kappa & \to \kappa\ n \\
\mathit{sump}' & (\mathit{Succ}\ p) & = \lambda\kappa & \to \mathit{sump}'\ p\ (\mathit{plus} \cdot (\kappa \times \kappa))
\end{array}$$

For clarity, we have inlined the definition of $\mathsf{Rsh}_{\mathit{Nat}}\ \mathit{Nat}$ and slightly optimised the result. Quite interestingly, the transformation turns a *generalised fold* in the sense of Bird and Paterson [5] into an *efficient generalised fold* in the sense of Hinze [18]. Both versions have a linear running time, but *sump′* avoids the repeated invocations of the mapping function (*fmap plus*). □

4.8 Type Composition: $\mathsf{Lan_J} \dashv (- \circ \mathsf{J}) \dashv \mathsf{Ran_J}$

Yes, we can.

Concession speech in the New Hampshire presidential primary—Barack Obama

Continuing the theme of the last section, functions over parametric types, consider the following example.

Haskell example 24. The function *concat* defined

$$\begin{array}{lll}
\mathit{concat} : \forall a\ .\ \mu\mathfrak{List}\ (\mathsf{List}\ a) & \to & \mathsf{List}\ a \\
\mathit{concat}\quad (\mathit{In}\ \mathfrak{Nil}) & = & \mathit{In}\ \mathfrak{Nil} \\
\mathit{concat}\quad (\mathit{In}\ (\mathfrak{Cons}\ (l, \mathit{ls}))) & = & \mathit{append}\ (l, \mathit{concat}\ \mathit{ls})
\end{array}$$

generalises the binary function *append* to a list of lists. □

The definition has the structure of an ordinary fold, but again the type is not quite right: we need a natural transformation of type $\mu\mathfrak{List} \mathrel{\dot\to} \mathsf{G}$, but *concat* has type $\mu\mathfrak{List} \circ \mathsf{List} \mathrel{\dot\to} \mathsf{List}$. Can we fit the definition into the framework of adjoint equations? The answer is an emphatic "Yes, we Kan!" Similar to the development of the previous section, the first step is to identify a left adjoint. To this end, we view pre-composition as a functor: $(- \circ \mathsf{List})\,(\mu\mathfrak{List}) \mathrel{\dot\to} \mathsf{List}$. (We interpret $\mathsf{List} \circ \mathsf{List}$ as $(- \circ \mathsf{List})\,\mathsf{List}$ rather than $(\mathsf{List} \circ -)\,\mathsf{List}$ because the outer list, written $\mu\mathfrak{List}$ for emphasis, drives the recursion.)

Given a functor $\mathsf{J} : \mathbb{C} \to \mathbb{D}$, define the higher-order functor $\mathsf{Pre}_\mathsf{J} : \mathbb{E}^\mathbb{D} \to \mathbb{E}^\mathbb{C}$ by $\mathsf{Pre}_\mathsf{J}\,\mathsf{F} = \mathsf{F} \circ \mathsf{J}$ and $\mathsf{Pre}_\mathsf{J}\,\alpha = \alpha \circ \mathsf{J}$. (The natural transformation α is composed with the functor J. In Haskell, type composition is invisible. Again, this is why the definition of *concat* looks like a fold, but it is not.) As usual, we should make sure that the data actually defines a functor: $\mathsf{Pre}_\mathsf{J}\,id_\mathsf{F} = id_\mathsf{F} \circ \mathsf{J} = id_{\mathsf{F}\circ\mathsf{J}}$ and $\mathsf{Pre}_\mathsf{J}\,(\alpha \cdot \beta) = (\alpha \cdot \beta) \circ \mathsf{J} = (\alpha \circ \mathsf{J}) \cdot (\beta \circ \mathsf{J}) = \mathsf{Pre}_\mathsf{J}\,\alpha \cdot \mathsf{Pre}_\mathsf{J}\,\beta$. Using the higher-order functor we can assign *concat* the type $\mathsf{Pre}_\mathsf{List}\,(\mu\mathfrak{List}) \mathrel{\dot\to} \mathsf{List}$. As a second step, we have to construct the right adjoint of the higher-order functor. Similar to the situation of the previous section, Pre_J has both a left and a right adjoint. For variety, we derive the latter.

$$\begin{aligned}
&\mathsf{F} \circ \mathsf{J} \mathrel{\dot\to} \mathsf{G}\\
\cong\quad &\{\text{ natural transformation as an end }\}\\
&\forall A{:}\mathbb{C}\,.\,\mathbb{E}(\mathsf{F}\,(\mathsf{J}\,A), \mathsf{G}\,A)\\
\cong\quad &\{\text{ Yoneda (4) }\}\\
&\forall A{:}\mathbb{C}\,.\,\forall X{:}\mathbb{D}\,.\,\mathbb{D}(X, \mathsf{J}\,A) \to \mathbb{E}(\mathsf{F}\,X, \mathsf{G}\,A)\\
\cong\quad &\{\text{ definition of power: } \mathbb{I} \to \mathbb{C}(A, B) \cong \mathbb{C}(A, \textstyle\prod \mathbb{I}\,.\,B) \}\\
&\forall A{:}\mathbb{C}\,.\,\forall X{:}\mathbb{D}\,.\,\mathbb{E}(\mathsf{F}\,X, \textstyle\prod \mathbb{D}(X, \mathsf{J}\,A)\,.\,\mathsf{G}\,A)\\
\cong\quad &\{\text{ interchange of quantifiers }\}\\
&\forall X{:}\mathbb{D}\,.\,\forall A{:}\mathbb{C}\,.\,\mathbb{E}(\mathsf{F}\,X, \textstyle\prod \mathbb{D}(X, \mathsf{J}\,A)\,.\,\mathsf{G}\,A)\\
\cong\quad &\{\text{ the functor } \mathbb{E}(\mathsf{F}\,X, -) \text{ preserves ends }\}\\
&\forall X{:}\mathbb{D}\,.\,\mathbb{E}(\mathsf{F}\,X, \forall A{:}\mathbb{C}\,.\,\textstyle\prod \mathbb{D}(X, \mathsf{J}\,A)\,.\,\mathsf{G}\,A)\\
\cong\quad &\{\text{ define } \mathsf{Ran}_\mathsf{J}\,\mathsf{G} = \Lambda\,X{:}\mathbb{D}\,.\,\forall A{:}\mathbb{C}\,.\,\textstyle\prod \mathbb{D}(X, \mathsf{J}\,A)\,.\,\mathsf{G}\,A \}\\
&\forall X{:}\mathbb{D}\,.\,\mathbb{E}(\mathsf{F}\,X, \mathsf{Ran}_\mathsf{J}\,\mathsf{G}\,X)\\
\cong\quad &\{\text{ natural transformation as an end }\}\\
&\mathsf{F} \mathrel{\dot\to} \mathsf{Ran}_\mathsf{J}\,\mathsf{G}
\end{aligned}$$

The functor $\mathsf{Ran}_\mathsf{J}\,\mathsf{G}$ is called the *right Kan extension* of G along J. (If we view $\mathsf{J} : \mathbb{C} \to \mathbb{D}$ as an inclusion functor, then $\mathsf{Ran}_\mathsf{J}\,\mathsf{G} : \mathbb{D} \to \mathbb{E}$ extends $\mathsf{G} : \mathbb{C} \to \mathbb{E}$ to the whole of $\mathbb{D}$.) Dually, the left adjoint is called the *left Kan extension* and is defined $\mathsf{Lan}_\mathsf{J}\,\mathsf{F} = \Lambda\,X{:}\mathbb{D}\,.\,\exists\,A{:}\mathbb{C}\,.\,\sum \mathbb{D}(\mathsf{J}\,A, X)\,.\,\mathsf{F}\,A$. The universally quantified object in the definition of Ran_J is a so-called *end*, which corresponds to a polymorphic type in Haskell. We refer the interested reader to Mac Lane's textbook [22] for further information. Dually, the existentially quantified object is a *coend*, which corresponds to an existential type in Haskell (hence the notation). The following diagrams summarise the type information.

$$\begin{tikzcd}
 & \mathbb{C} \arrow[dl, "\mathsf{F}"'] \arrow[d, "\mathsf{J}"] \\
\mathbb{E} & \mathbb{D} \arrow[l, shift right, "\mathsf{G}"'] \arrow[l, shift left, "\mathsf{Lan_J\,F}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathbb{E}^{\mathbb{D}} \arrow[r, shift right, "(-\circ \mathsf{J})"'] \arrow[r, phantom, "\perp"] & \mathbb{E}^{\mathbb{C}} \arrow[l, shift right, "\mathsf{Lan_J}"'] \arrow[r, shift right, "\mathsf{Ran_J}"'] \arrow[r, phantom, "\perp"] & \mathbb{E}^{\mathbb{D}} \arrow[l, shift right, "(-\circ \mathsf{J})"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathbb{C} \arrow[d, "\mathsf{J}"'] \arrow[dr, "\mathsf{G}"] & \\
\mathbb{D} \arrow[r, shift left, "\mathsf{F}"] \arrow[r, shift right, "\mathsf{Ran_J\,G}"'] & \mathbb{E}
\end{tikzcd}$$

Haskell definition 25. Like Exp, the definition of the right Kan extension requires rank-2 types (the data constructor *Ran* has a rank-2 type).

$$\textbf{newtype}\,\mathsf{Ran}_i\;g\;x = \mathit{Ran}\,\{\mathit{ran}^\circ : \forall a\;.\;(x \to i\;a) \to g\;a\}$$

$$\textbf{instance}\;\mathit{Functor}\;(\mathsf{Ran}_i\;g)\;\textbf{where}$$
$$\quad \mathit{fmap}\,f\;(\mathit{Ran}\;h) = \mathit{Ran}\;(\lambda\kappa \to h\;(\kappa\cdot f))$$

The type $\mathsf{Ran}_i\;g$ can be seen as a *generalised continuation type* — often, but not necessarily the type constructors i and g are identical. Morally, i and g are functors. However, their mapping functions are not needed to define the $\mathsf{Ran}_i\;g$ instance of *Functor*. Hence, we omit the $(\mathit{Functor}\;i, \mathit{Functor}\;g)$ context. The adjoint transpositions are defined

$$\phi_{\mathsf{Ran}} : \forall i\,f\,g\;.\;(\mathit{Functor}\,f) \Rightarrow (\forall x\;.\;f\;(i\;x) \to g\;x) \to (\forall x\;.\;f\;x \to \mathsf{Ran}_i\;g\;x)$$
$$\phi_{\mathsf{Ran}}\;\alpha = \lambda s \to \mathit{Ran}\;(\lambda\kappa \to \alpha\;(\mathit{fmap}\;\kappa\;s))$$

$$\phi^\circ_{\mathsf{Ran}} : \forall i\,f\,g\;.\;(\forall x\;.\;f\;x \to \mathsf{Ran}_i\;g\;x) \to (\forall x\;.\;f\;(i\;x) \to g\;x)$$
$$\phi^\circ_{\mathsf{Ran}}\;\beta = \lambda s \to \mathit{ran}^\circ\;(\beta\;s)\;\mathit{id}\quad.$$

Again, we omit *Functor* contexts that are not needed.

Turning to the definition of the left Kan extension we require another extension of the Haskell 98 type system: existential types.

$$\textbf{data}\,\mathsf{Lan}_i\,f\;x = \forall a\;.\;\mathit{Lan}\;(i\;a \to x, f\;a)$$

$$\textbf{instance}\;\mathit{Functor}\;(\mathsf{Lan}_i\,f)\;\textbf{where}$$
$$\quad \mathit{fmap}\,f\;(\mathit{Lan}\;(\kappa, s)) = \mathit{Lan}\;(f\cdot\kappa, s)$$

The existential quantifier is written as a universal quantifier *in front of* the data constructor *Lan*. Ideally, $\mathsf{Lan_J}$ should be given by a **newtype** declaration, but **newtype** constructors must not have an existential context. For similar reasons, we cannot use a deconstructor, that is, a selector function lan°. The type $\mathsf{Lan}_i\,f$ can be seen as a *generalised abstract data type*: $f\;a$ is the internal state and $i\;a \to x$ the observer function — again, the type constructors i and f are likely to be identical. The adjoint transpositions are given by

$$\phi_{\mathsf{Lan}} : \forall i\,f\,g\;.\;(\forall x\;.\;\mathsf{Lan}_i\,f\;x \to g\;x) \to (\forall x\;.\;f\;x \to g\;(i\;x))$$
$$\phi_{\mathsf{Lan}}\;\alpha = \lambda s \to \alpha\;(\mathit{Lan}\;(\mathit{id}, s))$$

$$\phi^\circ_{\mathsf{Lan}} : \forall i\,f\,g\;.\;(\mathit{Functor}\;g) \Rightarrow (\forall x\;.\;f\;x \to g\;(i\;x)) \to (\forall x\;.\;\mathsf{Lan}_i\,f\;x \to g\;x)$$
$$\phi^\circ_{\mathsf{Lan}}\;\beta = \lambda(\mathit{Lan}\;(\kappa, s)) \to \mathit{fmap}\;\kappa\;(\beta\;s)$$

The duality of the construction is somewhat obfuscated in the Haskell code. □

Again, let us specialise the adjoint equations.

$$\begin{array}{ll}
& x \cdot \mathsf{Pre}_{\mathsf{J}}\, \mathit{in} = \Psi\, x \\
\iff & \{ \text{ definition of } \mathsf{Pre}_{\mathsf{J}} \} \\
& x \cdot (\mathit{in} \circ \mathsf{J}) = \Psi\, x \\
\iff & \{ \text{ pointwise } \} \\
& x\, A\, (\mathit{in}\, (\mathsf{J}\, A)\, s) = \Psi\, x\, A\, s
\end{array}
\qquad
\begin{array}{ll}
& \mathsf{Pre}_{\mathsf{J}}\, \mathit{out} \cdot x = \Psi\, x \\
\iff & \{ \text{ definition of } \mathsf{Pre}_{\mathsf{J}} \} \\
& (\mathit{out} \circ \mathsf{J}) \cdot x = \Psi\, x \\
\iff & \{ \text{ pointwise } \} \\
& \mathit{out}\, (\mathsf{J}\, A)\, (x\, A\, s) = \Psi\, x\, A\, s
\end{array}$$

Note that '$\cdot$' in the original equations denotes the (vertical) composition of natural transformations: $(\alpha \cdot \beta)\, X = \alpha\, X \cdot \beta\, X$. Also note that the natural transformations x and in are applied to different type arguments. The usual caveat applies when reading the equations as Haskell definitions: as type application is invisible, the derived equation is indistinguishable from the original one.

Haskell example 26. Continuing Haskell Example 24, the base function of *concat* is straightforward, except perhaps for the types.

$$\begin{array}{llll}
\mathfrak{concat} : \forall x\, .\, (\forall a\, .\, x\, (\mathsf{List}\, a) \to \mathsf{List}\, a) \to & & & \\
\qquad\qquad (\forall a\, .\, \mathfrak{List}\, x\, (\mathsf{List}\, a) \to \mathsf{List}\, a) & & & \\
\mathfrak{concat} & \mathit{concat}\ \ (\mathfrak{Nil}) & = & \mathit{In}\, \mathfrak{Nil} \\
\mathfrak{concat} & \mathit{concat}\ \ (\mathfrak{Cons}\, (l, \mathit{ls})) & = & \mathit{append}\, (l, \mathit{concat}\, \mathit{ls})
\end{array}$$

The base function is a second-order natural transformation. The transpose of *concat* is quite revealing. First of all, its type is

$$\phi\, \mathit{concat} : \mathsf{List} \dot{\to} \mathsf{Ran}_{\mathsf{List}}\, \mathsf{List} \cong \forall a\, .\, \mathsf{List}\, a \to \forall b\, .\, (a \to \mathsf{List}\, b) \to \mathsf{List}\, b \enspace .$$

The type suggests that $\phi\, \mathit{concat}$ is the bind of the list monad (written $\gg\!=$ in Haskell), and this is indeed the case!

$$\begin{array}{llll}
\mathit{concat}' : \forall a\, b\, .\, \mu\mathfrak{List}\, a \to (a \to \mathsf{List}\, b) \to \mathsf{List}\, b & & & \\
\mathit{concat}' \qquad \mathit{as} & = \lambda \kappa & \to & \mathit{concat}\, (\mathit{fmap}\, \kappa\, \mathit{as})
\end{array}$$

For clarity, we have inlined $\mathsf{Ran}_{\mathsf{List}}\, \mathsf{List}$. □

Kan extensions generalise the constructions of the previous section: we have $\mathsf{Lsh}_A\, B \cong \mathsf{Lan}_{(\mathsf{K}\, A)}\, (\mathsf{K}\, B)$ and $\mathsf{Rsh}_A\, B \cong \mathsf{Ran}_{(\mathsf{K}\, A)}\, (\mathsf{K}\, B)$, where K is the constant functor. The double adjunction $\mathsf{Lsh}_X \dashv (-\, X) \dashv \mathsf{Rsh}_X$ is implied by $\mathsf{Lan}_{\mathsf{J}} \dashv (- \circ \mathsf{J}) \dashv \mathsf{Ran}_{\mathsf{J}}$. Here is the proof for the right adjoint:

$$\begin{array}{ll}
& \mathsf{F}\, A \to B \\
\cong & \{ \text{ arrows as natural transformations } \} \\
& \mathsf{F} \circ \mathsf{K}\, A \dot{\to} \mathsf{K}\, B \\
\cong & \{ (- \circ \mathsf{J}) \dashv \mathsf{Ran}_{\mathsf{J}} \} \\
& \mathsf{F} \dot{\to} \mathsf{Ran}_{\mathsf{K}\, A}\, (\mathsf{K}\, B) \\
\cong & \{ \mathsf{Ran}_{\mathsf{K}\, A}\, (\mathsf{K}\, B) \cong \mathsf{Rsh}_A\, B \} \\
& \mathsf{F} \dot{\to} \mathsf{Rsh}_A\, B \enspace .
\end{array}$$

Table 2 summarises our findings.

Table 2. Adjunctions and types of recursion

adjunction	initial fixed-point equation	final fixed-point equation
$\mathsf{L} \dashv \mathsf{R}$	$x \cdot \mathsf{L}\, in = \Psi\, x$ $\phi\, x \cdot in = (\phi \cdot \Psi \cdot \phi^{\circ})\, (\phi\, x)$	$\mathsf{R}\, out \cdot x = \Psi\, x$ $out \cdot \phi^{\circ}\, x = (\phi^{\circ} \cdot \Psi \cdot \phi)\, (\phi^{\circ}\, x)$
$\mathsf{Id} \dashv \mathsf{Id}$	standard fold standard fold	standard unfold standard unfold
$(- \times X) \dashv (-^{X})$	parametrised fold fold to an exponential	curried unfold unfold from a pair
$(+) \dashv \Delta$	recursion from a coproduct of mutually recursive types mutual value recursion on mutually recursive types	mutual value recursion single recursion from a coproduct domain
$\Delta \dashv (\times)$	mutual value recursion single recursion to a product domain	recursion to a product of mutually recursive types mutual value recursion on mutually recursive types
$\mathsf{Lsh}_X \dashv (-\, X)$	—	monomorphic unfold unfold from a left shift
$(-\, X) \dashv \mathsf{Rsh}_X$	monomorphic fold fold to a right shift	—
$\mathsf{Lan}_\mathsf{J} \dashv (- \circ \mathsf{J})$	—	polymorphic unfold unfold from a left Kan extension
$(- \circ \mathsf{J}) \dashv \mathsf{Ran}_\mathsf{J}$	polymorphic fold fold to a right Kan extension	—

5 Related Work

Building on the work of Hagino [17], Malcolm [23] and many others, Bird and de Moor gave a comprehensive account of the "Algebra of Programming" in their seminal textbook [3]. While the work was well received and highly appraised in general, it also received some criticism. Poll and Thompson write in an otherwise positive review [33]:

> The disadvantage is that even simple programs like factorial require some manipulation to be given a catamorphic form, and a two-argument function like concat requires substantial machinery to put it in catamorphic form, and thus make it amenable to manipulation.

The term 'substantial machinery' refers to Section 3.5 of the textbook where Bird and de Moor address the problem of assigning a unique meaning to the defining equation of *append* (called *cat* in the textbook). In fact, they generalise the problem slightly, considering equations of the form

$$x \cdot (in \times id) = h \cdot \mathsf{G}\, x \cdot \phi \ , \tag{14}$$

where ϕ is some suitable natural transformation and h a suitable arrow. Clearly, their approach is subsumed by the framework of adjoint folds.

The seed for this framework was laid in Section 6 of the paper "Generalised folds for nested datatypes" by Bird and Paterson [5]. In order to show that generalised folds are uniquely defined, they discuss conditions to ensure that the more general equation $x \cdot \mathsf{L}\,\mathit{in} = \Psi\,x$, our adjoint initial fixed-point equation, uniquely defines x. Two solutions are provided to this problem, the second of which requires L to have a right adjoint. They also show that the right Kan extension is the right adjoint of pre-composition. Somewhat ironically, the rest of the paper, which is concerned with folds for nested datatypes, does not build upon this elegant approach. Also, they do not consider (adjoint) unfolds. Nonetheless, Bird and Paterson deserve most of the credit for their fundamental insight, so three cheers to them! (As an aside, the first proof method uses colimits and is strictly more powerful. It can be used to give a semantics to functions such as *zip* that are defined by simultaneous recursion over a pair of datatypes: $\times(\mu\mathsf{F}) \to A$. Since the product is not a left adjoint, the framework developed in this paper is not applicable.) A slight variation of adjoint folds was introduced by Matthes and Uustalu [25] under the name *generalised iteration*. They essentially generalise (14) to an arbitrary left adjoint L:

$$x \cdot \mathsf{L}\,\mathit{in} = h \cdot \mathsf{G}\,x \cdot \phi \ ,$$

where $x : \mathsf{L}\,(\mu\mathsf{F}) \to A$, $\phi : \mathsf{L} \circ \mathsf{F} \mathrel{\dot{\to}} \mathsf{G} \circ \mathsf{L}$ and $h : \mathsf{G}\,A \to A$.

An alternative, type-theoretic approach to (co-) inductive types was proposed by Mendler [28]. His induction combinators R^{μ} and S^{ν} map a base function to its unique fixed point. Strong normalisation is guaranteed by the polymorphic type of the base function. The first categorical justification of Mendler-style recursion was given by de Bruin [6]. Interestingly, in contrast to traditional category-theoretic treatments of (co-) inductive types there is no requirement that the underlying type constructor is a covariant functor. Indeed, Uustalu and Vene have shown that Mendler-style folds can be based on difunctors [38]. It remains to be seen whether adjoint folds can also be generalised in this direction. Abel, Matthes and Uustalu extended Mendler-style folds to higher kinds [1]. Among other things, they demonstrate that suitable extensions of Girard's system F^{ω} retain the strong normalisation property and they show how to transform generalised Mendler-style folds into standard ones.

There is a large body of work on 'morphisms'. Building on the notions of functors and natural transformations Malcolm generalised the Bird-Meertens formalism to arbitrary datatypes [23]. Incidentally, he also discussed how to model mutually recursive types, albeit in an ad-hoc manner. His work assumed **Set** as the underlying category and was adapted by Meijer, Fokkinga and Paterson to the category **Cpo** [27]. The latter paper also popularised the now famous terms *catamorphism* and *anamorphism* (for folds and unfolds), along with the banana and lens brackets ($(\!|-|\!)$ and $[\!(-)\!]$). (The term catamorphism was actually coined by Meertens, the notation $(\!|-|\!)$ is due to Malcolm, and the name banana bracket is attributed to van der Woude.) The notion of a *paramorphism* was introduced by Meertens [26]. Roughly speaking, paramorphisms generalise primitive recursion to arbitrary datatypes. Their duals, *apomorphisms*, were only studied later by Vene and Uustalu [39]. (While initial algebras have been the subject of intensive research, final coalgebras have received less attention — they are certainly under-appreciated [13].) Fokkinga captured mutually recursive functions by *mutomorphisms* [10]. He also observed that Malcolm's *zygomorphisms* arise as a special case, where one function depends on the other, but not the other way round. (Paramorphisms further specialise zygomorphisms in that the independent function is the identity.) An alternative solution to the '*append*-problem' was proposed by Pardo [31]: he introduces *folds with parameters* and uses them to implement *generic accumulations*. His accumulations subsume Gibbons' *downwards accumulations* [12].

The discovery of nested datatypes and their expressive power [4,8,30] led to a flurry of research. Standard folds on nested datatypes, which are natural transformations by construction, were perceived as not being expressive enough. The aforementioned paper by Bird and Paterson [5] addressed the problem by adding extra parameters to folds leading to the notion of a *generalised fold*. The author identified a potential source of inefficiency — generalised folds make heavy use of mapping functions — and proposed *efficient generalised folds* as a cure [18]. The approach being governed by pragmatic concerns was put on a firm theoretical footing by Martin, Gibbons and Bayley [24] — rather imaginatively the resulting folds were called *disciplined, efficient, generalised folds*. The fact that standard folds are actually sufficient for practical purposes — every adjoint fold can be transformed into a standard fold — was later re-discovered by Johann and Ghani [21].

We have shown that all of these different morphisms and (un-) folds fall under the umbrella of adjoint (un-) folds. (Paramorphisms and apomorphisms require a slight tweak though: the argument or result must be guarded by an invocation of the identity.) It remains to be seen whether more exotic species such as *histomorphisms* or *futomorphisms* [37] are also subsumed by the framework. (It does work for the simple example of Fibonacci.)

6 Conclusion

I had the idea for this paper when I re-read "Generalised folds for nested datatypes" by Bird and Paterson [5]. I needed to prove the uniqueness of a certain function and I recalled that the paper offered a general approach for doing this. After a while I began to realise that the approach was far more general than I and possibly also the authors initially realised.

Adjoint folds and unfolds strike a fine balance between expressiveness and ease of use. We have shown that many if not most Haskell functions fit under this umbrella. The mechanics are straightforward: given a (co-) recursive function, we abstract away from the recursive calls, additionally removing occurrences of *in* and *out* that guard those calls. Termination and productivity are then ensured by a naturality condition on the resulting base function.

The categorical concept of an adjunction plays a central role in this development. In a sense, each adjunction captures a different recursion scheme — accumulating parameters, mutual recursion, polymorphic recursion on nested datatypes etc. — and allows the scheme to be viewed as an instance of an adjoint (un-) fold.

Of course, the investigation of adjoint (un-) folds is not complete; it has barely begun. For one thing, it remains to develop the calculational properties of adjoint (un-) folds. Their definitions

$$x = (\!|\Psi|\!)_{\mathsf{L}} \iff x \cdot \mathsf{L}\, in = \Psi\, x$$
$$x = [\!(\Psi)\!]_{\mathsf{R}} \iff \mathsf{R}\, out \cdot x = \Psi\, x$$

gives rise to the usual reflection, computation and fusion laws. In addition, one might hope for elegant laws manipulating the underlying adjoint functors. For another thing, it will be interesting to see whether other members of the morphism zoo can be fitted into the framework.

A final thought: most if not all constructions in category theory are parametric in the underlying category, resulting in a remarkable economy of expression. Perhaps, we should spend more time and effort into utilising this economy for programming. This possibly leads to a new style of programming, which could be loosely dubbed as *category-parametric programming*.

Acknowledgements

Thanks are due to Tom Harper, Daniel James and Nicolas Wu for carefully proof-reading (an even longer) draft of the manuscript and for suggesting numerous improvements regarding both contents and style. Many thanks also go the members of the Algebra of Programming group at Oxford for several fruitful discussions on adjoint functors and for suggesting the subtitle. I owe a particular debt of gratitude to Thorsten Altenkirch for showing me how to derive the right adjoint of $- \dot{\times} \mathsf{H}$ (at that time, I could not imagine that I would ever find an application for the construction). Thanks are also due to the anonymous referees of MPC 2010 who provided several pointers and some historical perspective. Finally, I would like to thank Richard Bird for general advice and for encouraging (even pushing me) to write things up.

References

1. 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)
2. Bird, R.: Introduction to Functional Programming using Haskell, 2nd edn. Prentice Hall, Europe (1998)
3. Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall, Europe (1997)
4. Bird, R., Meertens, L.: Nested datatypes. In: Jeuring, J. (ed.) MPC 1998. LNCS, vol. 1422, pp. 52–67. Springer, Heidelberg (1998)
5. Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects of Computing 11(2), 200–222 (1999)
6. de Bruin, P.J.: Inductive types in constructive languages. Ph.D. thesis, University of Groningen (1995)

7. Cockett, R., Fukushima, T.: About Charity. Yellow Series Report 92/480/18, Dept. of Computer Science, Univ. of Calgary (June 1992)
8. Connelly, R.H., Morris, F.L.: A generalization of the trie data structure. Mathematical Structures in Computer Science 5(3), 381–418 (1995)
9. Fokkinga, M.M., Meertens, L.: Adjunctions. Tech. Rep. Memoranda Inf. 94-31, University of Twente, Enschede, Netherlands (June 1994)
10. Fokkinga, M.M.: Law and Order in Algorithmics. Ph.D. thesis, University of Twente (February 1992)
11. Fokkinga, M.M., Meijer, E.: Program calculation properties of continuous algebras. Tech. Rep. CS-R9104, Centre of Mathematics and Computer Science, CWI, Amsterdam (January 1991)
12. Gibbons, J.: Generic downwards accumulations. Sci. Comput. Program. 37(1-3), 37–65 (2000)
13. Gibbons, J., Jones, G.: The under-appreciated unfold. In: Felleisen, M., Hudak, P., Queinnec, C. (eds.) Proceedings of the third ACM SIGPLAN international conference on Functional programming, pp. 273–279. ACM Press, New York (1998)
14. Gibbons, J., Paterson, R.: Parametric datatype-genericity. In: Jansson, P. (ed.) Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming, pp. 85–93. ACM Press, New York (August 2009)
15. Giménez, E.: Codifying guarded definitions with recursive schemes. In: Smith, J., Dybjer, P., Nordström, B. (eds.) TYPES 1994. LNCS, vol. 996, pp. 39–59. Springer, Heidelberg (1995)
16. Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Initial algebra semantics and continuous algebras. Journal of the ACM 24(1), 68–95 (1977)
17. Hagino, T.: A typed lambda calculus with categorical type constructors. In: Pitt, D.H., Rydeheard, D.E., Poigné, A. (eds.) Category Theory and Computer Science. LNCS, vol. 283. Springer, Heidelberg (1987)
18. Hinze, R.: Efficient generalized folds. In: Jeuring, J. (ed.) Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal, pp. 1–16 (July 2000); The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19
19. Hinze, R.: Functional Pearl: Streams and unique fixed points. In: Thiemann, P. (ed.) Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08), pp. 189–200. ACM Press, New York (September 2008)
20. Hinze, R., Peyton Jones, S.: Derivable type classes. In: Hutton, G. (ed.) Proceedings of the 2000 ACM SIGPLAN Haskell Workshop. Electronic Notes in Theoretical Computer Science, vol. 41(1), pp. 5–35. Elsevier Science, Amsterdam (August 2001); The preliminary proceedings appeared as a University of Nottingham technical report
21. Johann, P., Ghani, N.: Initial algebra semantics is enough! In: Della Rocca, S.R. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 207–222. Springer, Heidelberg (2007)
22. Mac Lane, S.: Categories for the Working Mathematician, 2nd edn. Graduate Texts in Mathematics. Springer, Berlin (1998)
23. Malcolm, G.: Data structures and program transformation. Science of Computer Programming 14(2-3), 255–280 (1990)
24. Martin, C., Gibbons, J., Bayley, I.: Disciplined, efficient, generalised folds for nested datatypes. Formal Aspects of Computing 16(1), 19–35 (2004)
25. Matthes, R., Uustalu, T.: Substitution in non-wellfounded syntax with variable binding. Theoretical Computer Science 327(1-2), 155–174 (2004)

26. Meertens, L.: Paramorphisms. Formal Aspects of Computing 4, 413–424 (1992)
27. Meijer, E., Fokkinga, 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)
28. Mendler, N.P.: Inductive types and type constraints in the second-order lambda calculus. Annals of Pure and Applied Logic 51(1-2), 159–172 (1991)
29. Mycroft, A.: Polymorphic type schemes and recursive definitions. In: Paul, M., Robinet, B. (eds.) Programming 1984. LNCS, vol. 167, pp. 217–228. Springer, Heidelberg (1984)
30. Okasaki, C.: Catenable double-ended queues. In: Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming, Amsterdam, The Netherlands, June 1997, pp. 66–74 (1997); ACM SIGPLAN Notices 32(8), (August 1997)
31. Pardo, A.: Generic accumulations. In: Gibbons, J., Jeuring, J. (eds.) Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, vol. 243, pp. 49–78. Kluwer Academic Publishers, Dordrecht (July 2002)
32. Peyton Jones, S.: Haskell 98 Language and Libraries. Cambridge University Press, Cambridge (2003)
33. Poll, E., Thompson, S.: Book review: "The Algebra of Programming". J. Functional Programming 9(3), 347–354 (1999)
34. Sheard, T., Pasalic, T.: Two-level types and parameterized modules. J. Functional Programming 14(5), 547–587 (2004)
35. The Coq Development Team: The Coq proof assistant reference manual, `http://coq.inria.fr`
36. Trifonov, V.: Simulating quantified class constraints. In: Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell, pp. 98–102. ACM, New York (2003)
37. Uustalu, T., Vene, V.: Primitive (co)recursion and course-of-value (co)iteration, categorically. Informatica, Lith. Acad. Sci. 10(1), 5–26 (1999)
38. Uustalu, T., Vene, V.: Coding recursion a la Mendler (extended abstract). In: Jeuring, J. (ed.) Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal, pp. 69–85 (July 2000); The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19
39. Vene, V., Uustalu, T.: Functional programming with apomorphisms (corecursion). Proceedings of the Estonian Academy of Sciences: Physics, Mathematics 47(3), 147–161 (1998)

An Abstract Machine for the Old Value Retrieval[⋆]

Piotr Kosiuczenko

Institute of Information Systems, WAT, Warsaw, Poland

Abstract. The evaluation of post-conditions requires the computation of old attribute values. Until recently, existing computation methods were not efficient in terms of time- and space-complexity. Moreover they were applicable only to a restricted form of post-conditions. Recently a new algorithm was proposed to overcome those deficiencies. In this paper, an abstract machine corresponding to this algorithm is defined. Its transitions simulate steps of object-oriented systems and preserve an invariant implying properties needed to compute old attribute values. The machine is based on a kind of structure called here sufficiently persistent, as opposed to persistent and partially persistent structures. A space-bound on the structure size is given. It is also demonstrated that methods which do not have post-conditions can be abstracted away.

Keywords: persistent data structures, old value retrieval, @pre, old.

1 Introduction

Contracts are used to specify object-oriented systems from the user point of view [10]. They consist of three basic constraint types: invariants, pre- and post-conditions. The system consistency is ensured by invariants. A pre-condition specifies in which states a method can be called. Post-conditions specify system states after a method execution. Their validation is not straightforward since it is necessary to compare attribute values in method pre- and post-states and method calls can be nested. Old attribute values are accessed with the help of operator `@pre` in case of Object Constraint Language (OCL, see [12]) and `old` in case of Eiffel [11], Java Modeling Language (JML, see [4]) and Spec# [1]. The copying of a whole pre-state before a method execution is out of question due to its time- and memory-cost. This problem can be avoided by saving before the execution values of those attributes whose @pre/old values are referred to in the corresponding post-condition. It is similar to the way old variable values are treated in the Hoare logic [6], which uses fresh variables to store values from before a method execution. This approach is followed in tools supporting other contractual languages such as OCL, JML and Spec# (see [9,13] for an OCL tools overview). The current implementations of `@pre` operator are discussed in [8,9,3].

⋆ This research has been partially supported by grant No PBZ-MNiSW-DBO-02/I/2007/.

This approach requires the restriction of post-conditions syntax to formulas of the form: `t`$_0$`[t`$_1$`@pre/x`$_1$`, ..., t`$_n$`@pre/x`$_n$`]` (we use here the OCL notation), where term `t`$_0$ does not include `@pre` and term `t`$_i$`@pre` is obtained from term `t`$_i$ by replacing every attribute `a` by `a@pre`, for i = 1,..., n (for example, term `(self.a.b)@pre` is an abbreviation of OCL term `self.a@pre.b@pre`). `[t`$_1$`@pre/x`$_1$`, ..., t`$_n$`@pre/x`$_n$`]` denotes here the simultaneous substitution of terms `t`$_i$ for variables `x`$_i$, for i = 1,..., n. Values of terms `t`$_i$ are computed before the underlying method is executed, saved and then after the method execution used to compute the value of the post-condition. For example constraint `self.a.b@pre = self.a@pre.b@pre+1` is not of the above form. In this case, it is not possible to compute in the pre-state values of attribute `b` for the objects related by `a` with `self` in the post-state. To do this one would have to know in advance which objects will be related by `a` in the post-state.

There are other problems with this approach. If terms `t`$_i$ are of collection types, then the actual collection values must be cloned. Such clones are computationally expensive and pose logical problems, since reference identity cannot be used for object comparison. Computation of all potentially needed values in the pre-state can be even nonterminating. For example, let us consider expression `if 1 + 1 = 2 then 1 else q@pre endif` where `q` is a computationally complex, or nonterminating, integer-valued query. Obviously, in general there is no need to evaluate `q` in the pre-state to compute the value of the whole expression.

In the paper [9] an algorithm was proposed which overcomes above mentioned problems. It allows to access @pre-values during post-condition evaluation as needed. Consequently, it is applicable to all forms of post-conditions. Its execution does not increase the complexity class of post-condition evaluation as if the computation of old values had a constant time. Since values of @pre-terms are not recorded in advance, there is neither need to clone system states nor to restrict the post-condition syntax. The algorithm is implemented in AspectJ. It superimposes the so called fat structures [5] on object-oriented systems. Those structures make arbitrary linked structures partially persistent, i.e. when a sequence of updates is performed, then all versions of a linked structure can be accessed and the newest version can be modified.

The method proposed in [5] applies to a sequence of operations. Basically, every modification of an attribute is accompanied by storing the modified value and the corresponding modification number in a data structure. The notion of persistence requires that all previous versions of a structure can be accessed and modified [5]. However method calls can be recursive and form a tree instead of a sequence. There is no obvious relation between consecutive versions of modified attributes and the call-stack. Thus we need a different way of handling attribute modifications. On the other hand, fat structures hinder the garbage collection. Therefore stored information should be minimized. In the paper [2], the notion of semi persistence has been coined for backtracking algorithms. The authors have observed that when backtracking from a branch it suffices to use the old version of a structure without undoing changes. The ancestors of the current version are reused, but never another version obtained from a common ancestor.

They call a data structure semi-persistent if only ancestors of the newest version can be updated. This notion differs from partial persistence, since it allows the update of ancestor structures. However, the authors have not considered ways of ensuring semi persistence, nor the way we can handle old attribute values and minimize the memory use.

In this paper we define a labelled transition system, or as we sometimes say an abstract machine, which simulates steps of object-oriented programs and manages attribute histories. Method execution is simulated by transitions like method call, setting object attributes and method return. For every attribute we define a history-stack whose elements are pairs consisting of an old attribute's values and its time-stamps being a calls number. Basically, if an attribute is set for the first time during a method execution and the stack does not include timely snapshots, its previous value is pushed on the stack with a time-stamp being the number of the currently executing method. We define also clean operations which remove outdated snapshots from history-stacks. We formulate an invariant which guarantees that @pre-attribute values are recoverable from the current ones and the corresponding histories. We show that it is preserved by the transitions.

An efficient space management is crucial here, since objects reclaim during garbage collection is restrained due to saving old attribute values in history queues. Partial persistence is more than we need and as such causes unnecessary space overhead. In that case, the history of an attribute has size equal to the number of its modifications. The access time to attribute values from before the current method execution is logarithmic if binary search trees are use to store attribute histories [5]. We call a structure sufficiently persistent, if all its versions directly proceeding non-terminated method calls can be accessed, but only the most recent version can be updated. Sufficiently persistent structures are related to semi persistent structures as partially persistent structures to persistent ones. Defined abstract machine corresponds to a general form of sufficiently persistent structures. It should be stressed, that our methods applies to all forms of linked structures, not only object-oriented ones.

We show that in case of the Towers of Hanoi algorithm, our method requires space of size $O(n^2)$, where n is the number of rings; whereas partially persistent structures require space of exponential size. In our case, at any point of execution the lengths of attribute histories are linearly bound by the maximal size of the call-stack reached up to that point. We show that the cumulative access time to an attribute value from before a method execution is constant. This is due to cleaning of outdated values and to the fact that in our approach we register only the attribute values prior to a method execution, not every version. If recursion is replaced by iteration, then our method results in leaner structures.

Some methods do not have post-conditions containing attributes of the form $a@pre$; we call them irrelevant. We prove that it is possible to abstract from calls of such methods. We do it by defining a proper bisimulation relation. This corresponds in a sense to the refactoring pattern called inline method [7], which results in a less structured code. Thus the length of attribute history can be linearly bound by the number of relevant methods on the call-stack.

This paper is organized as follows. In Section 2, we define a labelled transition system which manages old attribute values. In Section 3, we discuss its properties and define an invariant preserved by its transitions. In Section 4, we prove that when computing old attribute values it is possible to abstract from methods without post-conditions. In Section 5, we relate the abstract machine to the AspectJ implementation proposed in [9] and consider the time- and space-complexity of proposed method. Section 6 concludes this paper.

2 The Abstract Machine

In this section we define a labelled transition system, or as we sometimes say an abstract machine. Transitions of this system define steps of an object-oriented system and the way attribute values are archived. Executions of object-oriented programs, for example written in Java, correspond to a restricted subset of all runs of this machine. This machine does not take into consideration restrictions due to the use of method parameters. In any execution state it is possible to modify any object. For simplicity we do not deal with object creation explicitly. We assume that the initialization of an object consists of setting its attributes which are all initially equal to $\perp$. Similarly we do not model here the garbage collection. We discuss the issue of efficient space use in subsection 5.2.

2.1 States

In this subsection we define states of the abstract machine. We assume that there exists an infinite set of object locations/addresses OL and that the undefined symbol $\perp$ does not belong to OL. $A = \{a_1, ..., a_n\}$ is a finite set of attributes. Op is the set of methods/operations. We assume that *main* belongs to Op. N is the set of natural numbers. Below for an arbitrary set B, $B_\perp$ denotes the set $B \cup \{\perp\}$. We model a heap state (called here object store or simply store) by a function mapping pairs consisting of an attribute and an object location to object location, i.e. $St =_{def} \{st : A \times OL \rightarrow OL_\perp\}$.

In our model, all objects can have potentially all attributes. Classes can be modelled by infinite subsets of OL with the restriction on attributes. Thus if say objects of class B do not possess attribute a, then for every object o of class B and every store st, $st(a, o) = \perp$ must hold. Likewise we do not deal directly with inheritance. However, we can restrict in a similar way method calls. Below we abstract from method parameters making the machine runs even more general; considered methods can potentially modify any object. Clearly we get a number of machine runs which do not correspond to an execution of a method implemented in a language such as Java.

We call a pair consisting of an attribute value and the corresponding time-stamp 'attribute snapshot'. An attribute history for an object is a sequence of snapshots: $H =_{def} (OL \times N)^*$. Attribute histories are modelled by history functions. A history function for an attribute and a location is either undefined or equal to a sequence of snapshots. $AH =_{def} \{h : A \times OL \rightarrow H_\perp\}$ is the set of such functions. $SH =_{def} (St \times Op \times N)^+$ is the set consisting of store histories, or as we sometimes say heap histories. Such a history is a sequence of triples consisting of a store, the name of a currently executing method and a time-stamp corresponding to a method call. Method calls are numbered starting with one. For every subsequent call, the counter is increased by one.

Computation states are the states of the abstract machine. They consist of a nonempty store history, an attributes history function and a value corresponding to the number of executed method calls: $CS =_{def} SH \times AH \times N$. The initial state models the situation when method `main` starts to execute. The store is constantly equal to $\perp$ since no attributes are set. All values are undefined, since there is nothing to be archived when `main` is called. $st_\perp$ and $h_\perp$ are respectively the store and the history functions constantly equal to $\perp$. The call number is 0, since no method different from `main` started to execute, i.e. $inSt =_{def} ((st_\perp, main, 0), h_\perp, 0)$.

2.2 Transitions

In this subsection we define a transition relation R corresponding to computation following the post-condition specification style (cf. e.g. [12]). They are five kinds of transition steps. The first one corresponds to a method call. The second one concerns setting an attribute. This transition records changed attribute values in the attribute history-stacks. The third one concerns a method return. The last two concern cleaning an attribute history. In the first case, only the top of a history-stack is cleaned. In the second one, all outdated snapshots are removed.

We denote by $f[x \mapsto v]$ a function that maps x to v and differs from f only for argument x; i.e. $f[x \mapsto v](x) = v$ and $f[x \mapsto v](y) = f(y)$, for $y \neq x$. Let computation states cs, cs' be of the form (sh, h, n) and (sh', h', n'), respectively. Below we assume that store history sh has the form $(st_0, main, 0)...(st_k, op_k, n_k)$. We assume that sh_0 is its initial subsequence up to $i-1$, i.e. $sh = sh_0(st_k, op_k, n_k)$. If $h(a, o) \neq \perp$, then we assume that $h(a, o)$ is a sequence of the form $ah_0(o_l, r_l)$, where $ah_0 = (o_0, r_0)...(o_{l-1}, r_{l-1})$. Note that op_k is the method executing when the transitions below are started and n_k is its call number. Let $a \in A$, $o, o' \in OL$.

1. $cs\, R_{call(op)}\, cs' \Leftrightarrow n' = n + 1 \wedge sh' = sh(st_k, op, n+1) \wedge h' = h$
2. $cs\, R_{set(a,o,o')}\, cs' \Leftrightarrow \exists_{st' \in St} n' = n \wedge sh' = sh_0(st', op_k, n_k)$ where $st' = st_k[(a, o) \mapsto o']$. Moreover,
   (i) $h(a, o) = \perp \Rightarrow h' = h[(a, o) \mapsto (o', n_k)]$
   (ii) $r_l < n_k \wedge st_k(a, o) \neq o_l \Rightarrow h' = h[(a, o) \mapsto h(a, o)(st_k(a, o), n_k)]$
   (iii) $r_l < n_k \wedge st_k(a, o) = o_l \Rightarrow h' = h[(a, o) \mapsto ah_0(o_l, n_k)]$
   (iv) In other cases $h' = h$
3. $cs\, R_{return(op_k)}\, cs' \Leftrightarrow sh' = (st_0, main, 0)\, ...(st_{k-2}, op_{k-2}, n_{k-2})$ $(st_k, op_{k-1}, n_{k-1})$
   (i) $|sh_0| > 0 \Rightarrow h' = h \wedge n' = n$
   (ii) $|sh_0| = 0 \Rightarrow \forall_{a \in A,\, o \in OL} h'(a, o) = if\ h(a, o) \neq \perp\ then\ \epsilon\ else \perp endif$ $\wedge\, n' = 0$

4. $cs\ R_{cleanTop(a,o)}\ cs' \Leftrightarrow 1 < |h(a,o)| \wedge n_k \leqslant r_{l-1} \wedge sh' = sh \wedge n' = n \wedge$
   $h' = h[(a,o) \mapsto ah_0]$
5. $cs\ R_{cleanWhole(a,o)}\ cs' \Leftrightarrow 1 < |h(a,o)| \wedge sh' = sh \wedge n' = n \wedge$
   $h' = h[(a,o) \mapsto ah]$ where $ah = (o_{s0}, r_{s0})(o_{s1}, r_{s1})...(o_{sp}, r_{sp})$ is a subsequence of $h(a,o)$ such that $(\forall_{0 \leqslant i \leqslant k,\, 0 \leqslant j \leqslant l}\, n_i \leqslant r_j \Rightarrow \exists_{0 \leqslant d \leqslant p}\, n_i \leqslant r_{sd} \leqslant r_j) \wedge$
   $(\forall_{0 \leqslant d < p}\, \exists_{0 \leqslant i \leqslant k}\, r_{sd} < n_i \leqslant r_{s(d+1)})$

Transition (1) corresponds to a method call. Since in this case the attributes histories are not modified, the last store is simply duplicated. Every method call has a number being the successor of the number of the last method call. Similarly, the new store has the time-stamp corresponding to the last call number.

Transition (2) corresponds to attribute modification. It should be noted that in terms of aspect-oriented programming, o is the target of the set operation. Setting attribute a modifies only the heap, i.e. the topmost store st and possibly history of a for o. It does not change number n of the most recent method call, nor histories of other attributes.

(2i), (2ii) and (2iii) concern attribute history and do not constrain the store change. (2i) corresponds to the case when for location o attribute a was not set before. In this case the history of a for o is initialized by storing the first attribute value. (2ii) corresponds to the case when the attribute has a history, but either the history is empty or the topmost attribute value is different from the attribute value in the pre-state and the topmost time-stamp concerns an older method call than the currently executing one (i.e. $r_l < n_k$). In this case, the history is extended by the value of a in the pre-state. The corresponding time-stamp is the number of the currently executing method. If the history is empty or $r_l < n_k$, i.e. the last change of a for o happened before the current method execution, then the history is extended by the snapshot made prior to the execution of set. $st_k(a,o)$ is the value of a in the pre-state. In case of (2iii) as in case of (2ii), the topmost time-stamp concerns an older method call (i.e. $r_l < n_k$), but the topmost attribute value is equal to the attribute value in the pre-state. The attribute value is not modified after call number r_l up to n_k. In this case the history is not extended, but the time-stamp is updated and set equal to the number of the currently executing method. It should be noted that an attribute may be modified during the same method execution several times. If the time-stamp r_l of the last snapshot was not increased to n_k, then during the next modification of a the already modified value could be stored in a new snapshot. Thus the new snapshot can potentially contain a value unequal to the value from before the execution.

Transition (3) concerns method return. When a method returns, the control returns to the previously executing method, i.e. method which made the call. This results with replacement of the store st_{k-1} corresponding to the previously executing method with actual store st_k, since all changes made by methods called later contribute to the current state of the store; the method which called the terminated method resumes its execution on st_k.

(3i) handles the case when after the method return control returns to a method different from `main`. In this case attribute histories remain unchanged. (3ii) handles the case when the control returns to the `main` method. In this case, there are no other methods on the call-stack and all attribute histories can be emptied, i.e. set to ϵ, and the method counter can be set to 0. Undefined histories remain undefined. In this way we optimize the memory use before `main` calls another method.

Transition (4) does not change the store, but concerns cleaning an attribute history. Due to method returns, it may happen that an object history contains at its top snapshots made during the execution of terminated methods. The topmost snapshot does not refer to the pre-state of currently executing methods if the previous snapshot contains a time-stamp larger or equal to the time-stamp of currently executing method. In this case the topmost snapshot is removed from the attribute history. It minimizes the memory use. It should be stressed that in general popping an attribute history-stack after a method which modified it returns would be incorrect, since the new value of the attribute may differ from the previous one. After the return there would be neither a way to figure out that the attribute was modified nor to retrieve the @pre-value from the history.

Outdated snapshots can be located not only at the top of a history-stack, but also inside. Transition (5) cleans the whole attribute history. It leaves in the history sequence all snapshots with time-stamps being the smallest upper bound of a unterminated call number and removes all others. Thus, if $n_i \leqslant r_j < r_{j+1} \leqslant n_{i+1}$ or $m \leqslant r_j < r_{j+1}$, then the snapshot number $j+1$ is removed.

Clean operations can be invoked either when an attribute is modified or when the method which modified the attribute terminates. The second case requires registration of objects modified during a method execution. We followed this approach in the paper [9].

3 Invariant

In this section we discuss properties of proposed structure and the way attribute archiving is done. We formulate an invariant concerning states of the abstract machine and use it to prove the correctness of the proposed retrieval method.

A method which is called by another method terminates before the calling method. In general, method numbering is performed in proposed algorithm according to the pre-order traversal of the recursive call tree, and method returns are performed according to the post-order traversal of the tree. Thus the call numbers stamping attribute values are not necessarily used monotonically; i.e. a value snapshot made later may have a smaller time-stamp. Moreover, for every modified attribute, we store only the version prior to the method call, not every subsequent value.

A partially persistent data structure always preserves the previous version of itself when it is modified. It supports nonrecursive sequences of updates applying to its most recent version, but any previously existing version can be accessed [5]. In the case of method calls information about all previous forms of a modified structure is not needed, since we do not need information how terminated methods modified it. We need to know only how the methods on the call-stack modified it. In particular, if all method calls terminate, then there is no need for information about the past forms of the structure. In this case, the ephemeral form, i.e. the plain structure without any information about its previous forms [5], suffices. Thus, we need a notion which is more appropriate than the notion of partial persistence. We call a structure sufficiently persistent if for every non-terminated method call the version from before the call can be accessed, but only the most recent version can be updated. The abstract machine defined in the previous section corresponds to a general form of sufficiently persistent structures.

Below we assume that the conjunction binds stronger than the disjunction, that the disjunction binds stronger than the implication and that quantifiers bind weakest.

Invariant
Let $cs = (sh, h, n)$ be a computation state where store history sh has the form $(st_0, main, 0)(st_1, op_1, n_1)...(st_k, op_k, n_k)$. If $h(a, o) \neq \perp$, then we assume that $h(a, o)$ has the form $(o_0, r_0)...(o_l, r_l)$.

(a) $(\forall_{0 \leqslant i<k} st_i(a, o) \neq \perp \Rightarrow st_{i+1}(a, o) \neq \perp) \wedge (st_k(a, o) = \perp \Leftrightarrow h(a, o) = \perp)$
(b) $0 < n_1 < ... < n_k \leqslant n$
(c) $h(a, o) \neq \perp \Rightarrow 0 \leqslant r_0 < ... < r_l \leqslant n$
(d) $\forall_{0 \leqslant i<k} \perp \neq st_i(a, o) \neq st_{i+1}(a, o) \Rightarrow \exists_{0<j \leqslant l} n_{i+1} \leqslant r_j$
(e) $\forall_{0 \leqslant i<k,\, 0 \leqslant j \leqslant l} st_i(a, o) \neq \perp \wedge n_{i+1} \leqslant r_j \wedge (1 \leqslant j-1 \Rightarrow r_{j-1} < n_{i+1}) \Rightarrow st_i(a, o) = o_j$

By *Inv* we denote the above defined invariant and by $cs \models Inv$ we denote the fact that it is satisfied in computation state cs. Point (a) says that if an attribute is defined for a location, then it remains to be so. Moreover, an attribute is defined if, and only if, the corresponding history is defined as well. Point (b) says that the store numbers, corresponding to call numbers, grow strictly monotonically and are bound by the number of the last call. Similarly (c) says that time-stamp numbers in an attribute history grow strictly monotonically. (d) says that after any attribute value change, a snapshot of this attribute is made. (e) says that the earliest snapshot of an attribute following or made at the same time as a method call stores the value of the attribute in the preceding store or the attribute is not defined in that store. We call it the value witness for a in st_i. (d) and (e) imply that if the value of attribute a is different for two consecutive stores st_i and st_{i+1}, then there is a snapshot (o_j, r_j) of a which was made at the same time or later than the call of op_{i+1}, but before the next call on the call-stack if there is any. If the antecedent of (e) is satisfied, then we say that r_j is the direct follower of n_{i+1}. It is the smallest snapshot's time-stamp directly following n_{i+1}; it can be seen as the smallest upper bound of n_{i+1} (cf. transition 5). $o_j = st_i(a, o)$ if $st_i(a, o) \neq \perp$. Thus (d) and (e) imply that if $st_i(a, o) \neq \perp \wedge st_i(a, o) \neq st_{i+1}(a, o)$, then there is a direct follower r_j of n_{i+1} and the corresponding location o_j stores the @pre-value.

Definition 1. *For $0 \leqslant i < k$ and state st_{i+1}, we define $a@pre$ in the following way:* $st_{i+1}(a@pre, o) = st_i(a, o)$

The next lemma says that $a@pre$ can be retrieved from the attribute history. It is either unchanged or saved in the history as the earliest snapshot made during or after the current method call. This lemma shows that histories of object attributes contain enough information to compute $a@pre$-values. Thus, for every method on the call-stack and every attribute the value of the attribute prior to the method call can be retrieved.

Lemma 2. *Let cs be a computation state described in the invariant, let $0 \leqslant i < k$ and let $0 \leqslant j \leqslant l$. If $cs \models Inv$ and $st_i(a, o) \neq \bot$, then $st_{i+1}(a@pre, o) =$ if there exists an index j such that $n_{i+1} \leqslant r_j \wedge (0 < j-1 \Rightarrow r_{j-1} < n_{i+1})$ then o_j else $st_{i+1}(a, o)$ endif.*

Proof. Let $st_i(a, o) \neq \bot$. If the condition of the if-then-else-statement is satisfied, then part (e) of the invariant implies that $st_i(a, o) = o_j$. If it is not satisfied, then the consequent of (d) is negated. Consequently its antecedent is negated. Therefore the fact that $st_i(a, o) \neq \bot$ implies that $st_i(a, o) = st_{i+1}(a, o)$. ♦

The next theorem says that the invariant is satisfied in the initial states of the abstract machine and preserved by state transitions.

Theorem 3. ***[Invariant Preservation]***
Inv is satisfied in the initial state inSt. If $cs \models Inv$ and $cs\, R_{step}\, cs'$, where step is one of the transition steps defined in Subsection 2.2, then $cs' \models Inv$.

Proof. For a function f, $Rg(f)$ denotes the range of f. For $inSt$, $n = 0$ and $Rg(st_\bot) = Rg(h_\bot) = \{\bot\}$. Consequently, (a) and (e) are satisfied trivially. (b), (c) and (d) follow from the fact that there are no consecutive stores and that histories are empty.

Let us observe that every transition preserves the first part of (a), since there is no transition setting attributes or histories back to $\bot$. The second part follows from the fact that initially all attributes and histories are undefined. An attribute is set at the same time as its history is initialized. Afterwards it remains to be so, since there is no transition setting attributes or histories back to $\bot$. Similarly, it is easy to observe that every transition preserves (b).

We prove now preservation of (c), (d) and (e) by all kinds of transition steps. Let $cs = (sh, h, n)$, $cs' = (sh', h', n')$, $sh_0 = (st_0, main, 0)...(st_k, op_k, n_k)$ and $h(a, o) = (o_0, r_0)...(o_l, r_l)$. Moreover, let $cs \models Inv$.

A call does not modify the heap, but only adds a copy of the last store to the store history. Let $cs\, R_{call(op)}\, cs'$ and let $sh = sh_0$, then accordingly to the definition of call transition $cs' = (sh(st_k, op, n+1), h, n+1)$, i.e. the last store is duplicated, the previous history is kept unchanged and the last call number is increased by 1. Preservation of (c) follows from the assumption that $cs \models Inv$. If the value of attribute a for location o is different in two consecutive stores st_i, st_{i+1}, then $i+1 \leqslant k$. Preservation of (d) and (e) follows from that assumption and the fact that $h' = h$, $n_k \leqslant n$ and $r_j \leqslant r_l \leqslant n < n + 1$, for r_j being the time-stamp of the witness for a in st_i.

Set-operation does not extend the heap history, but only modifies the current store, i.e. the last store in the store history. We assume that $cs\, R_{set(a,o,o')}\, cs'$, $sh = sh_0(st, op, m)$ and $sh' = sh_0(st', op, m)$, for $st' = st[(a, o) \mapsto o']$. Note that $n_k < m$ according to the fact that (b) is satisfied in the pre-state.

If case (2i) applies, then (c) is preserved in a trivial way. If (2ii) applies, then a new snapshot is added to the attribute history; the corresponding time-stamp is larger than the previous one but smaller than or equal to the number of last call, and the saved value is different from the previous one if there is a previous snapshot. Similarly, if (2iii) is applicable, then the time-stamp of the last snapshot is increased, but its value is smaller than or equal to the number of last method call. No new snapshot is added to the attribute history. If (2iv) is applicable, then a is set again for o, the value of a was archived already and histories are not modified. Thus in all cases (c) is preserved.

If case (2i) is applies, then the antecedent of (d) is not satisfied in cs', since a is set for o for the first time. Likewise the antecedent of (e) is not satisfied for any particular i and j, since history $h(a, o)$ is empty. Thus (d) and (e) are preserved in a trivial way. If case (2ii) is applicable, i.e. a is set for o in store st_{i+1}, then the history is extended by a new snapshot. Let $h(a, o)$ be of the form $(o_0, r_0)...(o_l, r_l)$. Then $r_l < m$ and $h' = h[(a, o) \mapsto h(a, o)(st(a, o), m)]$. Let $i < k$. Since the invariant is satisfied before the transition, $st_i(a, o) \neq \bot$ and $st_i(a, o) \neq st_{i+1}(a, o)$ imply that n_{i+1} has a direct follower and it remains to be so after the execution of set. Similarly, if condition of (e) is satisfied before set, then it is satisfied after set and the corresponding object saves the @pre-value before and after considered set operation. Thus (d) and (e) are preserved in this case. Let $i = k$. Since the invariant is satisfied before the execution of set and since $r_l < m$, the value of a for o in the pre-state coincide with its value in st, i.e. $st_k(a, o) = st(a, o)$. (2ii) implies that the consequent of (d) is satisfied for $r_j = m$ as well as the consequent of (e).

Suppose that case (2iii) applies. Let $h(a, o) = (o_0, r_0)...(o_{l-1}, r_{l-1})(o_l, r_l)$ and $h'(a, o) = (o_0, r_0)...(o_{l-1}, r_{l-1})(o_l, m)$ be the histories before and after the transition respectively. (d) is preserved since $r_l < m$ and since (d) holds in the pre-state; i.e. if $\bot \neq st_i(a, o) \neq st_{i+1}(a, o)$, then $n_{i+1} \leqslant r_l < m$, and if $\bot \neq st_k(a, o) \neq st'(a, o)$, then the last witness time-stamp equals m which is the number of currently executing method. Thus (d) is preserved. Suppose that for $i < k$ formula $st_i(a, o) \neq \bot \wedge n_{i+1} \leqslant r_j \wedge (1 \leqslant j - 1 \Rightarrow r_{j-1} < n_{i+1})$ holds. Then from the assumption that the invariant is satisfied in cs follows that $st_i(a, o) = o_j$ and consequently (e) is preserved. Suppose $st_k(a, o) \neq \bot$. From (2iii) follows that $st(a, o) = o_l$. We want to prove that $st_k(a, o) = st(a, o)$. If it was not the case, then r_l would be equal to m due to the assumption that (d) is satisfied in cs. However this would contradict the applicability of (2iii), i.e. the assumption that $r_l < m$.

Suppose that case (2iv) applies. In this case the value of attribute $a@pre$ was stored for the last method call before and the histories are not modified at all. Thus if $\bot \neq st_i(a,o) \neq st_{i+1}(a,o)$, then consequent of (d) is preserved and similarly for (e). For the last two stores, if $\bot \neq st_k(a,o) \neq st'(a,o)$, then since the @pre-value was saved before, the time-stamp of the corresponding snapshot is equal to m and the corresponding value equals $st_k(a,o)$.

Let $cs\, R_{return(op)}\, cs'$. A method return does not change the current store. In our model it replaces only the previous store corresponding to the proceeding method by the most recent one. Let $sh = sh_0(st, op, m)$, for some m, and let $sh' = (st_0, main, n_0)...(st_{k-1}, op_{k-1}, n_{k-1})(st, op_k, n_k)$. Condition (c) is preserved in an obvious way.

First we consider case (3i), i.e. $|sh_0| > 0$. Note that in this case, no attribute history is modified by the return. For $i < k-1$ conditions (d) and (e) follow from the fact that $cs \models Inv$. If $\bot \neq st_{k-1}(a,o) \neq st_k(a,o)$, then let j be the smallest index such that $n_k \leqslant r_j$. Then $o_j = st_{k-1}(a,o)$, and j is the smallest index witnessing that, i.e. r_j directly follows n_k. Thus (d) and (e) are preserved. If $\bot \neq st_{k-1}(a,o) = st_k(a,o) \neq st(a,o)$, then let j be the smallest index such that $n_k \leqslant r_j$. There exists such a j due to the fact that the value of a changes and (d) and (e) are satisfied in cs. Therefore the consequent of (d) is satisfied in cs'. Since no change to a was performed in store st_k, $\bot \neq st_{k-1}(a,o) = st_k(a,o) = o_j$. Thus the consequent of (e) is satisfied. If $st_{k-1}(a,o) = st_k(a,o) = st(a,o)$, then the antecedent of (d) is not satisfied and (d) holds trivially. Moreover, if there is a direct follower r_j of n_k in cs', then it is a direct follower of n_k in cs; the invariant and the fact that a does not change imply that $o_j = st_{k-1}(a,o)$. Thus the consequent of (e) holds.

If (3ii) holds, then = 0; (c), (d) and (e) hold since the histories are emptied and there is no preceding store.

Let $cs\, R_{cleanTop(a,o)}\, cs'$ and let $sh = sh_0(st_{k+1}, op_{k+1}, n_{k+1})$. Clean modifies neither attributes nor the store history. It only modifies/cleans histories of an attribute. Conditions (b) and (c) are preserved in an obvious way. Conditions (d) is preserved, since it concerns existence of followers and clean removes only an outdated follower leaving an earlier one. Similarly (e) is preserved, since clean does not remove direct followers.

Let $cs\, R_{cleanWhole(a,o)}\, cs'$; the preservation of condition (c) follows trivially from the definition of this transition. The preservation of (d) and (e) follows from the fact that we leave in the filtrated history sequence snapshots with time-stamps being the least upper bounds of unterminated calls numbers.

4 Weak Bisimulation

In this section we prove that we can restrict our consideration to relevant method calls, i.e. calls of methods which have a post-condition including an attribute of the form $a@pre$. Consequently, we do not need to archive attribute histories for each and every method call. In this way we can in a sense flatten the structure of method calls. This corresponds to inlining methods as in the case of refactoring [7]. This is an important optimization possibility, since during a method execution several irrelevant methods, such as get and set, can be called. Intuitively it is clear that if a method does not have a post-condition referring to a pre-state, then there is no need to archive separately changes it performs; it is enough to treat them as changes done by the method which called it. We prove that every execution is weakly bisimilar with a flattened one, in which we abstract from calls of irrelevant methods.

We divide the set of methods/operations Op into two parts: ROp containing relevant operations and IOp containing irrelevant operations. We say that a flattened computation state is equivalent to a non-flattened one if there is a one to one correspondence of relevant method calls between the flattened and non-flattened ones and moreover for every call of a relevant operation, the corresponding object store in the flattened state is equal to the last object store following the relevant method call, but proceeding all later relevant calls if there are any. This notion of equivalence abstracts away from the irrelevant calls, making them part of the relevant call by considering only the most recent object store resulting from calls to irrelevant methods. The definition, though conceptually simple, is a bit involved.

Below ih_j denote irrelevant histories, i.e. heap histories consisting of irrelevant method calls and the corresponding object stores. More precisely, $ih_j \in (St \times IOp \times N)^*$. Relevant methods are denoted by rop_j, i.e. $rop_j \in ROp$. In those histories only the last object store counts. Formally, we say that computation state csa is not flattened and that csb is flattened, if condition (I) below is satisfied. We say that a non-flattened state csa and a flattened state csb are equivalent and write $csa \approx csb$ if, and only if, conditions (II) and (III) are satisfied.

(I) $csa{=}((sta_0, main, 0)\,ih_0\,(sta_1, rop_1, na_1)\,ih_1 ... (sta_m, rop_m, na_m)\,ih_m, ha, na)$,
$csb = ((stb_0, main, 0)(stb_1, rop_1, nb_1)...(stb_m, rop_m, nb_m), hb, nb)$
(II) If $|ih_j| > 0$, then the last object store in ih_j equals stb_j, for $j = 0, ..., m$
(III) If $|ih_j| = 0$, then $sta_j = stb_j$, for $j = 0, ..., m$

Condition (I) says that the computation states must contain calls of the same relevant operations, but in the first case they may be followed by calls of irrelevant methods. (II) requires that the last object store in a sequence of irrelevant method calls following a call of a relevant method coincides with the object store corresponding to the corresponding relevant method call in flattened state csb. (III) handles case when the sequence of irrelevant method calls is empty; in this case the object stores corresponding to the relevant method call must coincide. Note that $nb_i \leqslant na_i$.

From the definition of $\approx$ follows the next lemma, since the definition implies that object stores preceding relevant method calls coincide.

Lemma 4. *Let csa, csb be computation states of the form described by condition (I) of the above definition and let $csa \approx csb$. For $j = 1, ..., m$, if op_j is a relevant method, then $sta_j(a@pre, o) = stb_j(a@pre, o)$.*

We prove that $\approx$ is a kind of weak bisimulation. Thus the above lemma and the theorem below imply that for all reachable states we can flatten method calls without information loss about pre-states of relevant methods. We treat calls of clean operations and irrelevant methods as τ actions which are simulated by identity steps. Moreover, we treat here both clean transitions (4) and (5) as one transition, since the proof is the same in both cases.

Theorem 5. ***[Weak Bisimulation]***
$\approx$ is a weak bisimulation relation; i.e. $inSt \approx inSt$, and if $csa \approx csb$, then the following conditions hold:

1. $iop \in IOp \wedge csa \xrightarrow{call(iop)} csa' \Rightarrow csa' \approx csb$
2. $rop \in ROp \wedge csa \xrightarrow{call(rop)} csa' \Rightarrow \exists_{csb'} csb \xrightarrow{call(rop)} csb' \wedge csa' \approx csb'$
3. $rop \in ROp \wedge csb \xrightarrow{call(rop)} csb' \Rightarrow \exists_{csa'} csa \xrightarrow{call(rop)} csa' \wedge csa' \approx csb'$
4. $csa \xrightarrow{set(a,o,o')} csa' \Rightarrow \exists_{csb'} csb \xrightarrow{set(a,o,o')} csb' \wedge csa' \approx csb'$
5. $csb \xrightarrow{set(a,o,o')} csb' \Rightarrow \exists_{csa'} csa \xrightarrow{set(a,o,o')} csa' \wedge csa' \approx csb'$
6. $csa \xrightarrow{clean(a,o)} csa' \Rightarrow csa' \approx csb$
7. $csb \xrightarrow{clean(a,o)} csb' \Rightarrow csa \approx csb'$
8. $iop \in IOp \wedge csa \xrightarrow{return(iop)} csa' \Rightarrow csa' \approx csb$
9. $rop \in ROp \wedge csa \xrightarrow{return(rop)} csa' \Rightarrow \exists_{csb'} csb \xrightarrow{return(rop)} csb' \wedge csa' \approx csb'$
10. $rop \in ROp \wedge csb \xrightarrow{return(rop)} csb' \Rightarrow \exists_{csa_0,\ldots,csa_n,csa'} csa = csa_0 \wedge$
    $csa_i \xrightarrow{return(iop_i)} csa_{i+1} \wedge csa_{i+1} \approx csb$, *for* $i = 0, \ldots, n-1, \wedge$
    $csa_n \xrightarrow{return(rop)} csa' \wedge csa' \approx csb'$

Proof (Sketch). (1) holds trivially, since the last object stores in *csa* and *csb* coincide and calling an operation duplicates the last object store. Similarly, (2) and (3) follow from the fact that the last object stores coincide in *csa* and *csb* and that both computation histories are comparable in respect to $\approx$. (4) and (5) follow from the fact that setting an attribute modifies the last object store only. In case of cleaning, we note that the definition of bisimulation relation does not depend on attributes histories. Consequently (6) and (7) follow in a trivial way, since the definition of $\approx$ does not depend on histories. Return of an operation replaces the object store preceding the last state with the last state. If the operation is irrelevant, then *csb* does not need to be modified in order to be equivalent with *csa*. Thus (8) follows. In case of (9), we note that the return step concerning *csa* must be accompanied by the corresponding return step concerning *csb*, since according to the definition of $\approx$, *csb* must be ready to perform the corresponding return. As in the previous case, the resulting histories coincide.

(10) requires some attention. It may happen that computation history *csa* contains at its end a number of started, but not terminated, irrelevant method calls. If the flattened version, i.e. *csb* performs a return, *csa* needs first to terminate unterminated irrelevant calls following the last call of a relevant method. Those steps can be treated as τ transitions, as described in (1). The resulting states are

bisimilar to *csb*, as proved in case of (1). Finally the relevant method returns. As in case of (9), the resulting states are related by $\approx$. ♦

In a similar way we can prove that archiving attributes values and cleaning attribute histories do not have influence on object attributes. This can be done by defining weak bisimulation relation abstracting away from object histories. We can abstract from histories, since the part of transitions concerning method call and termination as well as attribute modification is defined independently of attribute histories (see Subsection 2.2).

5 Computing @pre

In this section we relate the abstract machine defined in Section 2 to the implementation presented in the paper [9]. We also consider the time- and space-complexity of our method. We demonstrate that the access to @pre-values requires cumulative constant time and that the size of attribute histories is linearly bound by the maximal size of the call-stack.

5.1 Relation to the AspectJ Implementation

In this section we relate the abstract algorithm defined above to the AspectJ implementation proposed in [9]. This implementation was developed prior to the abstract algorithm. Therefore it varies slightly from its formal counterpart. However it could be refactored to achieve much closer correspondence. Due to the space demands, we present only a part of the implementation.

For attribute `a` of type `T`, `SnapshotVal<T>` is the class of `T` snapshots. It contains attributes `val` and `meterReading` storing an attribute value and the corresponding time-stamp. `Stack<SnapshotVal<T>>` is the class of stacks containing snapshots; it corresponds to the set *AH* (see Subsection 2.1). In our implementation this class extends the functionality of `Vector`. It stores objects modified during a relevant method call in attribute `stack`. As usual for stacks it provides methods for popping the stack, for getting the top value and for pushing a value onto its top. In particular, `top()` is a method of class `Stack` returning the top element. Method `subTop()` returns the time-stamp of the snapshot before the last one. Method `clean()` cleans the tops of history-stacks. In contrast to transition *cleanTop* it cleans histories as long as they contain outdated pairs. The implementation does not include *cleanWhole*. This transition requires the use of linked lists, instead of vectors, in the Stack implementation. It can be implemented using three variables referencing elements of the history-stack and one variable referencing elements of the call-stack. These references can be moved down both lists; for every position a bound number of comparisons and removals can be made. This procedure can be implemented in a way guarantying that the time complexity is linear with respect to the length of both stacks. We do not present the implementation of the whole class `Stack`, but only method `clean()`.

```
public void clean() {
  while(size() >= 2 &&
              subTop().meterReading >= Meter.getReading())
  stack.remove(stack.size()-1);
}
```

Class `Meter`, not presented here, manages the numbers of executed method calls (see transitions (1) and (3)). It is based on class `Stack` and provides static method `getReading()` returning the number of currently executing method.

Below we present class `Archive` implementing the core logic for attribute archiving and old value retrieval. `getValueATpre(Stack<SnapshotVal <T>> st, T val)` is a generic method containing logic for computing of @pre-values. `doArchiving(Stack<SnapshotVal<S>> st, S cur)` contains the logic for attribute archiving. Parameter `st` corresponds to an actual attribute history and `cur` to its current value. It should be noted that history-stacks are initialized when the corresponding object is created. It is easy to see, that its condition corresponds to transition (2) of the machine defined in Subsection 2.2.

```
public class Archive {
  static <T> T getValueATpre(Stack<SnapshotVal<T>> st,
                                                    T val) {
    if(st.size() == 0) return val;
    st.clean();
    if(st.topReading() < Meter.getReading()) return val;
    else return st.top().value;
  }
  static <T, S> void doArchiving(Stack<SnapshotVal<S>> st,
                                                      S cur){
    if(st.size() == 0)    //corresponds to transition (2ii)
      st.push(new SnapshotVal<S>(cur, Meter.getReading()));
    else if(Meter.getReading() > st.top().meterReading) {
      if(st.top().value != cur)  //corresponds to (2ii)
        st.push(new SnapshotVal<S>(cur, Meter.getReading()));
      else st.top().meterReading = Meter.getReading();
      //corresponds to transition (2iii)
    }
  }
}
```

If a class `C` contains an attribute `b` of type `T` requiring archiving, then we introduce aspect `ArchiveC` which superimposes on `C` attribute `bHIST` of type `Stack<SnapshotVal<T>>` and method `getBATpre()`. The method is implemented using `getValueATpre()` and `getLastUpdateTime()` returning the last update time. Every manipulation of `b` is detected by pointcut `modB`. If the current meter-reading is larger than 0, meaning that there is a relevant method on the stack, then the archiving is performed by `doArchiving`. If no method with a post-condition is executed, then there is no need for archiving the pre-state.

```
public aspect ArchiveC {
  public Stack<SnapshotVal<T>> Anchor.bHIST =
                       new Stack<SnapshotVal<T>>();
  Element C.getBATpre() { return C.getValueATpre(bHIST, b);
  }
  Integer C.getBLastUpdateTime() {
    return C.getLastUpdateTime(bHIST);
  }
  pointcut modB(C target) : target(target) &&
                                          set(T C.b);
  before(C target) : modB(target) {
    if(Meter.getReading() > 0) {
        Archive.doArchiving(target.bHIST, target.b);
    }
  }
}
```

Lemma 2 implies that method `getValueATpre(Stack<SnapshotVal<T>> st, T val)` is correctly implemented. More precisely, if the if-part of the lemma is satisfied for n_{i+1} being the number of the currently executing method call, then the value stored in the topmost snapshot is returned. This implies that if on the top of history-stack `st` a snapshot is located with a time-stamp larger than or equal to the current one and the previous snapshot, if there is any, has a time-stamp smaller than the current one, then the value stored in the snapshot is the @pre-value. In the other case, `val` is returned. The optimization step allows us to archive attributes for relevant method calls.

5.2 Complexity

In this subsection we discuss the question of time and memory use. We show that the proposed algorithm does not increase the time complexity class of constrained methods and that the access to @pre-values during a post-condition evaluation can be treated as if it needed a constant time. We compare also the space requirements of our approach with the requirements of partially persistent structures.

Our method does not increase time complexity class of instrumented methods since setting an attribute is accompanied by at most one snapshot archiving which requires a bound number of steps. A call of a constrained method results in increasing the call-stack and the call counter. Access to a @pre-value may require removal of some outdated snapshots using method *cleanTop*. Removals require a bound number of steps and there are at most as many snapshots to remove as executions of *set* in the past. Thus the time for removal of an outdated snapshot can be accounted for when treating the execution of *set*. Similarly, we can account for the history removal when the control returns to method `main`.

In case of partially persistent structures, the size of an attribute history is equal to the number of its modifications. The access to old attribute values is logarithmic in respect to the number of attribute updates if binary search trees are used to store old values [5]. In our case, we store only the attribute value prior to a method execution when the attribute is modified for the first time. When a post-condition is evaluated, the access to an @pre-value may require the removal of outdated snapshot from the top of a history-stack (see subsection 5.1). However, this can be accounted for when considering operation *set*. Thus the evaluation can be treated as if the extraction of @pre-values had a constant time.

It should be noted that operation *cleanWhole* can be performed in linear time in respect to the actual stack size k and the history length l (see the previous subsection). Thus, there exists a constant c_1 such that the operation requires not more than $c_1 \cdot (k + l)$ steps. If we start *cleanWhole* only when the length of the history-stack doubles the size of the call-stack, i.e. $l = 2 \cdot k$, then the operation requires at most $c_1 \cdot l \cdot 3/2$ steps. Since afterwards the length of the history is smaller than or equal to k, every outdated snapshot removal requires on average at most $3 \cdot c_1$ steps. Similarly, there is a constant c_2 such that the removal of an outdated snapshot from the top requires at most c_2 steps. We define constant c to be the maximum of c_1 and c_2. c binds the number od steps needed for an outdated snapshot removal.

The use of space is really crucial, since the object reclaim during garbage collection is restrained by history attributes. If an object is stored in a history attribute of another object, then it cannot be deleted before it is removed from the history, or the other object is deleted. Thus, the information about the past should be kept as minimal as possible. During a method execution the corresponding call-stack evolves. For an arbitrary sequence of computation states $inSt$, cs_1,..., cs_m related by transition relation R, let k_0, k_1,..., k_m be the sizes of the corresponding call-stacks (see subsection 2.2) and let k_{max} be the maximal stack size. We show now that for an arbitrary sequence of this form, the cleaning can be performed in a way guarantying that the length of an arbitrary history does not exceed $2 \cdot k_{max}$ without increasing the complexity class of the executed method. This can be ensured by starting operation *cleanWhole* before every execution of $set(a, o, o')$ if the length of the corresponding history doubles the actual stack size, i.e. $|h(a, o)| = 2 \cdot k$. *set* is the only operation which increases the length of a history. Thus, it is guaranteed that at all times for every attribute a and object o, $|h(a, o)| \leqslant 2 \cdot k_{max}$, where k_{max} is the maximal stack size reached before. Unfortunately, we cannot prove in this way that the history length is bound by the actual stack size k times 2, i.e. $|h(a, o)| \leqslant 2 \cdot k$. However, this bound can be ensured using algorithms increasing the time-complexity class of instrumented methods.

We consider now the problem of Hanoi Towers with n rings. The underlying structure can be implemented using three linked lists storing natural numbers sorted increasingly. The standard solution uses a recursive algorithm which moves $n - 1$ rings from the source location to the intermediate one, then moves the last ring to the target location and finally moves rings from the intermediate location to the target one. This requires an exponential number of moves. Therefore, finally the corresponding partially persistent structure has an exponential size order with respect to n. The size of the call-stack is maximally n. Consequently, we can ensure that the sufficiently persistent structure has at most the size order n^2, since there are n rings to move, every rung is linked to at most one other ring and n is the maximal size of the call-stack. At the end of the algorithm execution, the histories can be emptied (see transition (3)), since there is no need to store information about the past forms of the structure.

Note that recursion should be used sparingly due to its high time and memory overhead. Instead one should use iteration. In case of iterative algorithms, our method performs much better than partially persistent structures.

6 Conclusion

In this paper we defined an abstract machine simulating steps of an object-oriented system and managing old attribute values. We defined an invariant and proved that it is preserved by transitions of this machine. Using this invariant we proved that it is possible to compute attribute values from before a method execution using attribute histories. We demonstrated that it is possible to abstract from calls of methods which do not have post-conditions. We related the abstract machine to AspectJ implementation proposed in an earlier paper. Finally, we showed that this machine requires less space to store old values than partially persistent structures. In the future, we are going to investigate how to apply our technique to partially persistent structures and if the space bounds presented in this paper can be improved.

References

1. Barnett, M., M. Leino, K.R., 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)
2. Conchon, S., Fillitre, J.-C.: Semi-persistent Data Structures. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 322–336. Springer, Heidelberg (2008)
3. Dzidek, W., Briand, L., Labiche, Y.: Lessons learned from developing a dynamic OCL constraint enforcement tool for java. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 9–19. Springer, Heidelberg (2006)
4. Darvas, A., Müller, P.: Reasoning About Method Calls in JML Specifications. In: Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'05), Glasgow, Scotland (July 2005)
5. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making Data Structures Persistent. Journal of Computer and System Sciences 38(1) (1989)
6. Floyd, R.W.: Assigning meanings to programs, in Mathematical Aspects of Computer Science. In: Proceedings of Symposium in Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society, Providence (1967)
7. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley, Reading (2000)
8. Hussmann, H., Finger, F., Wiebicke, R.: Using Previous Property Values in OCL Postconditions: An Implementation Perspective. In: Int. Workshop UML 2.0 - The Future of the UML Constraint Language OCL, York, UK (October 2, 2000)

9. Kosiuczenko, P.: On the implementation of @pre. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 246–261. Springer, Heidelberg (2009)
10. Meyer, B.: Applying design by contract. Computer 25(10), 40–51 (1992)
11. Meyer, B.: Eiffel: The Language. Object-Oriented Series. Prentice Hall, New York (1992)
12. OMG, OCL 2.0 Specification, Version 2005-06-06 (June 2005)
13. Toval, A., Requena, V., Fernandez, J.: Emerging OCL Tools. Journal of Software and System Modelling 2(4), 248–261 (2003)

A Tracking Semantics for CSP[⋆]

Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit

Universidad Politécnica de Valencia,
Camino de Vera S/N, E-46022 Valencia, Spain
{mllorens,fjoliver,jsilva,stamarit}@dsic.upv.es

Abstract. CSP is a powerful language for specifying complex concurrent systems. Due to the non-deterministic execution order of processes and to synchronizations, many analyses such as deadlock analysis, reliability analysis, and program slicing try to predict properties of the specification which can guarantee the quality of the final system. These analyses often rely on the use of CSP's traces. In this work, we introduce the theoretical basis for tracking concurrent and explicitly synchronized computations in process algebras such as CSP. Tracking computations is a difficult task due to the subtleties of the underlying operational semantics which combines concurrency, non-determinism and non-termination. We define an instrumented operational semantics that generates as a side-effect an appropriate data structure (a track) which can be used to track computations. Formal definition of a tracking semantics improves the understanding of the tracking process, but also, it allows to formally prove the correctness of the computed tracks.

Keywords: Concurrent Programming, CSP, Semantics, Tracking.

1 Introduction

One of the most important techniques for program understanding and debugging is tracing [3]. A trace gives the user access to otherwise invisible information about a computation. In the context of concurrent languages, computations are particularly complex due to the non-deterministic execution order of processes and to the restrictions imposed on this order by synchronizations; and thus, a tracer is a powerful tool to explore, understand and debug concurrent computations.

One of the most widespread concurrent specification languages is the *Communicating Sequential Processes* (CSP) [7,17] whose operational semantics allows the combination of parallel, non-deterministic and non-terminating processes. The study and transformation of CSP specifications often uses different analyses such as deadlock analysis [10], reliability analysis [8] and program slicing [19] which are based on a data structure able to represent computations.

⋆ This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant ACOMP/2009/017, and by the *Universidad Politécnica de Valencia* (Programs PAID-05-08 and PAID-06-08). Salvador Tamarit was partially supported by the Spanish MICINN under FPI grant BES-2009-015019.

In CSP a trace is a sequence of events. Concretely, the operational semantics of CSP is an event-based semantics in which the occurrence of events fires the rules of the semantics. Hence, the final trace of the computation is the sequence of events occurred (see Chapter 8 of [17] for a detailed study of this kind of traces). In this work we introduce an essentially different notion of trace [3] called track. In our setting, a track is a data structure which represents the sequence of expressions that have been evaluated during the computation, and moreover, this data structure is labelled with the location of these expressions in the specification. Therefore, a CSP track is much more informative than a CSP trace since the former not only contains a lot of information about original program structures but also explicitly relates the sequence of events with the parts of the specification that caused these events.

Example 1. Consider the following CSP specification:[1]

$$\underline{\mathtt{MAIN}} = \underline{\mathtt{CASINO} \parallel} \ \mathtt{GAMBLING}$$

$$\underline{\mathtt{CASINO}} = \underline{(\mathtt{PLAYER} \mid\mid\mid \mathtt{ROULETTE})} \ \underline{\underset{\{\mathtt{betred,red,black,prize}\}}{\parallel}} \ \underline{\mathtt{CROUPIER}}$$

$$\underline{\mathtt{PLAYER}} = \underline{\mathtt{betred} \rightarrow} \ (\underline{\mathtt{prize} \rightarrow \ \mathtt{STOP} \ \square} \ \mathtt{noprize} \rightarrow \mathtt{STOP})$$

$$\underline{\mathtt{ROULETTE}} = \mathtt{red} \rightarrow \mathtt{STOP} \ \underline{\square \ \mathtt{black} \rightarrow \ \mathtt{STOP}}$$

$$\begin{aligned}\underline{\mathtt{CROUPIER}} = \ & (\mathtt{betred} \rightarrow \mathtt{red} \rightarrow \mathtt{prize} \rightarrow \mathtt{STOP}) \\ & \underline{\square \ (\mathtt{betred} \rightarrow \mathtt{black} \rightarrow \mathtt{prize} \rightarrow \ \mathtt{STOP})} \\ & \square \ (\mathtt{betblack} \rightarrow \mathtt{black} \rightarrow \mathtt{prize} \rightarrow \mathtt{STOP}) \\ & \square \ (\mathtt{betblack} \rightarrow \mathtt{red} \rightarrow \mathtt{getmoney} \rightarrow \mathtt{STOP})\end{aligned}$$

$$\mathtt{GAMBLING} = \textit{Complex Composite Processes}$$

This specification models several gambling activities running in parallel and modelled by process `GAMBLING`. One of the games is the casino. A `CASINO` is modelled as the interaction of three parallel processes, namely a `PLAYER`, a `ROULETTE`, and a `CROUPIER`. The player bets for red, and she can win a prize or not. The roulette simply takes a color (either red or black); and the croupier checks the bet and the color of the roulette in order to give a prize to the player or just get the bet money.

This specification contains an error, because it allows the trace of events $t = \langle \mathtt{betred}, \mathtt{black}, \mathtt{prize} \rangle$ where the player bets for red and she wins a prize even though the roulette takes black.

Now assume that we execute the specification and discover the error after executing trace t. A track can be very useful to understand why the error was caused, and what part of the specification was involved in the wrong execution.

[1] We refer those readers non familiar with CSP syntax to Section 2 where we provide a brief introduction to CSP.

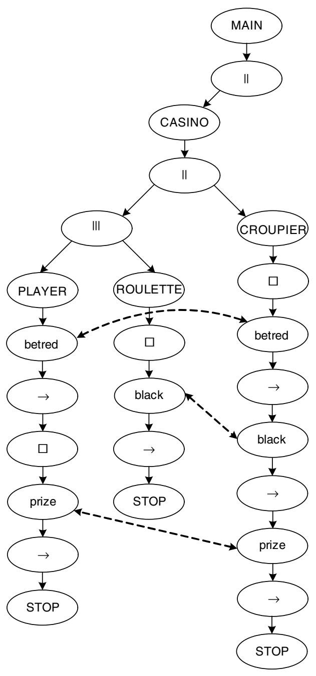


Fig. 1. Track of the program in Example 1

For instance, if we look at the track of Fig. 1, we can easily see that the three processes run in parallel, and that the prize is given because there is a synchronization (dashed edges represent synchronizations) between `CROUPIER` and `PLAYER` that should never happen. Observe that the track is intuitive enough as to be a powerful program comprehension tool that provides much more information than the trace.

Moreover, observe that the track contains explicit information about the specification expressions that were involved in the execution. Therefore, it can be used for program slicing (see [18] for an explanation of the technique and [14] for an adaptation of program slicing to CSP). In particular, in this example, we can use the track to extract the part of the program that was involved in the execution—note that this is the only part that could cause the error—. This part has been underscored in the example. With a quick look, one can see that the underscored part of process `CROUPIER` produced the wrong behavior. Event `prize` should be replaced by `getmoney`.

Another interesting application of tracks is related to component extraction and reuse. If we are interested in a particular trace, and we want to extract the part of the specification that models this trace to be used in another model, we can simply produce a slice, and slightly augment the code to make it syntactically correct (see [14] for an example and an explanation of this transformation). In our example, even though the system is very big due to the process `GAMBLING`, the track is able to extract the only information related to the trace.

We have implemented a tool [13] able to produce tracks and to automatically color parts of the code related to some point in the specification. This tool is integrated in the last version of ProB [11,12] which is the most extended IDE for CSP.

In languages such as Haskell, the tracks (see, e.g., [3,4,5,1]) are the basis of many analysis methods and tools. However, computing CSP tracks is a complex task due to the non-deterministic execution of processes, due to deadlocks, due to non-terminating processes and mainly due to synchronizations. This is probably the reason why no correctness result exists which formally relates the track of a specification to its execution. This semantics is needed because it would allow us to prove important properties (such as correctness and completeness) of the techniques and tools based on tracking.

To the best of our knowledge, there is only one attempt to define and build tracks for CSP [2]. Their notion of track is based on the standard *program dependence graph* [6]; therefore it is useful for program slicing but it is insufficient for other analyses that need a *context-sensitive graph* [9] (i.e., each different process call has a different representation). Moreover, their notion of track does not include synchronizations. Our tracks are able to represent synchronizations, and they are context-sensitive.

The main contributions of this work are the formal definition of tracks, the definition of the first tracking semantics for CSP and the proof that the trace of a computation can be extracted from the track of this computation. Concretely, we instrument the standard operational semantics of CSP in such a way that the execution of the semantics produces as a side-effect the track of the computation. It should be clear that the track of an infinite computation is also infinite. However, we design the semantics in such a way that the track is produced incrementally step by step. Therefore, if the execution is stopped (e.g., by the user because it is non-terminating or because a limit in the size of the track was specified), then the semantics produces the track of the computation performed so far. This semantics can serve as a theoretical foundation for tracking CSP computations because it formally relates the computations of the standard semantics with the tracks of these computations.

The rest of the paper has been organized as follows. Firstly, in Section 2 we recall the syntax and semantics of CSP. In Section 3 we define the concept of track for CSP. Then, in Section 4, we instrument the CSP semantics in such a way that its execution produces as a side-effect the track associated with the performed computation. In Section 5, we present the main results of the paper proving that the instrumented semantics presented is a conservative extension of the standard semantics, its computed tracks are correct and the corresponding trace can be extracted from the track. Finally, Section 6 concludes.

2 The Syntax and Semantics of CSP

In order to make the paper self-contained, we recall in this section the syntax and semantics of CSP. Figure 2 summarizes the syntax constructions used in CSP specifications. A *specification* is viewed as a finite set of process definitions. The left-hand side of each definition is the name of a process, which is defined in the right-hand side (abbrev. *rhs*) by means of an expression that can be a call to another process or a combination of the following operators:

Prefixing. It specifies that event a must happen before process P.

Internal choice. The system chooses non-deterministically to execute one of the two processes P or Q.

External choice. It is identical to internal choice but the choice comes from outside the system (e.g., the user).

Sequential composition. It specifies a sequence of two processes. When the first (successfully) finishes, the second starts.

Synchronized parallelism. Both processes are executed in parallel with a set X of synchronized events. In absence of synchronizations both processes can execute in any order. Whenever a synchronized event $a \in X$ happens in one of the processes it must also happen in the other at the same time. Whenever the set of synchronized events is not specified, it is assumed that processes are synchronized in all common events. A particular case of parallel execution is *interleaving* where no synchronizations exist (i.e., $X = \emptyset$).

Skip. It successfully finishes the current process. It allows us to continue the next sequential process.

Stop. Synonymous with deadlock: It finishes the current process and it does not allow the next sequential process to continue.

$S ::= \{D_1, \ldots, D_m\}$	(Entire specification)	*Domains*
		$M, N \ldots \in \mathcal{N}$ (Process names)
		$P, Q \ldots \in \mathcal{P}$ (Processes)
		$a, b \ldots \in \Sigma$ (Events)
$D ::= N = P$	(Process definition)	
$P ::= M$	(Process call)	
$\mid\ a \rightarrow P$	(Prefixing)	
$\mid\ P \sqcap Q$	(Internal choice)	
$\mid\ P \, \square \, Q$	(External choice)	
$\mid\ P \,;\, Q$	(Sequential composition)	
$\mid\ P \underset{X}{\parallel} Q$	(Synchronized parallelism)	where $X \subseteq \Sigma$
$\mid\ SKIP$	(Skip)	
$\mid\ STOP$	(Stop)	

Fig. 2. Syntax of CSP specifications

We now recall the standard operational semantics of CSP as defined by Roscoe [17]. It is presented in Fig. 3 as a logical inference system. A *state* of the semantics is a process to be evaluated called the *control*. The system starts with an initial state, and the rules of the semantics are used to infer how this state evolves. When no rules can be applied to the current state, the computation finishes. The rules of the semantics change the states of the computation due to the occurrence of events. The set of possible events is $\Sigma \cup \{\tau, \checkmark\}$. Events in $\Sigma = \{a, b, c \ldots\}$ are visible from the external environment, and can only happen with its co-operation (e.g., actions of the user). The special event τ cannot be observed from outside the system and it is an internal event that happens automatically as defined by the semantics. $\checkmark$ is a special event representing the successful termination of a process. We use the special symbol Ω to denote any process that successfully terminated.

In order to perform computations, we construct an initial state (e.g., `MAIN`) and (non-deterministically) apply the rules of Fig. 3. The intuitive meaning of each rule is the following:

(Process Call). The call is unfolded and the right-hand side of process named N is added to the control.

(Prefixing). When event a occurs, process P is added to the control.

(SKIP). After `SKIP`, the only possible event is $\checkmark$, which denotes the successful termination of the (sub)computation with the special symbol Ω. There is no rule for Ω (neither for `STOP`), hence, this (sub)computation has finished.

(Internal Choice 1 and 2). The system, with the occurrence of the internal event τ, (non-deterministically) selects one of the two processes P or Q which is added to the control.

(External Choice 1, 2, 3 and 4). The occurrence of τ develops one of the branches. The occurrence of an event $a \neq \tau$ is used to select one of the two processes P or Q and the control changes according to the event.

(Sequential Composition 1). In $P;Q$, P can evolve to P' with any event except $\checkmark$. Hence, the control becomes $P';Q$.

(Sequential Composition 2). When P successfully finishes (with event $\checkmark$), Q starts. Note that $\checkmark$ is hidden from outside the whole process becoming τ.

(Synchronized Parallelism 1 and 2). When event $a \notin X$ or events τ or $\checkmark$ happen, one of the two processes P or Q evolves accordingly, but only a is visible from outside the parallelism operator.

(Synchronized Parallelism 3). When event $a \in X$ happens, it is required that both processes synchronize, P and Q are executed at the same time and the control becomes $P' \underset{X}{||} Q'$.

(Synchronized Parallelism 4). When both processes have successfully terminated the control becomes Ω, performing the event $\checkmark$.

We illustrate the semantics with the following example.

(Process Call) $\frac{}{N \xrightarrow{\tau} rhs(N)}$	(Prefixing) $\frac{}{(a \rightarrow P) \xrightarrow{a} P}$	(SKIP) $\frac{}{\texttt{SKIP} \xrightarrow{\checkmark} \Omega}$
(Internal Choice 1) $\frac{}{(P \sqcap Q) \xrightarrow{\tau} P}$	(Internal Choice 2) $\frac{}{(P \sqcap Q) \xrightarrow{\tau} Q}$	
(External Choice 1) $\frac{P \xrightarrow{\tau} P'}{(P \,\Box\, Q) \xrightarrow{\tau} (P' \,\Box\, Q)}$	(External Choice 2) $\frac{Q \xrightarrow{\tau} Q'}{(P \,\Box\, Q) \xrightarrow{\tau} (P \,\Box\, Q')}$	
(External Choice 3) $\frac{P \xrightarrow{e} P'}{(P \,\Box\, Q) \xrightarrow{e} P'}$	(External Choice 4) $\frac{Q \xrightarrow{e} Q'}{(P \,\Box\, Q) \xrightarrow{e} Q'}$	$e \in \Sigma \cup \{\checkmark\}$
(Sequential Composition 1) $\frac{P \xrightarrow{e} P'}{(P;Q) \xrightarrow{e} (P';Q)}$ $e \in \Sigma \cup \{\tau\}$	(Sequential Composition 2) $\frac{P \xrightarrow{\checkmark} \Omega}{(P;Q) \xrightarrow{\tau} Q}$	
(Synchronized Parallelism 1) $\frac{P \xrightarrow{e'} P'}{(P \underset{X}{\parallel} Q) \xrightarrow{e} (P' \underset{X}{\parallel} Q)}$	(Synchronized Parallelism 2) $\frac{Q \xrightarrow{e'} Q'}{(P \underset{X}{\parallel} Q) \xrightarrow{e} (P \underset{X}{\parallel} Q')}$	$(e = e' = a \ \wedge \ a \notin X)$ $\vee \ (e = \tau \ \wedge \ e' \in \{\tau, \checkmark\})$
(Synchronized Parallelism 3) $\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{(P \underset{X}{\parallel} Q) \xrightarrow{a} (P' \underset{X}{\parallel} Q')}$ $a \in X$	(Synchronized Parallelism 4) $\frac{}{(\Omega \underset{X}{\parallel} \Omega) \xrightarrow{\checkmark} \Omega}$	

Fig. 3. CSP's operational semantics

Example 2. Consider the next CSP specification:

$$\texttt{MAIN} \ = \ (\texttt{a} \rightarrow \ \texttt{STOP}) \underset{\{\texttt{a}\}}{\parallel} (\texttt{P} \ \Box \ (\texttt{a} \rightarrow \ \texttt{STOP}))$$
$$\texttt{P} \ = \ \texttt{b} \rightarrow \ \texttt{SKIP}$$

If we use MAIN as the initial state to execute the semantics, we get the computation shown in Fig. 4 where the final state is $((\texttt{a} \rightarrow \texttt{STOP}) \underset{\{\texttt{a}\}}{\parallel} \Omega)$. This computation corresponds to the execution of the left branch of the choice (i.e., P) and thus only event b occurs. Each rewriting step is labelled with the applied rule, and the example should be read from top to bottom.

3 Tracking Computations

In this section we define the notion of track. Firstly, we introduce some notation that will be used throughout the paper.

$$\text{(Process Call)}\ \frac{}{\mathtt{MAIN} \xrightarrow{\tau} ((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel} (\mathtt{P}\Box(\mathtt{a} \rightarrow \mathtt{STOP})))}$$

$$\text{(Synchronized Parallelism 2)}\ \frac{\text{(External Choice 1)}\ \dfrac{\text{(Process Call)}\ \dfrac{}{\mathtt{P} \xrightarrow{\tau} (\mathtt{b} \rightarrow \mathtt{SKIP})}}{(\mathtt{P}\Box(\mathtt{a} \rightarrow \mathtt{STOP})) \xrightarrow{\tau} ((\mathtt{b} \rightarrow \mathtt{SKIP})\Box(\mathtt{a} \rightarrow \mathtt{STOP}))}}{((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel} (\mathtt{P}\Box(\mathtt{a} \rightarrow \mathtt{STOP}))) \xrightarrow{\tau} State_1}$$

$$\text{where } State_1 = ((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel} ((\mathtt{b} \rightarrow \mathtt{SKIP})\Box(\mathtt{a} \rightarrow \mathtt{STOP})))$$

$$\text{(Synchronized Parallelism 2)}\ \frac{\text{(External Choice 3)}\ \dfrac{\text{(Prefixing)}\ \dfrac{}{(\mathtt{b} \rightarrow \mathtt{SKIP}) \xrightarrow{\mathtt{b}} \mathtt{SKIP}}}{((\mathtt{b} \rightarrow \mathtt{SKIP})\Box(\mathtt{a} \rightarrow \mathtt{STOP})) \xrightarrow{\mathtt{b}} \mathtt{SKIP}}}{State_1 \xrightarrow{\mathtt{b}} ((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel} \mathtt{SKIP})}$$

$$\text{(Synchronized Parallelism 2)}\ \frac{(\mathtt{SKIP})\ \dfrac{}{\mathtt{SKIP} \xrightarrow{\checkmark} \Omega}}{((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel} \mathtt{SKIP}) \xrightarrow{\tau} ((\mathtt{a} \rightarrow \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel} \Omega)}$$

Fig. 4. A computation with the operational semantics in Fig. 3

A track is formed by the sequence of expressions that are evaluated during an execution. These expressions are conveniently connected to form a graph. However, several program analysis techniques such as program slicing make use of the locations of program expressions, and thus, this notion of track is insufficient for them. Therefore, we want our tracks to also store the location of each literal (i.e., events, operators and process names) in the specification so that the track can be used to know what portions of the source code have been executed and in what order. The inclusion of source positions in the track implies an additional level of complexity in the semantics, but the benefits of providing our tracks with this additional information are clear and, for some applications, essential. Therefore, we use labels (that we call *specification positions*) to uniquely identify each literal in a specification which roughly corresponds to nodes in the CSP specification's abstract syntax tree. We define a function $\mathcal{P}os$ to obtain the specification position of an element of a CSP specification and it is defined over nodes of an abstract syntax tree for a CSP specification. Formally,

Definition 1. *(Specification position) A* specification position *is a pair* (N, w) *where* $N \in \mathcal{N}$ *and* w *is a sequence of natural numbers (we use* Λ *to denote the empty sequence). We let* $\mathcal{P}os(o)$ *denote the specification position of an expression* o*. Each process definition* $N = P$ *of a CSP specification is labelled with specification positions. The specification position of its left-hand side is* $\mathcal{P}os(N) = (N, 0)$*. The right-hand side is labelled with the call* $\mathtt{AddSpPos}(P, (N, \Lambda))$*; where function* `AddSpPos` *is defined as follows:*

$$\texttt{AddSpPos}(P,(N,w)) = \begin{cases} P_{(N,w)} & \mathit{if}\ \ P \in \mathcal{N} \\ STOP_{(N,w)} & \mathit{if}\ \ P = STOP \\ SKIP_{(N,w)} & \mathit{if}\ \ P = SKIP \\ a_{(N,w.1)} \rightarrow_{(N,w)} \texttt{AddSpPos}(Q,(N,w.2)) & \mathit{if}\ \ P = a \rightarrow Q \\ \texttt{AddSpPos}(Q,(N,w.1))\ op_{(N,w)}\ \texttt{AddSpPos}(R,(N,w.2)) & \\ & \mathit{if}\ \ P = Q\ op\ R\ \ \forall\ op \in \{\sqcap, \square, ||, ;\} \end{cases}$$

Example 3. Consider again the CSP specification in Example 2 where literals are labelled with their associated specification positions (they are underlined) so that labels are unique:

$$\texttt{MAIN}_{\underline{(\texttt{MAIN},0)}} = (\texttt{a}_{\underline{(\texttt{MAIN},1.1)}} \rightarrow_{\underline{(\texttt{MAIN},1)}} \texttt{STOP}_{\underline{(\texttt{MAIN},1.2)}}) \underset{\{\texttt{a}\}}{||}_{\underline{(\texttt{MAIN},\Lambda)}}$$
$$(\texttt{P}_{\underline{(\texttt{MAIN},2.1)}} \square_{\underline{(\texttt{MAIN},2)}} (\texttt{a}_{\underline{(\texttt{MAIN},2.2.1)}} \rightarrow_{\underline{(\texttt{MAIN},2.2)}} \texttt{STOP}_{\underline{(\texttt{MAIN},2.2.2)}}))$$
$$\texttt{P}_{\underline{(\texttt{P},0)}} = \texttt{b}_{\underline{(\texttt{P},1)}} \rightarrow_{\underline{(\texttt{P},\Lambda)}} \texttt{SKIP}_{\underline{(\texttt{P},2)}}$$

In the following, specification positions will be represented with greek letters ($\alpha, \beta, \ldots$) and we will often use indistinguishably an expression and its associated specification position when it is clear from the context (e.g., in Example 3 we will refer to $(\texttt{P},\texttt{1})$ as $\texttt{b}$).

In order to introduce the formal definition of track, we need first to define the concept of *control-flow*, which refers to the order in which the individual literals of a CSP specification are executed. Intuitively, the control can pass from a specification position α to a specification position β iff an execution exists where α is executed before β. This notion of control-flow is similar to the control-flow used in the *control-flow graphs* (CFG) [18] of imperative programming. We have adapted the same idea to CSP where choices and parallel composition appear; and in a similar way to the CFG, we use this definition to draw control arcs in our tracks. Formally,

Definition 2. *(Static control-flow) Given a CSP specification $\mathcal{S}$ and two specification positions α, β in $\mathcal{S}$, we say that the control* can pass from α to β*, denoted by $\alpha \Rightarrow \beta$, iff one of the following conditions holds:*

i) $\alpha = N \ \wedge\ \beta = \mathit{first}((N,\Lambda))$ with $N = rhs(N) \in \mathcal{S}$
ii) $\alpha \in \{\sqcap, \square, ||\} \ \wedge\ \beta \in \{\mathit{first}(\alpha.1), \mathit{first}(\alpha.2)\}$
iii) $\alpha \in \{\rightarrow, ;\} \ \wedge\ \beta = \mathit{first}(\alpha.2)$
iv) $\alpha = \beta.1 \ \wedge\ \beta = \rightarrow$
v) $\alpha \in \mathit{last}(\beta.1) \ \wedge\ \beta =\ ;$

where $\mathit{first}(\alpha)$ is the specification position of the subprocess denoted by α which must be executed first:

$$\mathit{first}(\alpha) = \begin{cases} \alpha.1 & \text{if } \alpha = \ \rightarrow \\ \mathit{first}(\alpha.1) & \text{if } \alpha = \ ; \\ \alpha & \text{otherwise} \end{cases}$$

and $last(\alpha)$ is the set of all possible termination points of the subprocess denoted by α:

$$last(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha = \texttt{SKIP} \\ \emptyset & \text{if } \alpha = \texttt{STOP} \vee \\ & (\alpha \in \{||\} \wedge (last(\alpha.1) = \emptyset \vee last(\alpha.2) = \emptyset)) \\ last(\alpha.1) \cup last(\alpha.2) & \text{if } \alpha \in \{\sqcap, \Box\} \vee \\ & (\alpha \in \{||\} \wedge last(\alpha.1) \neq \emptyset \wedge last(\alpha.2) \neq \emptyset) \\ last(\alpha.2) & \text{if } \alpha \in \{\rightarrow, ;\} \\ last((N, \Lambda)) & \text{if } \alpha = N \end{cases}$$

For instance, in Example 3, we can see how the control can pass from a specification position to another one, e.g., we have $(\texttt{MAIN}, 2) \Rightarrow (\texttt{MAIN}, 2.1)$ and $(\texttt{MAIN}, 2) \Rightarrow (\texttt{MAIN}, 2.2.1)$ due to rule ii). And $(\texttt{MAIN}, 2.2.1) \Rightarrow (\texttt{MAIN}, 2.2)$ due to rule iv); $(\texttt{MAIN}, 2.2) \Rightarrow (\texttt{MAIN}, 2.2.2)$ due to rule iii) and $(\texttt{MAIN}, 2.1) \Rightarrow (\texttt{P}, 1)$ due to rule i).

We also need to define the notions of *rewriting step* and *derivation.*

Definition 3. *(Rewriting Step, Derivation) Given a CSP process P, a* rewriting step *for P, denoted by $P \overset{\Theta}{\rightsquigarrow} P'$, is the transformation of P into P' by using a rule of the CSP semantics. Therefore, $P \overset{\Theta}{\rightsquigarrow} P'$ iff a rule of the form $\dfrac{\Theta}{P \xrightarrow{e} P'}$ is applicable, where $e \in \Sigma \cup \{\tau, \checkmark\}$ and Θ is a (possibly empty) set of rewriting steps. Given a CSP process P_0, we say that the sequence $P_0 \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} P_{n+1}$, $n \geq 0$, is a* derivation *of P_0 iff $\forall\ i, 0 \leq i \leq n, P_i \overset{\Theta_i}{\rightsquigarrow} P_{i+1}$ is a rewriting step. We say that the derivation is* complete *iff there is no possible rewriting step for P_{n+1}. We say that the derivation has* successfully finished *iff P_{n+1} is Ω.*

For instance, in Fig. 5(a), one (possible) complete derivation of Example 3 is shown (for the time being, the reader can ignore the underlined part). The rules applied in each rewriting step (ignoring subderivations) are (Process Call) and (Synchronized Parallelism 3) (abbrev. (PC) and (SP3), respectively).

Function *last* of Definition 2 can be used to determine the last specification position in a derivation. However, this function computes all possible final specification positions, and a derivation only reaches (non-deterministically) a set of them. Therefore, we will use in the following a modified version of *last* called *last'* whose behaviour is exactly the same as *last* except in the case of choices where only one of the branches is selected:

For each derivation $(P \sqcap P' \overset{\Theta}{\rightsquigarrow} P)$ or $(P \ \Box \ P' \overset{\Theta_0}{\rightsquigarrow} \dots \overset{\Theta_n}{\rightsquigarrow} P'', n \geq 0$ such that $P \overset{\Theta'_0}{\rightsquigarrow} \dots \overset{\Theta'_m}{\rightsquigarrow} P'', m \geq 0)$, $last'(P \sqcap P') = last'(P \ \Box \ P') = last'(P)$.

Note that, while *last* is static, *last'* is dynamic; it is defined in the context of a particular derivation which implies one particular way of resolving any non-determinism. The same happens with the definition of control-flow. Control-flow is defined statically and says whether the control can pass from α to β in some derivation. However, the track is a dynamic structure produced for a particular derivation. Therefore, we produce a dynamic version of the definition of control-flow which is defined for a particular derivation.

Definition 4. *(Dynamic control-flow) Let $\mathcal{S}$ be a CSP specification and $\mathcal{D}$ a derivation in $\mathcal{S}$. Given two specification positions α, β in $\mathcal{S}$, we say that the control* can dynamically pass from α to β*, denoted by $\alpha \Rrightarrow \beta$, iff the control can pass from α* to β *($\alpha \Rightarrow \beta$) in derivation $\mathcal{D}$. For each $P \overset{\Theta}{\rightsquigarrow} P' \in \mathcal{D}$ and for all rewriting steps in Θ, we have that:*

1. *if P is a prefixing $(a \rightarrow Q)$ or a sequential composition $(Q; R)$, then $\mathcal{P}os(a) \Rrightarrow \mathcal{P}os(\rightarrow)$ or $\forall p \in last'(Q)$, $\mathcal{P}os(p) \Rrightarrow \mathcal{P}os(;)$ respectively,*
2. *if $P \Rightarrow first(P'')$ where $P'' \overset{\Theta'}{\rightsquigarrow} P''' \in \Theta$, then $\mathcal{P}os(P) \Rrightarrow \mathcal{P}os(first(P''))$,*
3. *if $P \Rightarrow first(P')$, then $\mathcal{P}os(P) \Rrightarrow \mathcal{P}os(first(P'))$.*

Clauses 1, 2 and 3 define the cases in which the control passes between two specification positions in a given derivation. In clause 1, if we have a prefixing in the control then Θ is empty and the rewriting step applied is of the form $\dfrac{}{(a \rightarrow P) \xrightarrow{a} P}$. In this case, clause 1 guarantees that the control can dynamically pass from a to $\rightarrow$; and clause 3 guarantees that the control can dynamically pass from $\rightarrow$ to P. However, in general, Θ is not empty, and the rewriting step is of the form $\dfrac{P'' \longrightarrow P'''}{P \longrightarrow P'}$. Here, clause 2 ensures that the control can dynamically pass from P to P''; and clause 3 ensures that the control can dynamically pass from P to P' and from P'' to P'''. For instance, it is possible that we have a rewriting step to evaluate the process $P \,\Box\, P'$. Clearly, the control can pass from $\Box$ to both P and P' ($\Box \Rightarrow P$ and $\Box \Rightarrow P'$), but in the rewriting step the control will only pass to one of them ($\Box \Rrightarrow P$ or $\Box \Rrightarrow P'$). In this case, clauses 2 and 3 are used.

We are now in a position to formally define the concept of *track* of a derivation.

Definition 5. *(Track) Given a CSP specification $\mathcal{S}$, and a derivation $\mathcal{D}$ in $\mathcal{S}$, the* track *of $\mathcal{D}$ is a graph $\mathcal{G} = (N, E_c, E_s)$ where N is a set of nodes uniquely identified with a natural number and that are labelled with specification positions ($l(n)$ refers to the* label *of node n), and edges are divided into two groups:*

- control-flow edges *(E_c) are a set of one-way edges (denoted with $\mapsto$) representing the control-flow between two nodes, and*
- synchronization edges *(E_s) are a set of two-way edges (denoted with $\leftrightarrow$) representing the synchronization of two (event) nodes;*

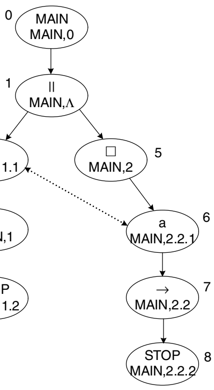
and

1. *E_c contains a control-flow edge $a \mapsto a'$ iff $a \Rrightarrow a'$ with respect to $\mathcal{D}$, and*
2. *E_s contains a synchronization edge $a \leftrightarrow a'$ for each synchronization occurring in $\mathcal{D}$ where a and a' are the nodes of the synchronized events.*

The only nodes in N are the nodes induced by E_c and E_s.

$$\begin{aligned}
\texttt{MAIN} &\underset{\textsf{(PC)}}{\rightsquigarrow} (\texttt{a} \rightarrow \texttt{STOP}) \underset{\{\texttt{a}\}}{\parallel} (\texttt{P}\Box(\texttt{a} \rightarrow \texttt{STOP}))\\
&\underset{\textsf{(SP3)}}{\rightsquigarrow} \texttt{STOP} \underset{\{\texttt{a}\}}{\parallel} \texttt{STOP}\\
&\underset{\underline{\textsf{(SP1)}}}{\rightsquigarrow} \underline{\bot} \underset{\underline{\{\texttt{a}\}}}{\parallel} \underline{\texttt{STOP}}\\
&\underset{\underline{\textsf{(SP2)}}}{\rightsquigarrow} \underline{\bot} \underset{\underline{\{\texttt{a}\}}}{\parallel} \underline{\bot}
\end{aligned}$$

(a) Derivation



(b) Track

Fig. 5. Derivation and track associated with the specification of Example 3

Example 4. Consider again the specification of Example 3. We show in Fig. 5(a) one possible derivation (ignoring subderivations) of this specification (for the time being, the underlined part should be ignored). Its associated track is shown in Fig. 5(b). In the example, we see that the track is a connected and directed graph. Apart from the control-flow edges, there is one synchronization edge between nodes $(\texttt{MAIN}, \texttt{1.1})$ and $(\texttt{MAIN}, \texttt{2.2.1})$ representing the synchronization of event $\texttt{a}$. To illustrate the inclusion of edges in Definition 5, we see that the edge between nodes 2 and 3 is introduced according to clause 1 of Definition 4; the edge between nodes 5 and 6 is introduced according to clause 2 of Definition 4 because, in the subderivations of (SP3), there is a rewriting step $\textsf{(External Choice 4)}\ \dfrac{\textsf{(Prefixing)}\dfrac{}{(\texttt{a} \rightarrow \texttt{STOP}) \xrightarrow{\texttt{a}} (\texttt{STOP})}}{(\texttt{P}\Box(\texttt{a} \rightarrow \texttt{STOP})) \xrightarrow{\texttt{a}} (\texttt{STOP})}$ and $\mathit{first}(\texttt{a} \rightarrow \texttt{STOP}) = \texttt{a}$; the edge between nodes 7 and 8 is introduced according to clause 3 of Definition 4 because there is also a rewriting step $\textsf{(Prefixing)}\dfrac{}{(\texttt{a} \rightarrow \texttt{STOP}) \xrightarrow{\texttt{a}} (\texttt{STOP})}$ and $\mathit{first}(\texttt{STOP}) = \texttt{STOP}$; and the synchronization edge between nodes 2 and 6 is introduced according to clause 2 of Definition 5.

The trace associated with the derivation in Fig. 5(a) is $\langle \texttt{a} \rangle$. Therefore, note that the track is much more informative: it shows the exact processes that have been evaluated with an explicit causality relation; and, in addition, it shows the specification positions that have been evaluated and in what order.

4 Instrumenting the Semantics for Tracking

The generation of tracks in CSP introduces new challenges such as non-deterministic execution of processes, deadlocks, non-terminating processes and synchronizations. In this work, we design a solution that overcomes these difficulties.

Firstly, we generate tracks with an augmented semantics which is conservative with respect to the standard operational semantics. Therefore, the execution order is the standard order, thus non-determinism and synchronizations are solved by the semantics. Moreover, the semantics generates the track incrementally, step by step. Therefore, infinite computations can be tracked until they are stopped. Hence, it is not needed to actually finish a computation to get the track of the subcomputations performed.

Example 5. In the following CSP specification two non-terminating processes run in parallel and synchronize infinitely.

$$\mathtt{MAIN}_{\underline{(\mathtt{MAIN},0)}} = \mathtt{P}_{\underline{(\mathtt{MAIN},1)}} \parallel_{\{\mathtt{a}\}}{}_{\underline{(\mathtt{MAIN},\Lambda)}} \mathtt{P}_{\underline{(\mathtt{MAIN},2)}}$$
$$\mathtt{P}_{\underline{(\mathtt{P},0)}} = \mathtt{a}_{\underline{(\mathtt{P},1)}} \rightarrow_{\underline{(\mathtt{P},\Lambda)}} \mathtt{P}_{\underline{(\mathtt{P},2)}}$$

Because the computation is infinite, the track (shown in Fig. 6) is also infinite.

In order to solve the problem of deadlocks (that stop the computation), and have a representation for them in the tracks; when a deadlock happens, the semantics performs some additional steps to be able to generate a part of the track that represents the deadlock. These additional steps do not influence the other rules of the semantics, thus it remains conservative.

This section introduces an instrumented operational semantics of CSP which generates as a side-effect the tracks associated with the computations performed with the semantics. The tracking semantics is shown in Fig. 7, where we assume that every literal in the program has been labelled with its specification

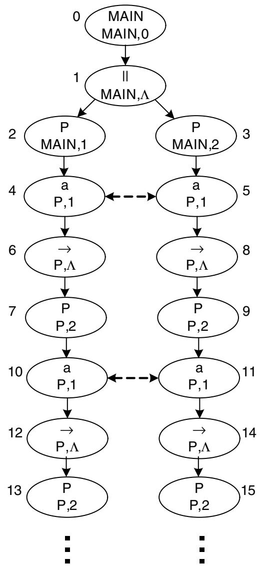


Fig. 6. Track of the program in Example 5

position (denoted by a subscript, e.g., P_α). In this semantics, a *state* is a tuple (P, G, m, Δ), where P is the process to be evaluated (the *control*), G is a directed graph (i.e., the track built so far), m is a numeric reference to the current node in G, and Δ is a set of references to nodes that may be synchronized. Concretely, m references the node in G where the specification position of the control P must be stored. Reference m is a fresh[2] reference generated to add new nodes to G. The basic idea of the graph construction is to record the current control with the current reference in every step by connecting it to its parent. We use the notation $G[m \underset{n}{\mapsto} \alpha]$ to introduce a node in G. For instance, if we are adding a node to G this new node has reference m, it is labelled with specification position α, and its successor is n (a fresh reference). Successor arrows are denoted by $m \underset{n}{\mapsto}$ which means that node n is the successor of node m. Every time an event in Σ happens during the computation, this event is stored in the set Δ of the current state. Therefore, when a synchronized parallelism is evaluated, all the events that must be synchronized are in Δ. We use the special symbol $\perp$ to denote any process that is deadlocked. In order to perform computations, we construct an initial state (e.g., $(\texttt{MAIN}, \emptyset, 0, \emptyset)$) and (non-deterministically) apply the rules of Fig. 7. When the execution has finished or has been interrupted, the semantics has produced the track of the computation performed so far.

An explanation for each rule of the semantics follows:

(Process Call). The called process N is unfolded, node m is added to the graph with specification position α and successor n (a fresh reference). The new process in the control is $rhs(N)$. The set Δ of events to be synchronized is put to $\emptyset$.

(Prefixing). This rule adds nodes m (the prefix) and n (the prefixing operator) to the graph. In the new state, n becomes the parent reference and the fresh reference p represents the current reference. The new control is P. The set Δ is $\{m\}$ to indicate that event a has occurred and it must be synchronized when required by (Synchronized Parallelism 3).

(SKIP and STOP). Whenever one of these rules is applied, the subcomputation finishes because Ω (for rule `SKIP`) and $\perp$ (for rule `STOP`) are put in the control, and these special symbols have no associated rule. A node with the `SKIP` (respectively `STOP`) specification position is added to the graph.

(Internal Choice 1 and 2). The choice operator is added to the graph, and the (non-deterministically) selected branch is put into the control with the fresh reference n as the successor of the choice operator.

(External Choice 1, 2, 3 and 4). External choices can develop both branches while τ events happen (rules 1 and 2), until an event in $\Sigma \cup \{\checkmark\}$ occurs (rules 3 and 4). This means that the semantics can add nodes to both branches of the track alternatively, and thus, it needs to store the next reference to use in every branch of the choice. This is done by labelling choice operators with a tuple of the form (α, n_1, n_2) where α is the specification position of the choice operator; and n_1 and n_2 are respectively the references to be used

[2] We assume that fresh references are numeric and generated incrementally.

(Process Call)

$$\frac{}{(N_\alpha, G, m, \Delta) \xrightarrow{\tau} (rhs(N), G[m \underset{n}{\mapsto} \alpha], n, \emptyset)}$$

(Prefixing)

$$\frac{}{(a_\alpha \rightarrow_\beta P, G, m, \Delta) \xrightarrow{a} (P, G[m \underset{n}{\mapsto} \alpha, n \underset{p}{\mapsto} \beta], p, \{m\})}$$

(SKIP)

$$\frac{}{(\mathtt{SKIP}_\alpha, G, m, \Delta) \xrightarrow{\checkmark} (\Omega, G[m \underset{n}{\mapsto} \alpha], n, \emptyset)}$$

(STOP)

$$\frac{}{(\mathtt{STOP}_\alpha, G, m, \Delta) \xrightarrow{\tau} (\bot, G[m \underset{n}{\mapsto} \alpha], n, \emptyset)}$$

(Internal Choice 1)

$$\frac{}{(P \sqcap_\alpha Q, G, m, \Delta) \xrightarrow{\tau} (P, G[m \underset{n}{\mapsto} \alpha], n, \emptyset)}$$

(Internal Choice 2)

$$\frac{}{(P \sqcap_\alpha Q, G, m, \Delta) \xrightarrow{\tau} (Q, G[m \underset{n}{\mapsto} \alpha], n, \emptyset)}$$

(External Choice 1)

$$\frac{(P_1, G', n', \Delta) \xrightarrow{\tau} (P', G'', n'', \emptyset)}{(P_1 \,\square_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{\tau} (P' \,\square_{(\alpha,n'',n_2)} P_2, G'', m, \emptyset)}$$

where $(G', n') = \mathtt{FirstEval}(G, n_1, m, \alpha)$

(External Choice 2)

$$\frac{(P_2, G', n', \Delta) \xrightarrow{\tau} (P', G'', n'', \emptyset)}{(P_1 \,\square_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{\tau} (P_1 \,\square_{(\alpha,n_1,n'')} P', G'', m, \emptyset)}$$

where $(G', n') = \mathtt{FirstEval}(G, n_2, m, \alpha)$

(External Choice 3)

$$\frac{(P_1, G', n', \Delta) \xrightarrow{e} (P', G'', n'', \Delta')}{(P_1 \,\square_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{e} (P', G'', n'', \Delta')} \quad e \in \Sigma \cup \{\checkmark\}$$

where $(G', n') = \mathtt{FirstEval}(G, n_1, m, \alpha)$

(External Choice 4)

$$\frac{(P_2, G', n', \Delta) \xrightarrow{e} (P', G'', n'', \Delta')}{(P_1 \,\square_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{e} (P', G'', n'', \Delta')} \quad e \in \Sigma \cup \{\checkmark\}$$

where $(G', n') = \mathtt{FirstEval}(G, n_2, m, \alpha)$

(Sequential Composition 1)

$$\frac{(P, G, m, \Delta) \xrightarrow{e} (P', G', m', \Delta')}{(P; Q, G, m, \Delta) \xrightarrow{e} (P'; Q, G', m', \Delta')} \quad e \in \Sigma \cup \{\tau\}$$

(Sequential Composition 2)

$$\frac{(P, G, m, \Delta) \xrightarrow{\checkmark} (\Omega, G', n, \emptyset)}{(P ;_\alpha Q, G, m, \Delta) \xrightarrow{\tau} (Q, G'[n \underset{p}{\mapsto} \alpha], p, \emptyset)}$$

(Synchronized Parallelism 1)

$$\frac{(P_1, G', n', \Delta) \xrightarrow{e'} (P', G'', n'', \Delta')}{(P_1 \underset{X}{\|}_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{e} (P' \underset{X}{\|}_{(\alpha,n'',n_2)} P_2, G'', m, \Delta')} \quad \begin{array}{l}(e = e' = a \ \wedge \ a \notin X) \\ \vee\ (e = \tau \ \wedge \ e' \in \{\tau, \checkmark\})\end{array}$$

where $(G', n') = \mathtt{FirstEval}(G, n_1, m, \alpha)$

(Synchronized Parallelism 2)

$$\frac{(P_2, G', n', \Delta) \xrightarrow{e'} (P', G'', n'', \Delta')}{(P_1 \underset{X}{\|}_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{e} (P_1 \underset{X}{\|}_{(\alpha,n_1,n'')} P', G'', m, \Delta')} \quad \begin{array}{l}(e = e' = a \ \wedge \ a \notin X) \\ \vee\ (e = \tau \ \wedge \ e' \in \{\tau, \checkmark\})\end{array}$$

where $(G', n') = \mathtt{FirstEval}(G, n_2, m, \alpha)$

Fig. 7. An instrumented operational semantics to generate CSP tracks

(Synchronized Parallelism 3)
$$\frac{\mathit{RewritingStep}_1 \qquad\qquad \mathit{RewritingStep}_2}{(P_1 \underset{X}{\parallel}_{(\alpha,n_1,n_2)} P_2, G, m, \Delta) \xrightarrow{a} (P'_1 \underset{X}{\parallel}_{(\alpha,n''_1,n''_2)} P'_2, G'', m, \Delta_1 \cup \Delta_2)} \quad a \in X$$

$$\begin{aligned}
&\text{where } G'' = G''_1 \cup G''_2 \cup \{s_1 \overset{a}{\leftrightarrow} s_2 \mid s_1 \in \Delta_1 \wedge s_2 \in \Delta_2\}\\
&\wedge\ \mathit{RewritingStep}_1 = (P_1, G'_1, n'_1, \Delta) \xrightarrow{a} (P'_1, G''_1, n''_1, \Delta_1)\\
&\wedge\ (G'_1, n'_1) \ = \ \texttt{FirstEval}(G, n_1, m, \alpha)\\
&\wedge\ \mathit{RewritingStep}_2 = (P_2, G'_2, n'_2, \Delta) \xrightarrow{a} (P'_2, G''_2, n''_2, \Delta_2)\\
&\wedge\ (G'_2, n'_2) \ = \ \texttt{FirstEval}(G, n_2, m, \alpha)
\end{aligned}$$

(Synchronized Parallelism 4)
$$\frac{}{(\Omega \underset{X}{\parallel}_{(\alpha,n_1,n_2)} \Omega, G, m, \Delta) \xrightarrow{\checkmark} (\Omega, G', r, \emptyset)}$$
$$\text{where } G' = G[\{p \underset{r}{\mapsto} \mid p \underset{q}{\mapsto} \ \in G \text{ where } q \in \{n_1, n_2\}\}]$$

Fig. 7. (*continued)*

in the left and right branches of the choice, and they are initialized to •, a symbol used to express that the branch has not been evaluated yet. Therefore, the first time a branch is evaluated, we generate a new reference for this branch. For this purpose, function `FirstEval` is used:

$$\texttt{FirstEval}(G, n, m, \alpha) \ = \ \begin{cases} (G[m \underset{p}{\mapsto} \alpha], p) & \text{if } n = \bullet \\ (G, n) & \text{otherwise} \end{cases}$$

This function checks whether this is the first time that the branch is evaluated (this only happens when the reference of this branch is empty, i.e., $n = \bullet$). In this case, the choice operator is added to G. For instance, consider the rewriting step (EC4) of Fig. 8. The choice operator in the rewriting step R is labelled with $((\texttt{MAIN}, \Lambda), \bullet, \bullet)$. Therefore, it is evaluated for the first time, and thus, in the left-hand side state of the upper rewriting step, node $5 \underset{6}{\mapsto} (\texttt{MAIN}, 2)$, which refers to the choice operator, is added to G.

(Sequential Composition 1 and 2). Sequential Composition 1 is used to evolve process P until it is finished. P is evolved to P' which is put into the control. When P successfully finishes (it becomes Ω), ✓ happens. Then, Sequential Composition 2 is used and Q is put into the control. The sequential composition operator ; is added to the graph with successor p that is the reference to be used in the first node added in the subderivation associated with Q.

(Synchronized Parallelism 1 and 2). In a synchronized parallel composition, both parallel processes can be intertwiningly executed until a synchronized event is found. Therefore, nodes from both processes can be added interwoven to the graph. Hence, each parallelism operator is labelled with a tuple of the form (α, n_1, n_2) as it happens with external choices.

These rules develop the branches of the parallelism until they are finished or until they must synchronize. In order to introduce the parallelism operator into the graph, function `FirstEval` is used, as it happens in the external choice rules. For instance, consider the rewriting step (Synchronized Parallelism 3) of Fig. 8. The parallelism operator in the rewriting step *State 1* is labelled with $((\texttt{MAIN}, \Lambda), \bullet, \bullet)$. Therefore, it is evaluated for the first time, and thus, in the left-hand side state of the rewriting step L, node $1 \underset{2}{\mapsto} (\texttt{MAIN}, \Lambda)$, which refers to the parallelism operator, is added to G.

(Synchronized Parallelism 3). This rule is used to synchronize the parallel processes. In this case, both branches must perform a rewriting step with the same visible (and synchronized) event. Each branch derivation has a non-empty set of events (Δ_1, Δ_2) to be synchronized (note that this is a set because many parallelisms could be nested). Then, all references in the sets Δ_1 and Δ_2 are mutually linked with synchronization edges. Both sets are joined to form the new set of synchronized events.

(Synchronized Parallelism 4). It is used when none of the parallel processes can proceed because they already successfully finished. In this case, the control becomes Ω indicating the successful termination of the synchronized parallelism. In the new state, the new (fresh) reference is r. This rule also adds to the graph the arcs from all the parents of the last references of each branch (n_1 and n_2) to r. Here, we use the notation $p \underset{r}{\mapsto}$ to add an edge from p to r. Note that the fact of generating the next reference in each rule allows (Synchronized Parallelism 4) to connect the final node of both branches to the next node. This simplifies other rules such as (Sequential Composition) that already has the reference of the node ready.

We illustrate this semantics with a simple example.[3]

Example 6. Consider again the specification in Example 3. Figure 5(a) shows one possible derivation (excluding subderivations) for this example. Note that the underlined part corresponds to the additional rewriting steps performed by the tracking semantics. This derivation corresponds to the execution of the instrumented semantics with the initial state $(\texttt{MAIN}, \emptyset, 0, \emptyset)$ shown in Fig. 8. Here, for clarity, each computation step is labelled with the applied rule; in each state, G denotes the current graph. This computation corresponds to the execution of the right branch of the choice (i.e., $\mathsf{a} \rightarrow \texttt{STOP}$). The final state is $(\bot \underset{\{\mathtt{a}\}}{\|}_{((\texttt{MAIN},\Lambda),9,10)} \bot, G', 1, \emptyset)$. The final track G' computed for this execution is depicted in Fig. 5(b) where we can see that nodes are numbered with the references generated by the instrumented semantics. Note that nodes 9 and 10 were prepared by the semantics (edges to them were produced) but never used because the subcomputations were stopped in STOP. Note also that the track contains all the parts of the specification executed by the semantics. This means that if the left branch of the choice had been developed (i.e., unfolding the call to P, thus using rule (External Choice 3)), this branch would also belong to the track.

[3] We refer the reader to [16] where another example is discussed.

$$\text{(Process Call)}\ \frac{}{(\mathtt{MAIN}, \emptyset, 0, \emptyset) \xrightarrow{\tau} \mathit{State}\ 1}\ \text{where}$$

$$\mathit{State}\ 1 = \ ((\mathsf{a} \to \mathtt{STOP}) \underset{\{\mathtt{a}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),\bullet,\bullet)} (\mathtt{P}\Box(\mathsf{a} \to \mathtt{STOP})), G[0 \underset{1}{\mapsto} (\mathtt{MAIN}, 0)], 1, \emptyset)$$

$$\text{(Synchronized Parallelism 3)}\ \frac{L \quad R}{\mathit{State}\ 1 \xrightarrow{a} \mathit{State}\ 2}\ \text{where}$$

$$L = \text{(Prefixing)}\ \frac{}{(\mathsf{a} \to \mathtt{STOP}, G[1 \underset{2}{\mapsto} (\mathtt{MAIN}, \Lambda)], 2, \emptyset) \xrightarrow{\mathtt{a}} (\mathtt{STOP}, G[2 \underset{3}{\mapsto} (\mathtt{MAIN}, 1.1), 3 \underset{4}{\mapsto} (\mathtt{MAIN}, 1)], 4, \{2\})}$$

$$R = \text{(EC 4)}\ \frac{\text{(Prefixing)}\ \dfrac{}{(\mathsf{a} \to \mathtt{STOP}, G[5 \underset{6}{\mapsto} (\mathtt{MAIN}, 2)], 6, \emptyset) \xrightarrow{\mathtt{a}} (\mathtt{STOP}, G[6 \underset{7}{\mapsto} (\mathtt{MAIN}, 2.2.1), 7 \underset{8}{\mapsto} (\mathtt{MAIN}, 2.2)], 8, \{6\})}}{((\mathtt{P}\Box_{((\mathtt{MAIN},\Lambda),\bullet,\bullet)}(\mathsf{a} \to \mathtt{STOP})), G[1 \underset{5}{\mapsto} (\mathtt{MAIN}, \Lambda)], 5, \emptyset) \xrightarrow{\mathtt{a}} (\mathtt{STOP}, G', 5, \{6\})}$$

$$\text{and } \mathit{State}\ 2 = \ ((\mathtt{STOP} \underset{\{\mathtt{a}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),4,8)} \mathtt{STOP}), G' \cup \{2 \overset{\mathtt{a}}{\leftrightarrow} 6\}, 1, \{2, 6\})$$

$$\text{(Synchronized Parallelism 1)}\ \frac{\text{(STOP)}\ \dfrac{}{(\mathtt{STOP}, G, 4, \{2, 6\}) \xrightarrow{\tau} (\bot, G[4 \underset{9}{\mapsto} (\mathtt{MAIN}, 1.2)], 9, \emptyset)}}{\mathit{State}\ 2 \xrightarrow{\tau} \mathit{State}\ 3}$$

$$\text{where } \mathit{State}\ 3 = (\bot \underset{\{\mathtt{a}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),9,8)} \mathtt{STOP}, G', 1, \emptyset)$$

$$\text{(Synchronized Parallelism 2)}\ \frac{\text{(STOP)}\ \dfrac{}{(\mathtt{STOP}, G, 8, \emptyset) \xrightarrow{\tau} (\bot, G[8 \underset{10}{\mapsto} (\mathtt{MAIN}, 2.2.2)], 10, \emptyset)}}{\mathit{State}\ 3 \xrightarrow{\tau} \mathit{State}\ 4}$$

$$\text{where } \mathit{State}\ 4 = (\bot \underset{\{\mathtt{a}\}}{\parallel}_{((\mathtt{MAIN},\Lambda),9,10)} \bot, G', 1, \emptyset)$$

Fig. 8. An example of computation with the tracking semantics in Fig. 7

5 Correctness

In this section we prove the correctness of the tracking semantics (in Fig. 7) by showing that (i) the computations performed by the tracking semantics are equivalent to the computations performed by the standard semantics; and (ii) the graph produced by the tracking semantics is the track of the derivation. We also prove that the trace of a derivation can be automatically extracted from the track of this derivation.

The first theorem shows that the computations performed with the tracking semantics are all and only the computations performed with the standard semantics. The only difference between them from an operational point of view is that the tracking semantics needs to perform one step when a STOP is evaluated (to add its specification position to the track) and then finishes, while the standard semantics finishes without performing any additional step.

Theorem 1 (Conservativeness). *Let $\mathcal{S}$ be a CSP specification, P a process in $\mathcal{S}$, and $\mathcal{D}$ and $\mathcal{D}'$ the derivations of P performed with the standard semantics of CSP and with the tracking semantics, respectively. Then, the sequence of rules applied in $\mathcal{D}$ and $\mathcal{D}'$ is exactly the same except that $\mathcal{D}'$ performs one rewriting step more than $\mathcal{D}$ for each (sub)computation that finishes with STOP.*

Proof. Firstly, rule (STOP) of the tracking semantics is the only rule that is not present in the standard semantics. When a (STOP) is reached in a derivation, the standard semantics stops the (sub)computation because no rule is applicable. In the tracking semantics, when a STOP is reached in a derivation, the only rule applicable is (STOP) which performs τ and puts $\perp$ in the control:

$$\frac{}{(\mathtt{STOP}_\alpha, G, m, \Delta) \xrightarrow{\tau} (\perp, G[m \underset{n}{\mapsto} \alpha], n, \emptyset)}$$

Then, the (sub)computation is stopped because no rule is applicable for $\perp$. Therefore, when the control in the derivation is STOP, the tracking semantics performs one additional rewriting step with rule (STOP).

The claim follows from the fact that both semantics have exactly the same number of rules except for rule (STOP), and these rules have the same control in all the states of the rules (thus the tracking semantics is a conservative extension of the standard semantics). Therefore, all derivations in both semantics have exactly the same number of steps and they are composed of the same sequences of rewriting steps except for (sub)derivations finishing with STOP that perform one rewriting step more (applying rule (STOP)).

The second theorem states the correctness of the tracking semantics by ensuring that the graph produced is the track of the computation. To prove this theorem, the following lemmas (proven in [16]) are used.

Lemma 1. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each prefixing $(a \rightarrow P)$ in the control of the left state of a rewriting step in $\mathcal{D}$, we have that $\mathcal{P}os(a)$ and $\mathcal{P}os(\rightarrow)$ are nodes of $\mathcal{G}$ and $\mathcal{P}os(\rightarrow)$ is the successor of $\mathcal{P}os(a)$.*

Lemma 2. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each sequential composition $(P;Q)$ in the control of the left state of a rewriting step in $\mathcal{D}$, we have that $last'(P)$ and $\mathcal{P}os(;)$ are nodes of $\mathcal{G}$ and $\mathcal{P}os(;)$ is the successor of all the elements of the set $last'(P)$ whenever P has successfully finished.*

Lemma 3. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a complete derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, for each rewriting step in $\mathcal{D}$ of the form $R_i \overset{\Theta_i}{\rightsquigarrow} R_{i+1}$ we have that:*

1. *E_c contains an edge $\mathcal{P}os(R_i) \mapsto \mathcal{P}os(first(R'))$ where $R' \overset{\Theta'}{\rightsquigarrow} R'' \in \Theta_i$ and $R_i \Rightarrow first(R')$, and*
2. *if $R_i \Rightarrow first(R_{i+1})$ then E_c contains an edge $\mathcal{P}os(R_i) \mapsto \mathcal{P}os(first(R_{i+1}))$.*

Lemma 4. *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, there exists a synchronization edge ($a \leftrightarrow a'$) in $\mathcal{G}$ for each synchronization in $\mathcal{D}$ where a and a' are the nodes of the synchronized events.*

Theorem 2 (Semantics correctness). *Let $\mathcal{S}$ be a CSP specification, $\mathcal{D}$ a derivation of $\mathcal{S}$ performed with the tracking semantics, and $\mathcal{G}$ the graph produced by $\mathcal{D}$. Then, $\mathcal{G}$ is the track associated with $\mathcal{D}$.*

Proof. In order to prove that $\mathcal{G} = (N, E_c, E_s)$ is a track, we need to prove that it satisfies the properties of Definition 5. For each $R \overset{\Theta}{\rightsquigarrow} R' \in \mathcal{D}$ and for all rewriting steps in Θ we have

1. E_c contains a control-flow edge $a \mapsto a'$ iff $a \Rrightarrow a'$ with respect to $\mathcal{D}$. This is ensured by the three clauses of Definition 4:
   - by Lemma 1, if R is a prefixing $(a \rightarrow P)$, then E_c contains an edge $\mathcal{P}os(a) \mapsto \mathcal{P}os(\rightarrow)$;
   - by Lemma 2, if R is a sequential composition $(Q; P)$, then E_c contains an edge $\forall p \in last'(Q), \mathcal{P}os(p) \mapsto \mathcal{P}os(;)$;
   - by Lemma 3, if $R \Rightarrow first(R'')$ where $R'' \overset{\Theta'}{\rightsquigarrow} R''' \in \Theta$, then E_c contains an edge $\mathcal{P}os(R) \mapsto \mathcal{P}os(first(R''))$; and if $R \Rightarrow first(R')$ then E_c contains an edge $\mathcal{P}os(R) \mapsto \mathcal{P}os(first(R'))$; and
2. by Lemma 4, E_s contains a synchronization edge $a \leftrightarrow a'$ for each synchronization occurring in the rewriting step where a and a' are the synchronized events.

Moreover, we know that the only nodes in N are the nodes induced by E_c and E_s because all the nodes inserted in $\mathcal{G}$ are inserted by connecting the new node to the last inserted node (i.e., if the current reference is m and the new fresh reference is n, then the new node is always inserted as $G[m \underset{n}{\mapsto} \alpha]$). Hence, all nodes are related by control or synchronization edges and thus the claim holds.

Our last result states that the trace of a derivation can be extracted from its associated track. To prove it, we define first an order on the event nodes of a track that corresponds to the order in which they were generated by the tracking semantics.

Definition 6. *(Event node order) Given a track $\mathcal{G} = (N, E_c, E_s)$ and nodes $m, n \in N$ such that $l(m), l(n) \in \Sigma$, m is* smaller *than n, represented by $m \ll n$ iff $m' < n'$ where $(m, m'), (n, n') \in E_c$.*

Intuitively, an event node m is smaller than an event node n if and only if the successor of m has a reference smaller than the reference of the successor of n. The following lemma is also necessary to prove that the order in which events occur in a derivation is directly related with the order of Definition 6. In the following we consider an augmented version of derivation $\mathcal{D}$ which includes the event fired by the application of the rule. So, we can represent derivation $\mathcal{D}$ as $P_1 \underset{e_1}{\overset{\Theta_1}{\rightsquigarrow}} \dots \underset{e_j}{\overset{\Theta_j}{\rightsquigarrow}} P_{j+1}$.

Lemma 5. *Given a derivation* $\mathcal{D} = P_1 \underset{e_1}{\overset{\Theta_1}{\rightsquigarrow}} \dots \underset{e_j}{\overset{\Theta_j}{\rightsquigarrow}} P_{j+1}$ *of the tracking semantics, and the track* $\mathcal{G} = (N, E_c, E_s)$ *produced by* $\mathcal{D}$*, then* $\forall e_i \in \Sigma, 1 \le i \le j$,

- $\exists n \in N$ *such that* $l(n) = e_i$*, and*
- $\exists (n, n') \in E_c$ *such that* $n' = n + 1$.

Therefore, Lemma 5 (proven in [16]) ensures that the order of Definition 6 corresponds to the order in which the semantics generates the nodes, because each event is added to the graph together with a new fresh reference for the prefixing operator. Since references are generated incrementally, the occurrence of an event e will generate a reference which is less than the reference generated with a posterior event e'. With this order, we can easily define a transformation to extract a trace from a track based on the following proposition:

Proposition 1. *Given a track* $\mathcal{G} = (N, E_c, E_s)$*, the trace induced by* $\mathcal{G}$ *is the sequence of events* $T = e_1, \dots, e_m$ *that labels the associated sequence of nodes* $T' = n_1, \dots, n_m$ *(i.e.,* $\forall e_i \in T, n_i \in T', 1 \le i \le m, l(n_i) = e_i$ *and* $e_i \in \Sigma$*) where:*

1. $\forall n_i \in T',\ 0 < i < m,\ n_i \ll n_{i+1}$
2. $\forall n \in N$ *such that* $l(n) \in \Sigma$*, if* $(\nexists n' \in N \mid (n, n') \in E_s)$*, then* $n \in T'$
3. $\forall n \in N$ *such that* $l(n) \in \Sigma$*, if* $(\forall n' \in N \mid (n, n') \in E_s \wedge\ n' \ll n)$*, then* $n \in T'$

The proof of this proposition can be found in [16].

Theorem 3 (Track correctness). *Let* $\mathcal{S}$ *be a CSP specification,* $\mathcal{D}$ *a derivation of* $\mathcal{S}$ *produced by the sequence of events (i.e., the trace)* $T = e_1, \dots, e_m$*, and* $\mathcal{G}$ *the track associated with* $\mathcal{D}$*. Then, there exists a function* f *that extracts the trace* T *from the track* $\mathcal{G}$*, i.e.,* $f(\mathcal{G}) = T$.

Proof. Proposition 1 allows to trivially define a function f such that $f(\mathcal{G}) = T$ being $\mathcal{G}$ the track of a derivation $\mathcal{D}$, and being T the trace of the same derivation. For a track $\mathcal{G} = (N, E_c, E_s)$ we have that

$$f((n : ns), E_c, E_s) = \begin{cases} \{f((ns), E_c, E_s)\} & \text{if } (\exists n' \in N | (n, n') \in E_s \wedge n \ll n') \\ (l(n) : f((ns), E_c, E_s)) & \text{otherwise} \end{cases}$$

where list $(n : ns)$ corresponds to the set $\{n \in N \mid l(n) \in \Sigma\}$ ordered with respect to order $\ll$ of Definition 6.

6 Conclusions

This work introduces the first semantics of CSP instrumented for tracking. Therefore, it is an interesting result because it can serve as a reference mark to define and prove properties such as completeness of static analyses which are based on tracks [13,14,15]. The execution of the tracking semantics produces a graph as a side effect which is the track of the computation. This track is produced step by step by the semantics, and thus, it can be also used to produce a track of an infinite computation until it is stopped. The generated track can be useful not only for tracking computations but for debugging and program comprehension. This is due to the fact that our generated track also includes the specification positions associated with the expressions appearing in the track. Therefore, tracks could be used to analyse what parts of the program are executed (and in what order) in a particular computation. Also, this information allows a track viewer tool to highlight the parts of the code that are executed in each step. Notable analyses that use tracks are [3,4,5,1,13,14,15]. The introduction of this semantics allows us to adapt these analyses to CSP. On the practical side, we have implemented a tool called *SOC* [13] which is able to automatically generate tracks of a CSP specification. These tracks are later used for debugging. *SOC* has been integrated into the most extended CSP animator and model-checker ProB [11,12], that shows the maturity and usefulness of this tool and of tracks. The implementation, source code and several examples are publicly available at: `http://users.dsic.upv.es/~jsilva/soc/`

Acknowledgements

We want to thank the anonymous referees for many valuable comments and useful suggestions.

References

1. Brassel, B., Hanus, M., Huch, F., Vidal, G.: A Semantics for Tracing Declarative Multi-paradigm Programs. In: Moggi, E., Warren, D.S. (eds.) 6th ACM SIGPLAN Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'04), pp. 179–190. ACM, New York, NY, USA (2004)
2. Brückner, I., Wehrheim, H.: Slicing an Integrated Formal Method for Verification. In: Lau, K.K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 360–374. Springer, Heidelberg (2005)
3. Chitil, O.: A Semantics for Tracing. In: Arts, T., Mohnen, M. (eds.) 13th Int'l Workshop on Implementation of Functional Languages (IFL'01), pp. 249–254. Ericsson CSL (2001)
4. Chitil, O., Runciman, C., Wallace, M.: Transforming Haskell for Tracing. In: Peña, R., Arts, T. (eds.) IFL 2002, Revised Selected Papers. LNCS, vol. 2670, pp. 165–181. Springer, Heidelberg (2003)
5. Chitil, O., Lou, Y.: Structure and Properties of Traces for Functional Programs. Electronic Notes in Theoretical Computer Science (ENTCS). 176(1), 39–63 (2007)

6. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The Program Dependence Graph and its Use in Optimization. ACM Transactions on Programming Languages and Systems. 9(3), 319–349 (1987)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River, NJ, USA (1985)
8. Kavi, K.M., Sheldon, F.T., Shirazi, B., Hurson, A.R.: Reliability analysis of CSP specifications using Petri nets and Markov processes. In: 28th Annual Hawaii Int'l Conf. on System Sciences (HICSS'95), vol. 2 (Software Technology), pp. 516–524. IEEE Computer Society, Washington, DC, USA (1995)
9. Krinke, J.: Context-Sensitive Slicing of Concurrent Programs. ACM SIGSOFT Software Engineering Notes. 28(5) (2003)
10. Ladkin, P., Simons, B.: Static Deadlock Analysis for CSP-Type Communications. Responsive Computer Systems (Chapter 5), Kluwer Academic Publishers (1995)
11. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Journal of Software Tools for Technology Transfer. 10(2), 185–203 (2008)
12. Leuschel, M., Fontaine, M.: Probing the depths of CSP-M: A new FDR-compliant validation tool. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 278–297. Springer, Heildeberg (2008)
13. Leuschel, M., Llorens, M., Oliver, J., Silva, J., Tamarit, S.: SOC: a Slicer for CSP Specifications. In: Puebla, G., Vidal, G. (eds.) 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'09), pp. 165–168. ACM, New York, NY, USA (2009)
14. Leuschel, M., Llorens, M., Oliver, J., Silva, J., Tamarit, S.: The MEB and CEB Static Analysis for CSP Specifications. In: Hanus, M. (ed.) LOPSTR 2008, Revised Selected Papers. LNCS, vol. 5438, pp. 103–118. Springer, Heildeberg (2009)
15. Llorens, M., Oliver, J., Silva, J., Tamarit, S.: An Algorithm to Generate the Context-sensitive Synchronized Control Flow Graph. In: 25th ACM Symposium on Applied Computing (SAC 2010), vol. 3, pp. 2144–2148. ACM, New York (2010)
16. Llorens, M., Oliver, J., Silva, J., Tamarit, S.: A Tracking Semantics for CSP (Extended Version). Technical report, DSIC-II/03/10, Universidad Politécnica de Valencia (March 2010)
17. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall, Upper Saddle River, NJ, USA (2005)
18. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages. 3, 121–189 (1995)
19. Weiser, M.D.: Program Slicing. IEEE Transactions on Software Engineering. 10(4), 352–357 (1984)

Matrices as Arrows!
A Biproduct Approach to Typed Linear Algebra

Hugo Daniel Macedo and José Nuno Oliveira

Minho University, Portugal
{hmacedo,jno}@di.uminho.pt

Abstract. Motivated by the need to formalize generation of fast running code for linear algebra applications, we show how an index-free, calculational approach to matrix algebra can be developed by regarding matrices as morphisms of a category with biproducts. This shifts the traditional view of matrices as indexed structures to a type-level perspective analogous to that of the pointfree algebra of programming. The derivation of fusion, cancellation and abide laws from the biproduct equations makes it easy to calculate algorithms implementing matrix multiplication, the kernel operation of matrix algebra, ranging from its divide-and-conquer version to the conventional, iterative one.

From errant attempts to learn how particular products and coproducts emerge from biproducts, we not only rediscovered block-wise matrix combinators but also found a way of addressing other operations calculationally such as e.g. Gaussian elimination. A strategy for addressing vectorization along the same lines is also given.

1 Introduction

Automatic generation of fast running code for linear algebra applications calls for matrix multiplication as kernel operator, whereby matrices are viewed and transformed in an index-free way [1]. Interestingly, the successful language SPL [2] used in generating automatic parallel code has been created envisaging the same principles as advocated by the purist computer scientist: index-free abstraction and composition (multiplication) as a kernel way of connecting objects of interest (matrices, programs, etc).

There are several domain specific languages (DSLs) bearing such purpose in mind [2,3,1]. However, they arise as programming dialects with poor type checking. This may lead to programming errors, hindering effective use of such languages and calling for a "type structure" in linear algebra systems similar to that underlying modern functional programming languages such as Haskell, for instance [4].

It so happens that, in the same way function composition is the kernel operation of functional programming, leading to the *algebra of programming* [5], so does matrix multiplication once matrices are viewed and transformed in an index-free way. So, rather than interpreting the product AB of matrices A and B as an algorithm for computing a new matrix C out of A and B, and trying to build and explain matrix algebra systems out of such an algorithm, one wishes to abstract from *how* the operation is carried out. Instead, the emphasis is put on its type structure, regarded as the pipeline $A \cdot B$ (to be read as "A after B"), as if A and B were functions

$$C = A \cdot B \tag{1}$$

or binary relations — the actual building block of the algebra of programming [5]. In this discipline, relations are viewed as (typed) composable arrows (morphisms) which can be combined in a number of ways, namely by joining or intersecting relations of the same type, reversing them (thus swapping their source and target types), and so on.

If relations, which are Boolean matrices, can be regarded as morphisms of a suitable mathematical framework, why not regard arbitrary matrices in the same way? This matches with the categorical characterization of matrices, which can be traced back to Mac Lane [6], whereby matrices are regarded as arrows in a category whose objects are natural numbers (matrix dimensions):

$$A = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}_{m \times n} \qquad\qquad m \xleftarrow{A} n \tag{2}$$

Such a category $\mathbf{Mat}_K$ of matrices over a field K merges categorical products and coproducts into a single construction termed *biproduct* [6]. Careful analysis of the biproduct axioms as a system of equations provides one with a rich *palette* of constructs for building matrices from smaller ones. In [7], for instance, we outlined an approach to matrix blocked operation stemming from one particular solution to such equations, which in fact offers explicit operators for building block-wise matrices (row and column-wise) as defined by [8].

In the current paper we elaborate on [7] and show in detail how block-driven divide-and-conquer algorithms for linear algebra arise from biproduct laws emerging from the underlying categorial basis. In summary, this paper gives the details of a constructive approach to matrix algebra operations leading to elegant, index-free proofs of the corresponding algorithms. As happens with state-of-the-art algebra of programming, the whole framework is fully typed, enabling parametric type checking of matrix combinators.

2 The Category of Matrices $\mathbf{Mat}_K$

Matrices are mathematical objects that can be traced back to ancient times, documented as early as 200 BC [9]. The word "matrix" was introduced in the western culture much later, in the 1840's, by the mathematician James Sylvester (1814-1897) when both matrix theory and linear algebra emerged.

The traditional way of viewing matrices as rectangular tables (2) of elements or entries (the "container view") which in turn are other mathematical objects such as e.g. complex numbers (in general: inhabitants of the field K which underlies $\mathbf{Mat}_K$), encompasses as special cases one column and one line matrices, referred to as column (resp. row) *vectors*, that is, matrices of shapes

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_m \end{bmatrix} \qquad \text{and} \qquad w = \begin{bmatrix} w_1 \dots w_n \end{bmatrix}$$

What is a matrix? The standard answer to this question is to regard matrix A (2) as a computation unit, or transformation, which commits itself to producing a (column) vector of size m provided it is supplied with a (column) vector of size n. How is such output produced? Let us abstract from this at this stage and look at diagram

$$\begin{tikzcd} m & n \arrow[l, "A"'] & 1 \arrow[l, "v"'] \arrow[ll, bend left, squiggly, "w"] \end{tikzcd}$$

arising from depicting the situation in arrow notation. This suggests a pictorial representation of the product of matrix $A_{m\times n}$ and matrix $B_{n\times q}$, yielding a new matrix $C = (AB)_{m\times q}$ with dimensions $m \times q$, as follows,

$$\begin{tikzcd} m & n \arrow[l, "A"'] & q \arrow[l, "B"'] \arrow[ll, bend left, squiggly, "C=A\cdot B"] \end{tikzcd} \tag{3}$$

which automatically "type-checks" the construction: the "target" of $n \xleftarrow{B} q$ simply matches the "source" of $m \xleftarrow{A} n$ yielding a matrix whose type $m \longleftarrow q$ is the composition of the given types.

Having defined matrices as composable arrows in a category, we need to define its identities [6]: for every object n, there must be an arrow of type $n \longleftarrow n$ which is the unit of composition. This is nothing but the identity matrix of size n, which we will denote by $n \xleftarrow{id_n} n$. For every matrix $m \xleftarrow{A} n$, one has

$$id_m \cdot A \;=\; A \;=\; A \cdot id_n \qquad \begin{tikzcd} n \arrow[d, "A"'] & n \arrow[l, "id_n"'] \arrow[d, "A"] \arrow[dl, "A"] \\ m & m \arrow[l, "id_m"] \end{tikzcd} \tag{4}$$

(Subscripts m and n can be omitted wherever the underlying diagrams are assumed.)

Transposed matrices. One of the kernel operations of linear algebra is *transposition*, whereby a given matrix changes shape by turning its rows into columns and vice-versa. Type-wise, this means converting an arrow $n \xleftarrow{A} m$ into an arrow $m \xleftarrow{A^T} n$, that is, source and target types (dimensions) switch over. By analogy with relation algebra, where a similar operation is termed *converse* and denoted A°, we will use this notation instead of A^T (which misleadingly suggests a kind of exponential) and will say "A converse" wherever reading A°. Index-wise, we have, for A as in (2):

$$A^\circ = \begin{bmatrix} a_{11} & \dots & a_{m1} \\ \vdots & \ddots & \vdots \\ a_{1n} & \dots & a_{mn} \end{bmatrix} \qquad n \xleftarrow{A^\circ} m$$

Instead of telling how transposition is carried out index-wise, again we prefer to stress on (index-free) properties of this operation such as, among others, idempotence and contravariance:

$$(A^\circ)^\circ = A \tag{5}$$

$$(A \cdot B)^\circ = B^\circ \cdot A^\circ \tag{6}$$

Bilinearity. Given two matrices of the same type $m \xleftarrow{M,N} n$ (i.e., in the same homset of $\mathbf{Mat}_K$) it makes sense to add them up index-wise, leading to matrix $M + N$ where symbol + promotes the underlying element-level additive operator to matrix-level. In fact, matrices form an *Abelian category*: each homset in the category forms an additive Abelian (ie. commutative) group with respect to which composition is bilinear:

$$M \cdot (N + L) = M \cdot N + M \cdot L \tag{7}$$

$$(N + L) \cdot K = N \cdot K + L \cdot K \tag{8}$$

Polynomial expressions (such as in the properties above) denoting matrices built up in an index-free way from addition and composition play a major role in matrix algebra. This can be appreciated in the explanation of the very important concept of a *biproduct* which follows.

Biproducts. In an Abelian category, a *biproduct* diagram for the objects m, n is a diagram of shape

$$\begin{tikzcd} m \arrow[r, shift right, "i_1"'] & r \arrow[l, shift right, "\pi_1"'] \arrow[r, shift left, "\pi_2"] & n \arrow[l, shift left, "i_2"] \end{tikzcd}$$

whose arrows π_1, π_2, i_1, i_2 satisfy the identities which follow:

$$\pi_1 \cdot i_1 = id_m \tag{9}$$

$$\pi_2 \cdot i_2 = id_n \tag{10}$$

$$i_1 \cdot \pi_1 + i_2 \cdot \pi_2 = id_r \tag{11}$$

Morphisms π_i and i_i are termed *projections* and *injections*, respectively. From the underlying arithmetics one easily derives the following orthogonality properties (see e.g. [6]):

$$\pi_1 \cdot i_2 = 0 \qquad , \qquad \pi_2 \cdot i_1 = 0 \tag{12}$$

One wonders: how do biproducts relate to products and co-products in the category? The answer in Mac Lane's [6] words is as follows:

Theorem 2: Two objects a and b in Abelian category A have a product in A iff they have a biproduct in A. Specifically, given a biproduct diagram, the object r with the projections π_1 and π_2 is a product of m and n, while, dually, r with i_1 and i_2 is a coproduct. In particular, two objects m and n have a product in A if and only if they have a coproduct in A.

The diagram and definitions below depict how products and coproducts arise from biproducts (the product diagram is in the lower half; the upper half is the coproduct one):

$$\begin{tikzcd}
 & m & \\
n \arrow[ur, "R"] \arrow[r, shift right, "i_1"'] & n+p \arrow[u, "{\left[\,R|S\,\right]}"] \arrow[l, shift right, "\pi_1"'] \arrow[r, shift left, "\pi_2"] & p \arrow[ul, "S"'] \arrow[l, shift left, "i_2"] \\
 & t \arrow[ul, "U"] \arrow[ur, "V"'] \arrow[u, "{\left[\frac{U}{V}\right]}"] &
\end{tikzcd}$$

$$\left[\,R|S\,\right] = R \cdot \pi_1 + S \cdot \pi_2 \tag{13}$$

$$\left[\frac{U}{V}\right] = i_1 \cdot U + i_2 \cdot V \tag{14}$$

By analogy with the algebra of programming [5], expressions $\left[\,R|S\,\right]$ and $\left[\frac{U}{V}\right]$ will be read "R junc S" and "U split V", respectively. What is the intuition behind these combinators, which come out of the blue in texts such as e.g. [8]? Expressed in terms of definitions (13) and (14), axiom (11) rewrites to both

$$\left[\,i_1|i_2\,\right] = id \tag{15}$$

$$\left[\frac{\pi_1}{\pi_2}\right] = id \tag{16}$$

somehow suggesting that the two injections and the two projections "decompose" the identity matrix. On the other hand, each of (15,16) has the shape of a *reflection* corollary [5] of some universal property. Below we derive such a property for $\left[\,R|S\,\right]$,

$$X = \left[\,R|S\,\right] \Leftrightarrow \begin{cases} X \cdot i_1 = R \\ X \cdot i_2 = S \end{cases} \tag{17}$$

from the underlying biproduct equations, by circular implication:

$$X = \left[\,R|S\,\right]$$

$$\Leftrightarrow \qquad \{ \text{ identity (4) ; (13) } \}$$

$$
\begin{array}{ll}
& X \cdot id = R \cdot \pi_1 + S \cdot \pi_2 \\
\Leftrightarrow & \qquad \{ \text{ (11) } \} \\
& X \cdot (i_1 \cdot \pi_1 + i_2 \cdot \pi_2) = R \cdot \pi_1 + S \cdot \pi_2 \\
\Leftrightarrow & \qquad \{ \text{ bilinearity (7) } \} \\
& X \cdot i_1 \cdot \pi_1 + X \cdot i_2 \cdot \pi_2 = R \cdot \pi_1 + S \cdot \pi_2 \\
\Rightarrow & \qquad \{ \text{ Leibniz (twice) } \} \\
& \begin{cases} (X \cdot i_1 \cdot \pi_1 + X \cdot i_2 \cdot \pi_2) \cdot i_1 = (R \cdot \pi_1 + S \cdot \pi_2) \cdot i_1 \\ (X \cdot i_1 \cdot \pi_1 + X \cdot i_2 \cdot \pi_2) \cdot i_2 = (R \cdot \pi_1 + S \cdot \pi_2) \cdot i_2 \end{cases} \\
\Leftrightarrow & \qquad \{ \text{ bilinearity (8) ; biproduct (9,10) ; orthogonality (12) } \} \\
& \begin{cases} X \cdot i_1 + X \cdot i_2 \cdot 0 = R + S \cdot 0 \\ X \cdot i_1 \cdot 0 + X \cdot i_2 = R \cdot 0 + S \end{cases} \\
\Leftrightarrow & \qquad \{ \text{ trivia } \} \\
& \begin{cases} X \cdot i_1 = R \\ X \cdot i_2 = S \end{cases} \\
\Rightarrow & \qquad \{ \text{ Leibniz (twice) } \} \\
& \begin{cases} X \cdot i_1 \cdot \pi_1 = R \cdot \pi_1 \\ X \cdot i_2 \cdot \pi_2 = S \cdot \pi_2 \end{cases} \\
\Rightarrow & \qquad \{ \text{ Leibniz } \} \\
& X \cdot i_1 \cdot \pi_1 + X \cdot i_2 \cdot \pi_2 = R \cdot \pi_1 + S \cdot \pi_2 \\
\Leftrightarrow & \qquad \{ \text{ as shown above } \} \\
& X = \left[R | S \right]
\end{array}
$$

The derivation of the universal property of $\left[\frac{U}{V}\right]$,

$$X = \left[\frac{U}{V}\right] \Leftrightarrow \begin{cases} \pi_1 \cdot X = U \\ \pi_2 \cdot X = V \end{cases} \qquad (18)$$

is (dually) analogous.

Parallel with relation algebra. Similar to matrix algebra, relation algebra [10,5] can also be explained in terms of biproducts once morphism addition (11) is interpreted as relational union, object union is disjoint union, i_1 and i_2 as the corresponding injections and π_1,π_2 their converses, respectively. Relational product should not, however, be confused with the *fork* construct [11] in fork (relation) algebra, which involves pairing. (For this to become a product one has to restrict to functions.)

In the next section we show that the converse relationship (duality) between projections and injections is not a privilege of relation algebra: the most intuitive biproduct solution in the category of matrices also offers such a duality.

3 Chasing Biproducts

Let us now address the intuition behind products and coproducts of matrices. This has mainly to do with the interpretation of projections π_1, π_2 and injections i_1,i_2 arising as solutions of biproduct equations (9,10,11). Concerning this, Mac Lane [6] laconically writes:

"In other words, the [biproduct] equations contain the familiar calculus of matrices."

In what way? The answer to this question proved more interesting than it seems at first, because of the multiple solutions arising from a non-linear system of three equations (9,10,11) with four variables. In trying to exploit this freedom we became aware that each solution offers a particular way of putting matrices together via the corresponding "junc" and "split" combinators.

Our inspection of solutions started by reducing the "size" of the objects involved and experimenting with the smaller biproduct depicted below:

$$\begin{tikzcd} 1 \arrow[r, shift right, "i_1"'] & 1+1 \arrow[l, shift right, "\pi_1"'] \arrow[r, shift left, "\pi_2"] & 1 \arrow[l, shift left, "i_2"] \end{tikzcd}$$

The "puzzle" in this case is more manageable,

$$\begin{cases} \pi_1 \cdot i_1 & = [1] \\ \pi_2 \cdot i_2 & = [1] \\ i_1 \cdot \pi_1 + i_2 \cdot \pi_2 & = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{cases}$$

yet the set of solutions is not small. We used the Mathematica software [12] to solve this system by inputting the projections and injections as suitably typed matrices leading to a larger, non-linear system:

$$\begin{cases} \begin{bmatrix} \pi_{11} & \pi_{12} \end{bmatrix} \cdot \begin{bmatrix} i_{11} \\ i_{12} \end{bmatrix} & = [1] \\ \begin{bmatrix} \pi_{21} & \pi_{22} \end{bmatrix} \cdot \begin{bmatrix} i_{21} \\ i_{22} \end{bmatrix} & = [1] \\ \begin{bmatrix} i_{11} \\ i_{12} \end{bmatrix} \cdot \begin{bmatrix} \pi_{11} & \pi_{12} \end{bmatrix} + \begin{bmatrix} i_{21} \\ i_{22} \end{bmatrix} \cdot \begin{bmatrix} \pi_{21} & \pi_{22} \end{bmatrix} & = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{cases}$$

This was solved using the standard Solve command obtaining the output presented in Figure 1, which offers several solutions. Among these we first picked the one which purports the most intuitive reading of the *junc* and *split* combinators

sol = Solve[{pi1.i1 == I1, pi2.i2 == I1, i1.pi1 + i2.pi2 == I2}]
Solve::svars : Equations may not give solutions for all "solve" variables. ⟩⟩

$$\left\{\left\{i_{21} \to \tfrac{1}{\pi_{21}}, i_{22} \to -\tfrac{\pi_{11} i_{12}}{\pi_{21}}, \pi_{12} \to \tfrac{1}{i_{12}}, i_{11} \to 0, \pi_{22} \to 0\right\},\right.$$
$$\left.\left\{i_{21} \to -\tfrac{\pi_{12} i_{11}}{\pi_{22}}, i_{22} \to \tfrac{\pi_{22}+\pi_{12}\pi_{21} i_{11}}{(\pi_{22})^2}, \pi_{11} \to \tfrac{\pi_{22}+\pi_{12}\pi_{21} i_{11}}{\pi_{22} i_{11}}, i_{12} \to -\tfrac{\pi_{21} i_{11}}{\pi_{22}}\right\}\right\}$$

Fig. 1. Fragment of Mathematica script

— that of simply gluing matrices vertically and horizontally (respectively) with no further computation of matrix entries:

$$\pi_1 = \begin{bmatrix} 1 & 0 \end{bmatrix} \qquad \pi_2 = \begin{bmatrix} 0 & 1 \end{bmatrix}$$
$$i_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad i_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Interpreted in this way, $\left[\frac{R}{S}\right]$ (14) and $\left[\, R|S \,\right]$ (13) are the block gluing matrix operators which we can find in [8]. Our choice of notation — R above S in the case of (14) and R besides S in the case of (13) reflects this semantics.

The obvious generalization of this solution to higher dimensions of the problem leads to the following matrices with identities of size m and n in the appropriate place, so as to properly typecheck:

$$\pi_1 = \; m \xleftarrow{\left[\, id_m \mid 0 \,\right]} m+n \quad , \quad \pi_2 = \; n \xleftarrow{\left[\, 0 \mid id_n \,\right]} m+n$$
$$i_1 = \; m+n \xleftarrow{\left[\frac{id_m}{0}\right]} m \quad , \quad i_2 = \; m+n \xleftarrow{\left[\frac{0}{id_m}\right]} n$$

By inspection, one immediately infers the same duality found in relation algebra,

$$\pi_1^\circ = i_1 \; , \; \pi_2^\circ = i_2 \tag{19}$$

whereby *junc* (13) and *split* (14) become self dual:

$$\begin{array}{cl} & \left[\, R|S \,\right]^\circ \\ = & \quad \{ \;\; (13) \; ; \; (6) \;\; \} \\ & \pi_1^\circ \cdot R^\circ + \pi_2^\circ \cdot S^\circ \\ = & \quad \{ \;\; (19) \; ; \; (14) \;\; \} \\ & \left[\frac{R^\circ}{S^\circ}\right] \end{array}$$

This particular solution to the biproduct equations captures what in the literature is meant by *blocked* matrix algebra, a generalization of the standard element-wise operations to sub-matrices, or blocks, leading to *divide-and-conquer* versions of the corresponding algorithms. The next section shows the exercise of deriving such laws, thanks to the algebra which emerges from the universal properties of the block-gluing matrix combinators *junc* (17) and *split* (18). We combine the standard terminology with that borrowed from the algebra of programming [5] to stress the synergy between blocked matrix algebra and relational algebra.

4 Blocked Linear Algebra — Calculationally!

Further to reflection laws (15,16), the derivation of the following equalities from universal properties (17,18) is a standard exercise in (high) school algebra, where capital letters A, B, etc. denote suitably typed matrices (the types, ie. dimensions, involved in each equality can be inferred by drawing the corresponding diagram):

– Two "fusion"-laws:

$$C \cdot \left[\, A \middle| B \,\right] = \left[\, C \cdot A \middle| C \cdot B \,\right] \tag{20}$$

$$\left[\frac{A}{B}\right] \cdot C = \left[\frac{A \cdot C}{B \cdot C}\right] \tag{21}$$

– Four "cancellation"-laws:

$$\left[\, A \middle| B \,\right] \cdot i_1 = A \;,\; \left[\, A \middle| B \,\right] \cdot i_2 = B \tag{22}$$

$$\pi_1 \cdot \left[\frac{A}{B}\right] = A \;,\; \pi_2 \cdot \left[\frac{A}{B}\right] = B \tag{23}$$

– Three "abide"-laws [1]: the *junc/split* exchange law

$$\left[\frac{\left[\, A \middle| B \,\right]}{\left[\, C \middle| D \,\right]}\right] = \left[\left[\frac{A}{C}\right] \middle| \left[\frac{B}{D}\right]\right] = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \tag{24}$$

which tells the equivalence between row-major and column-major construction of matrices (thus the four entry *block* notation on the right), and two *blocked addition* laws:

$$\left[\, A \middle| B \,\right] + \left[\, C \middle| D \,\right] = \left[\, A + C \middle| B + D \,\right] \tag{25}$$

$$\left[\frac{A}{B}\right] + \left[\frac{C}{D}\right] = \left[\frac{A + C}{B + D}\right] \tag{26}$$

[1] Neologism "abide" (= "above and beside") was introduced by Richard Bird [13] as a generic name for algebraic laws in which two binary operators written in infix form change place between "above" and "beside", e.g.

$$\frac{a}{b} \times \frac{c}{d} = \frac{a \times c}{b \times d}$$

The laws above are more than enough for us to derive standard linear algebra rules and algorithms in a calculational way. As an example of their application we provide a simple proof of the rule which underlies *divide-and-conquer* matrix multiplication:

$$\left[\, R \middle| S \,\right] \cdot \left[\frac{U}{V}\right] = R \cdot U + S \cdot V \tag{27}$$

We calculate:

$$\begin{array}{ll}
& \left[\, R \middle| S \,\right] \cdot \left[\frac{U}{V}\right] \\
= & \qquad \{ \ (14) \ \} \\
& \left[\, R \middle| S \,\right] \cdot (i_1 \cdot U + i_2 \cdot V) \\
= & \qquad \{ \ \text{bilinearity (7)} \ \} \\
& \left[\, R \middle| S \,\right] \cdot i_1 \cdot U + \left[\, R \middle| S \,\right] \cdot i_2 \cdot V \\
= & \qquad \{ \ \text{+-cancellation (22)} \ \} \\
& R \cdot U + S \cdot V
\end{array}$$

As another example, let us show how standard block-wise matrix-matrix multiplication (MMM),

$$\left[\begin{array}{c|c} R & S \\ \hline U & V \end{array}\right] \cdot \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] = \left[\begin{array}{c|c} RA + SC & RB + SD \\ \hline UA + VC & UB + VD \end{array}\right] \tag{28}$$

relies on *divide-and-conquer* (27):

$$\begin{array}{ll}
& \left[\left[\frac{R}{U}\right] \middle| \left[\frac{S}{V}\right] \right] \cdot \left[\left[\frac{A}{C}\right] \middle| \left[\frac{B}{D}\right] \right] \\
= & \qquad \{ \ \textit{junc}\text{-fusion (20)} \ \} \\
& \left[\left[\left[\frac{R}{U}\right] \middle| \left[\frac{S}{V}\right] \right] \cdot \left[\frac{A}{C}\right] \middle| \left[\left[\frac{R}{U}\right] \middle| \left[\frac{S}{V}\right] \right] \cdot \left[\frac{B}{D}\right] \right] \\
= & \qquad \{ \ \text{divide and conquer (27) twice} \ \} \\
& \left[\left[\frac{R}{U}\right] \cdot A + \left[\frac{S}{V}\right] \cdot C \ \middle| \ \left[\frac{R}{U}\right] \cdot B + \left[\frac{S}{V}\right] \cdot D \right] \\
= & \qquad \{ \ \text{split-fusion (20) four times} \ \} \\
& \left[\left[\frac{R \cdot A}{U \cdot A}\right] + \left[\frac{S \cdot C}{V \cdot C}\right] \ \middle| \ \left[\frac{R \cdot B}{U \cdot B}\right] + \left[\frac{S \cdot D}{V \cdot D}\right] \right]
\end{array}$$

$$= \quad \{ \text{ blocked addition (26) twice } \}$$

$$\left[\left[\frac{R \cdot A + S \cdot C}{U \cdot A + V \cdot C} \right] \middle| \left[\frac{R \cdot B + S \cdot D}{U \cdot B + V \cdot D} \right] \right]$$

$$= \quad \{ \text{ the same in block notation (24) } \}$$

$$\left[\begin{array}{c|c} RA + SC & RB + SD \\ \hline UA + VC & UB + VD \end{array} \right]$$

5 Calculating Triple Nested Loops

By putting together the universal factorization of matrices in terms of the *junc* and *split* combinators, one easily infers yet another such property handling four blocks at a time:

$$X = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right] \Leftrightarrow \begin{cases} \pi_1 \cdot X \cdot i_1 = A_{11} \\ \pi_1 \cdot X \cdot i_2 = A_{12} \\ \pi_2 \cdot X \cdot i_1 = A_{21} \\ \pi_2 \cdot X \cdot i_2 = A_{22} \end{cases}$$

Alternatively, one may generalize (13,14) to blocked notation

$$\left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right] = i_1 \cdot A_{11} \cdot \pi_1 + i_1 \cdot A_{12} \cdot \pi_1 + i_1 \cdot A_{21} \cdot \pi_1 + i_2 \cdot A_{22} \cdot \pi_2$$

which rewrites to

$$\left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right] = \begin{bmatrix} A_{11} & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & A_{12} \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ A_{21} & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & A_{22} \end{bmatrix}$$

once injections and projections are replaced by the biproduct solution found in Section 3.

Iterated Biproducts. It should be noted that biproducts generalize to finitely many arguments, leading to an n-ary generalization of the (binary) *junc* / *split* combinators. The following notation is adopted in generalizing (13,14):

$$A = \left[A_1 \middle| \ldots \middle| A_p \right] = \bigoplus_{1 \leq j \leq p} A \cdot i_j = \sum_{j=1}^{p} A \cdot i_j \cdot \pi_j \tag{29}$$

$$A = \left[\begin{array}{c} A_1 \\ \hline \vdots \\ \hline A_m \end{array} \right] = \bigominus_{1 \leq j \leq m} \pi_j \cdot A = \sum_{j=1}^{m} i_j \cdot \pi_j \cdot A \tag{30}$$

Note that all laws given so far generalize accordingly to n-ary *splits* and *juncs*. In particular, we have the following universal properties:

$$X = \bigoplus_{1 \leq j \leq p} A_j \Leftrightarrow \bigwedge_{1 \leq j \leq p} X \cdot i_j = A_j \tag{31}$$

$$X = \bigominus_{1 \leq j \leq m} A_j \Leftrightarrow \bigwedge_{1 \leq j \leq m} \pi_j \cdot X = A_j \tag{32}$$

Further note that m, p can be chosen as large as possible, the limit taking place when blocks A_i become atomic. In this limit situation, a given matrix $m \xleftarrow{A} n$ is defined in terms of its elements A_{jk} as:

$$A = \left[\begin{array}{c|c|c} a_{11} & \ldots & a_{1n} \\ \hline \vdots & \ddots & \vdots \\ \hline a_{m1} & \ldots & a_{mn} \end{array}\right] = \sum_{\substack{1\leq j\leq m \\ 1\leq k\leq n}} i_j \cdot \pi_j \cdot A \cdot i_k \cdot \pi_k = \bigoplus_{\substack{1\leq j\leq m \\ 1\leq k\leq n}} \pi_j \cdot A \cdot i_k \tag{33}$$

where $\bigoplus_{\substack{1\leq j\leq m \\ 1\leq k\leq n}}$ abbreviates $\ominus_{1\leq j\leq m} \oplus_{1\leq k\leq n}$ — equivalent to $\oplus_{1\leq k\leq n} \ominus_{1\leq j\leq m}$ by the generalized exchange law (24).

Our final calculation shows how iterated biproducts "explain" the traditional for-loop implementation of MMM. Interestingly enough, such iterative implementation is shown to stem from generalized divide-and-conquer (27):

$$\begin{aligned}
& C = A \cdot B \\
= & \quad \{ \ (33), (29) \text{ and } (30) \ \} \\
& (\bigominus_{1\leq j\leq m} \pi_j \cdot A) \cdot (\bigoplus_{1\leq k\leq n} B \cdot i_k) \\
= & \quad \{ \ \text{generalized split-fusion } (21) \ \} \\
& \bigominus_{1\leq j\leq m} (\pi_j \cdot A \cdot (\bigoplus_{1\leq k\leq n} B \cdot i_k)) \\
= & \quad \{ \ \text{generalized either-fusion } (20) \ \} \\
& \bigominus_{1\leq j\leq m} (\bigoplus_{1\leq k\leq n} \pi_j \cdot A \cdot B \cdot i_k) \\
= & \quad \{ \ (29), (30) \text{ and generalized } (21) \text{ and } (20) \ \} \\
& \bigominus_{1\leq j\leq m} (\bigoplus_{1\leq k\leq n} ((\bigoplus_{1\leq l\leq p} \pi_j \cdot A \cdot i_l) \cdot (\bigominus_{1\leq l\leq p} \pi_l \cdot B \cdot i_k))) \\
= & \quad \{ \ \text{generalized divide-and-conquer } (27) \ \} \\
& \bigominus_{1\leq j\leq m} (\bigoplus_{1\leq k\leq n} (\sum_{1\leq l\leq p} \pi_j \cdot A \cdot i_l \times \pi_l \cdot B \cdot i_k))
\end{aligned}$$

As we can see in the derivation path, the choices for the representation of A and B impact on the derivation of the intended algorithm. Different choices will alter the order of the triple loop obtained. Proceeding to the loop inference will involve the expansion of C and the normalization of the formula into sum-wise notation:

$$\begin{aligned}
& \bigoplus_{\substack{1\leq k\leq m \\ 1\leq j\leq n}} \pi_j \cdot C \cdot i_k = \bigominus_{1\leq j\leq m} (\bigoplus_{1\leq k\leq n} (\sum_{1\leq l\leq p} \pi_j \cdot A \cdot i_l \times \pi_l \cdot B \cdot i_k)) \\
\Leftrightarrow & \quad \{ \ (33), (29) \text{ and } (30) \ \} \\
& \bigominus_{1\leq j\leq m} (\bigoplus_{1\leq k\leq n} \pi_j \cdot C \cdot i_k) = \bigominus_{1\leq j\leq m} (\bigoplus_{1\leq k\leq n} (\sum_{1\leq l\leq p} \pi_j \cdot A \cdot i_l \times \pi_l \cdot B \cdot i_k))
\end{aligned}$$

At this point we rely on the universality of the *junc* and *split* constructs (31,32) to obtain from above the post-condition of the algorithm:

$$\bigwedge_{1\leq j\leq m} \left(\bigwedge_{1\leq k\leq n} (\pi_j \cdot C \cdot i_k = \sum_{1\leq l\leq p} \pi_j \cdot A \cdot i_l \times \pi_l \cdot B \cdot i_k)\right)$$

This predicate expresses an outer traversal indexed by j, an inner traversal indexed by k and what the expected result in each element of output matrix C is. Thus we reach three nested for-loops of two different kinds: the two outer-loops (corresponding to indices j, k) provide for *navigation*, while the inner loop performs an *accumulation* (thus the need for initialization):

```
for $j = 1$ to $m$ do
  for $k = 1$ to $n$ do
    $C[j][k] \leftarrow 0$
    for $l = 1$ to $p$ do
      $C[j][k] \leftarrow C[j][k] + A[j][l] * B[l][k]$
    end for
  end for
end for
```

Different matrix memory mapping schemes give rise to the interchange of the j, k and l in the loop above [14]. This is due to corresponding choices in the derivation granted by the generalized exchange law (24), among others.

Other variants of blocked MMM (28) such as e.g. Strassen's or Winograd's [15] rely mainly on the additive structure of Mat_K and thus don't pose new challenges. However, whether such algorithms can be better explained in more structured, biproduct-based derivations is a matter of future research.

6 Related Work

The formulation of categories of matrices can be traced back to [6] and [16], where the focus is either on exemplifying additive categories and on the relationship between linear transformations and matrices. No effort on exploiting biproducts calculationally is present, let alone algorithm derivation.

Bloom et al [8] make use of what we have identified as the standard biproduct (enabling blocked matrix algebra) to formalize column and row-wise matrix join and fusion. Instead of a calculational approach to linear algebra algorithmics, the emphasis is on iteration theories which matricial theories are a particular case of. Furthermore this work makes use of other algebraic properties of $\mathbf{Mat}_K$ which we aim to encompass later.

Other categorial approaches to linear algebra include relative monads [17], whereby the category of finite-dimensional vector spaces arises as a kind of Kleisli category. Efforts by the mathematics of program construction community in the derivation of matrix algorithms include the study of two-dimensional pattern matching [18]. An account of work on calculational, index-free reasoning about regular and Kleene algebras of matrices can be found in [19].

7 Conclusions and Current Work

A comprehensive calculational approach to linear algebra algorithm specification, transformation and generation is still missing. However, the successes reported by the engineering field in the automatic library generation are a good cue to the feasibility of such an approach.

In this paper we have presented a formalization of matrices as categorial morphisms (arrows) in a way which relates categories of matrices to relation algebra and program calculation. Our case study — matrix multiplication — is dealt with in an elegant, calculational style whereby its divide-and-conquer and triple nested loop algorithmic implementations were derived.

The notion of a categorial biproduct is at the heart of the whole approach. Using the category of matrices and its biproducts the conversion from the declarative definition of a matrix to its indexed version is made possible thanks to the properties of projections and injections, as shown in the derivation of the triple for-loop.

We plan to carry on this work in several directions. The background of our project is the formalization of the SPL language [2] and, in this respect, work has only started. However in its beginning, our biproduct-centered approach is already telling us what to do next, as happens for instance with the biproduct nature of the *gather/scatter* matrices of the SPIRAL system [20].

Next in the plan we want to exploit other solutions to the biproduct equations, while checking which "chapters" of linear algebra [16] they are able to constructively explain. Think of Gaussian elimination, for instance, whose main steps involve row-switching, row-multiplication and row-addition, and suppose one defines the following transformation t catering for the last two, for a given α:

$$t : (\, n \longleftarrow n \,) \times (\, n+n \longleftarrow m \,) \to (\, n+n \longleftarrow m \,)$$

$$t(\alpha, \left[\frac{A}{B}\right]) = \left[\frac{A}{\alpha A + B}\right]$$

(Thinking in terms of blocks A and B rather than rows is more general; in this setting, arrow $n \xleftarrow{\alpha} n$ means $n \xleftarrow{id} n$ with all 1s replaced by αs.) Let us analyze the essence of t by using the blocked-matrix calculus in reverse order :

$$\begin{array}{ll} & t(\alpha, \left[\frac{A}{B}\right]) = \left[\frac{A}{\alpha A + B}\right] \\ = & \{ \text{ (28) in reverse order } \} \\ & \left[\begin{array}{c|c} 1 & 0 \\ \hline \alpha & 1 \end{array}\right] \cdot \left[\frac{A}{B}\right] \\ = & \{ \text{ divide-and-conquer (27) } \} \\ & \left[\frac{1}{\alpha}\right] \cdot A + \left[\frac{0}{1}\right] \cdot B \end{array}$$

It can be shown that the last expression, which has the same shape as (14), is in fact the *split* combinator generated by another biproduct, parametric on α:

$$\pi'_1 = \begin{bmatrix} 1 \; 0 \end{bmatrix}, \; \pi'_2 = \begin{bmatrix} -\alpha \; 1 \end{bmatrix}$$
$$i'_1 = \begin{bmatrix} 1 \\ \alpha \end{bmatrix}, \; i'_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

In summary, this biproduct, which extends the one studied earlier on (they coincide for $\alpha := 0$) provides a categorial interpretation of one of the steps of Gaussian elimination. We are currently investigating its role in a constructive proof of the corresponding, well-known algorithm, which we lay down recursively as follows, using block-notation (24):

$$\begin{aligned}
&ge : (\; 1+n \longleftarrow 1+m \;) \to (\; 1+n \longleftarrow 1+m \;)\\
&ge \left[\begin{array}{c|c} x & M \\ \hline N & Q \end{array}\right] = \left[\begin{array}{c|c} x & M \\ \hline 0 & ge(Q - \frac{N}{x} \cdot M) \end{array}\right]\\
&ge\ x = x
\end{aligned}$$

In particular, we want to provide a calculational alternative to the FLAME-styled derivation of the algorithm given in e.g. [3].

Last but not least, we want to address *vectorization* calculationally. The linearization of an arbitrary matrix into a vector is a data refinement step. This means finding suitable abstraction/representation relations [21] between the two formats and reasoning about them, including the refinement of all matrix operations into vector form.

The first part of the exercise proves easier than first expected: vectorization is akin to exponentiation, that is, *currying* [4] in functional languages. While currying "thins" the input of a given binary function by converting it into its unary (higher-order) counterpart, so does vectorization by thinning a given matrix $n \xleftarrow{M} km$ into $kn \xleftarrow{\mathbf{vec}\, M} m$, where k is the "thinning factor" [7]. (For $m = 1$, $\mathbf{vec}\, M$ is a column vector — the standard situation [22].) Once again, our approach relies on capturing such a relationship by a universal property

$$X = \mathbf{vec}\, M \;\Leftrightarrow\; M = \epsilon \cdot (id \otimes X)$$

$$\begin{tikzcd}
k \times n & k \times (k \times n) \arrow[r, "\epsilon"] & n \\
m \arrow[u, "\mathbf{vec}\, M"] & k \times m \arrow[u, "id_k \otimes (\mathbf{vec}\, M)"] \arrow[ur, "M"'] &
\end{tikzcd}$$

where $\otimes$ denotes Kronecker product [2], granting **vec** as a bijective transformation. So its converse **unvec** is also a bijection, whereby $\epsilon = \mathbf{unvec}\, id$. Put in other words, we are in presence of an adjunction between functor $FX = id_k \otimes X$ and itself. Taking advantage of this mathematical framework [23] in calculating the whole algebra of vectorization will keep the authors busy for a while [24].

[2] Given $p \xleftarrow{A} m$ and $q \xleftarrow{B} n$, the Kronecker product $pq \xleftarrow{A \otimes B} mn$ is the matrix $A \otimes B = (a_{ij} B)$ [22].

Broadening scope, an aspect that needs investigation is how this "non-standard" treatment of matrices (data structures represented as arrows, as opposed to datatypes as objects) combines with theories of the rest of programming. For instance, its application to the emerging field of *linear algebra of programming* [25] and its combination with the monadic framework of [17] are topics for future research.

Acknowledgements. The authors would like to thank Markus Püschel (CMU) for driving their attention to the relationship between linear algebra and program transformation. Hugo Macedo further thanks the SPIRAL group for granting him an internship at CMU.

Thanks are also due to Michael Johnson and Robert Rosebrugh (Macquarie Univ.) for pointing the authors to the categories of matrices approach. Yoshiki Kinoshita (AIST, Japan) and Manuela Sobral (Coimbra Univ.) helped with further indications in the field.

This research was carried out in the context of the Mondrian Project funded by FCT contract PTDC/EIA-CCO/108302/2008. Hugo Macedo was partially supported by the *Fundação para a Ciência e a Tecnologia*, Portugal, under grant number SFRH/BD/33235/2007.

References

1. Franchetti, F., de Mesmay, F., McFarlin, D., Püschel, M.: Operator language: A program generation framework for fast kernels. In: Taha, W.M. (ed.) Domain-Specific Languages. LNCS, vol. 5658, pp. 385–409. Springer, Heidelberg (2009)
2. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B.W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation 93(2), 232–275 (2005)
3. de Geijn, R.A.V., Quintana-Ortí, E.S.: The Science of Programming Matrix Computations (2008), `http://www.lulu.com`
4. Jones, S.P., Hughes, J., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C., Wadler, P.: Report on the programming language Haskell 98 — a non-strict, purely functional language. Technical report (February 1999)
5. Bird, R., de Moor, O.: Algebra of Programming. In: Hoare, C.A.R. (series ed.). Series in Computer Science. Prentice-Hall International, Englewood Cliffs (1997)
6. MacLane, S.: Categories for the Working Mathematician (Graduate Texts in Mathematics). Springer, Heidelberg (September 1998)
7. Macedo, H., Oliveira, J.: Matrices as arrows: a typed approach to linear algebra, Extended abstract. In: CALCO-JNR Workshop, September 6-10, Udine, Italy (2009)
8. Bloom, S., Sabadini, N., Walters, R.: Matrices, machines and behaviors. Applied Categorical Structures 4(4), 343–360 (1996)
9. Allenby, R.B.J.T.: Linear Algebra. Elsevier, Amsterdam (1995)

10. Tarski, A., Givant, S.: A Formalization of Set Theory without Variables. In: AMS, Providence, Rhode Island, vol. 41. AMS Col. Pub, Washington (1987)
11. Frias, M.: Fork algebras in algebra, logic and computer science, Logic and Computer Science. World Scientific Publishing Co., Singapore (2002)
12. Wolfram, S., et al.: Mathematica: a system for doing mathematics by computer. Addison-Wesley, New York (1988)
13. Bird, R.: Lecture notes on constructive functional programming. In: Broy, M. (ed.) CMCS Int. Summer School directed by Bauer, F.L., et al. NATO Adv. Science Institute, Series F: Comp. and System Sciences, vol. 55, Springer, Heidelberg (1989)
14. Goto, K., Geijn, R.A.v.d.: Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. 34(3), 1–25 (2008)
15. D'Alberto, P., Nicolau, A.: Adaptive Strassen's matrix multiplication. In: ICS '07: Proc. of the 21st annual int. conf. on Supercomputing. ACM, NY (2007)
16. MacLane, S., Birkhoff, G.: Algebra. AMS Chelsea (1999)
17. Altenkirch, T., Chapman, J., Uustalu, T.: Monads need not be endofunctors. In: Foundations of Software Science and Computational Structures, pp. 297–311 (2010)
18. Jeuring, J.: The derivation of hierarchies of algorithms on matrices. In: Moller, B. (ed.) Constructing Programs from Specifications, pp. 9–32. North-Holland, Amsterdam (1991)
19. Backhouse, R.: Mathematics of Program Construction, Univ. of Nottingham, Draft of book in preparation, 608 pages (2004)
20. Voronenko, Y.: Library Generation for Linear Transforms. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University (2008)
21. Oliveira, J.N.: Transforming data by calculation. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering II. LNCS, vol. 5235, pp. 134–195. Springer, Heidelberg (2008)
22. Magnus, J., Neudecker, H.: The commutation matrix: Some properties and applications. The Annals of Statistics 7(2), 381–394 (1979)
23. Došen, K., Petrić, Z.: Self-adjunctions and matrices. Journal of Pure and Applied Algebra 184, 7–39 (2003)
24. Macedo, H.D., Oliveira, J.N.: Exploring self-adjunctions in vectorization (2010) (in preparation)
25. Sernadas, A., Ramos, J., Mateus, P.: Linear algebra techniques for deciding the correctness of probabilistic programs with bounded resources. Technical report, SQIG-IT and TU Lisbon, 1049-001 Lisboa, Portugal, — Short paper presented at LPAR 2008, Doha, Qatar. November 22-27 (2008)

Lucy-n: a n-Synchronous Extension of Lustre[⋆]

Louis Mandel[1], Florence Plateau[1], and Marc Pouzet[1,2]

[1] LRI, Univ. Paris-Sud 11, Orsay, France and INRIA Saclay
[2] Institut Universitaire de France

Abstract. Synchronous functional languages such as Lustre or Lucid Synchrone define a restricted class of Kahn Process Networks which can be executed with no buffer. Every expression is associated to a clock indicating the instants when a value is present. A dedicated type system, the clock calculus, checks that the actual clock of a stream equals its expected clock and thus does not need to be buffered. The n-synchrony relaxes synchrony by allowing the communication through bounded buffers whose size is computed at compile-time. It is obtained by extending the clock calculus with a subtyping rule which defines buffering points.

This paper presents the first implementation of the n-synchronous model inside a Lustre-like language called Lucy-n. The language extends Lustre with an explicit `buffer` construct whose size is automatically computed during the clock calculus. This clock calculus is defined as an inference type system and is parametrized by the clock language and the algorithm used to solve subtyping constraints. We detail here one algorithm based on the abstraction of clocks, an idea originally introduced in [5]. The paper presents a simpler, yet more precise, clock abstraction for which the main algebraic properties have been proved in Coq. Finally, we illustrate the language on various examples including a video application.

Keywords: Process networks, Synchronous model, Type systems.

1 Introduction

This paper focuses on programming models and languages for implementing real time streaming applications as found in video systems. These applications transform infinite streams of pixels through successive *filters* and are thus naturally expressed as Kahn Process Networks [8]. In this model, processes execute concurrently and communicate through unbounded FIFO buffers with blocking reads when the buffer is empty and non blocking writes. The model is deterministic (a network defines a stream function) and is delay insensitive (computation and communication time do not change the network semantics). Kahn networks with bounded buffers can be implemented by adding a *back pressure* mechanism in order to avoid writes into a full buffer. Nonetheless, this may introduce artificial blocking when the size of buffers have been underestimated. The size of

[⋆] This work is funded by the French "Action d'envergure" INRIA Synchronics.

buffers can be increased dynamically [11] but this solution cannot be used for real time applications where execution in bounded memory must be guaranteed at compile time.

To know whether a Kahn network is deadlock free or can be executed in bounded memory is undecidable in the general case [1]. *Synchronous Data Flow* (or SDF) [9] and its variants (*Cyclo Static Data Flow* [10] among others) are restricted classes of networks where every node consumes and produces a fixed number of tokens at every step. The size of buffers can be computed at compile time and a periodic static schedule can be generated. This makes SDF a good candidate for modeling and programming video intensive applications with periodic behavior [14].

Synchronous languages such as Lustre [2] or Lucid Synchrone [13] also define a restricted class of Kahn Networks [3] as they can be executed without any implicit buffering (i.e., synchronously). They are not limited to periodic behavior and ensure strong safety properties at compile-time such as determinism and absence of deadlock. Moreover, they can be compiled into statically scheduled executable code. Nonetheless, they do not offer the same flexibility as SDF-like tools do. Buffers have to be inserted manually and their size computed adequately which is both difficult and error-prone. We thus want to extend synchronous languages with conveniences to communicate through bounded buffers, like in SDF.

In synchronous languages, time is defined as the succession of discrete instants. In a data-flow framework, every stream s is associated to a boolean sequence or *clock* with value `1` at instants at which s is present and `0` otherwise. Two streams can be composed (e.g., added) without any buffer when their clocks are equal. The purpose of the *clock calculus* is to give sufficient condition for a system to be executed synchronously. This is essentially a typing problem [3,6]. Every expression is given a clock type (or simply type) and must satisfy a typing rule such as:

$$\frac{H \vdash e_1 : ck_1 \mid C_1 \qquad\qquad H \vdash e_2 : ck_2 \mid C_2}{H \vdash e_1 \texttt{ + } e_2 : ck_3 \mid \{ck_1 \;=\; ck_2 \;=\; ck_3\} \cup C_1 \cup C_2}$$

This rule states that under the typing environment H, if e_1 has type ck_1 under the constraints C_1 and if e_2 has type ck_2 under the constraints C_2, then $e_1 \texttt{ + } e_2$ has type ck_3 under the constraint that $ck_1 \;=\; ck_2 \;=\; ck_3$ and the constraints C_1 and C_2. Equality of types ensures equality of clocks. Hence, the composition of two flows of same type can be done without buffer. Synchronous languages only consider equality constraints. The n-synchrony [4] relaxes these constraints by allowing to compose streams whose type are not equal but can be synchronized through the introduction of a bounded buffer. If a stream x with type ck can be consumed later with type ck' using a bounded buffer, we shall say that ck is a subtype of ck' and we write $ck <: ck'$. We extend the language with a `buffer` construct which indicates the points where the subtyping rule should be applied.

$$\frac{H \vdash e : ck \mid C}{H \vdash \texttt{buffer}\; e : ck' \mid \{ck <: ck'\} \cup C}$$

In terms of sequences of values, `buffer` e is equivalent to e but it may delay its input using a bounded buffer. The `buffer` construct gives more freedom to the designer while preserving an execution in bounded memory.

The purpose of the extended clock calculus is to check that bounds exist for buffer sizes and to compute them. To this aim, subtyping constraints have to be solved. In order to reduce the algorithmic complexity of constraints resolution, an abstraction of clocks has been introduced in [5]. It consists in reasoning on sets of clocks (or *envelopes*) defined by an asymptotic rate and two shifts bounding the potential delay with respect to this rate. Then, subtyping constraints can be replaced by linear constraints on those rates and shifts and solved with a tool such as Glpk. On several examples such as the *Picture in Picture* given at the end of the paper, the over-estimation due to the abstraction is small with respect to the exact solution.

Contribution and Organization of the Paper. This paper presents the design and implementation of an n-synchronous extension of Lustre called Lucy-n. The clock calculus is generic in the sense that it is parametrized by the clock language and the algorithm used to solve subtyping constraints. In this paper, we present an algorithm using clock abstraction. The abstraction presented in [5] has been improved in various ways: the formulae are simpler; the abstraction is more precise and no restrictions are imposed anymore on clocks when computing their abstraction. Moreover, the precision of abstract operators has been studied. Finally, the main algebraic properties and the correctness of the abstraction have been proved in Coq (1800 lines of specification and 7000 lines of proof).

The paper is organized in the following way. The language is presented in Section 2. Some algebraic properties on boolean sequences are stated in Section 3. We present the basics of the clock calculus in Section 4. We then introduce the improved version of clock abstraction in Section 5 followed by the constraint solving algorithm in Section 6. The implementation is discussed in Section 7. Finally, we illustrate the use of the language on a video application in Section 8.

All examples presented in the paper have been programmed in Lucy-n and buffer sizes have been computed automatically. The prototype is available at `http://www.lri.fr/~plateau/mpc10`. Definitions and properties that have been proved in Coq are marked with ✿ which is a link to the corresponding code. We give a proof sketch for each property. Full proofs on paper are only available in a French document [12].

2 The Language

We consider a first-order synchronous dataflow language reminiscent of Lustre but extended with an explicit buffering operator. The syntax is given in Figure 1. A program (d) is a sequence of definitions of stream functions called *nodes* and definitions of clock names (c). The inputs of a node are described by a pattern (pat) and its body is an expression (e). The operators are the basic ones of Lucid Synchrone and their intuitive semantics is detailed later. e_1 *op* e_2 denotes

d	::=	| `let node` f $pat = e$	node definition
		| `let clock` $c = ce$	clock definition
		| d d	sequence of definitions
pat	::=	x | $(pat,\dots,pat)$	pattern
e	::=	| i	constant flow
		| x	flow variable
		| $(e,\dots,e)$	tuple
		| e op e	imported operator
		| `if` e `then` e `else` e	mux operator
		| f e	node application
		| e `where rec` eqs	local definitions
		| e `fby` e	initialized delay
		| e `when` ce | e `whenot` ce	sampling
		| `merge` ce e e	merging
		| `buffer` e	buffering
eqs	::=	$pat = e$ | eqs `and` eqs	mutually recursive equations

Fig. 1. Language kernel

the point-wise application of a binary operator; `if` e_1 `then` e_2 `else` e_3 is the point-wise application of a conditional; f e is the application of a node f to an expression e; e_1 `fby` e_2 conses the head of e_1 to e_2 (and thus corresponds to an initialized delay); e `when` ce samples a stream e according to a clock expression ce whereas `merge` ce e_1 e_2 merges two streams with complementary clocks. Finally `buffer` e buffers e. We write e `where rec` eqs for an expression defined by a collection of mutually recursive equations (eqs). In this paper, we restrict the clock language ce to define ultimately periodic boolean sequences only:

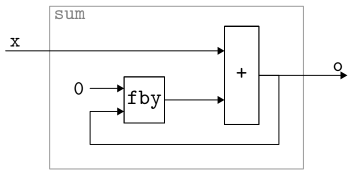
$$\begin{array}{lcl} ce & ::= & c \mid u(v) \\ u & ::= & \varepsilon \mid 0.u \mid 1.u \\ v & ::= & 0 \mid 1 \mid 0.v \mid 1.v \end{array}$$

It can be a variable name (c) or a periodic word ($u(v)$) made of a finite prefix (u) followed by the infinite repetition of a binary word (v). For example, `(10)` defines the half sequence `101010`...

A First Program. Let us write a node that sums the values taken by its input. We depict the corresponding block diagram on the right and give the clock type signature produced by the compiler.

```
let node sum x = o where
  rec o = x + (0 fby o)

val sum :: forall 'a. 'a -> 'a
```

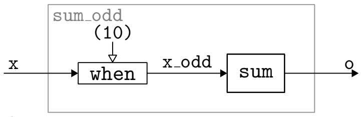


At the first instant, `0 fby o` is equal to `0` then it is equal to the previous value of `o`. The type of `sum` means that for any clock c given to the input `x`, then the output of `sum x` has the same clock c. Here is an example of the execution of `sum x` on the input sequence `x` $= 5, 7, 3, \ldots$ set on clock `(1)`:

flow	values							clock
`x`	5	7	3	6	2	8	...	`(1)`
`0 fby o`	0	5	12	15	21	23	...	`(1)`
`o`	5	12	15	21	23	31	...	`(1)`

Sampling and Merging Streams. We now introduce two special operators to remove and add values on a stream. The expression e `when` ce returns a subsequence of e keeping the values of e at the instants where ce equals `1`. For example, the sum of elements of odd index is:

```
let node sum_odd x = o where
  rec x_odd = x when (10)
  and o = sum(x_odd)
```



```
val sum_odd :: forall 'a. 'a -> 'a on (10)
```

On the input sequence of the previous example, we get:

flow	values							clock
`x`	5	7	3	6	2	8	...	`(1)`
`(10)`	1	0	1	0	1	0	...	`(1)`
`x_odd`	5		3		2		...	`(10)`
`o`	5		8		10		...	`(10)`

We can observe that `x` is present at each instant and that `x_odd` and `o` are defined one instant over two. Thus, the clock of `x` is `(1)` and the one of `x_odd` and `o` is `(10)`. The clock of `x_odd` is the clock of `x when (10)`, it can be computed from the clock of the flow `x` and the sampling condition `(10)`. This is the result of the operation `(1)` *on* `(10)` defined below.

Definition 1 (*on* Operator). ✿

$$
\begin{aligned}
0.w_1 \ \mathit{on} \ w_2 &\overset{def}{=} 0.(w_1 \ \mathit{on} \ w_2) \\
1.w_1 \ \mathit{on} \ 1.w_2 &\overset{def}{=} 1.(w_1 \ \mathit{on} \ w_2) \\
1.w_1 \ \mathit{on} \ 0.w_2 &\overset{def}{=} 0.(w_1 \ \mathit{on} \ w_2)
\end{aligned}
$$

In the previous example, we have considered an input flow on the base clock `(1)` (true all the time) but this is not necessarily the case. Indeed, `x` could in particular be the result of a sampling and be on clock `(110)` for example. In that case, the clock of `x_odd` would be `(110)` *on* `(10)` which is equal to `(100)` according to the definition of *on*. The diagram below illustrates what happen when several sampling are composed.

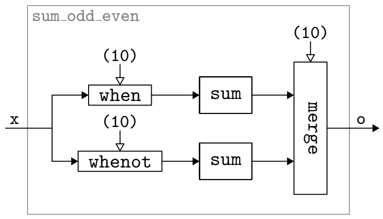
flow	values	clock
`x`	5 7 3 6 ...	`(110)`
`x when (10)`	5 3 ...	`(110)` *on* `(10)`

The type signature for `sum_odd`, that is, $\forall \alpha . \alpha \rightarrow \alpha$ `on (10)` reflects the fact that the clock of the output stream is a sub-clock of the input stream. It states that for any clock c, if `x` has clock c then the result has clock c *on* `(10)`. To avoid any possible confusion, we write on purpose `on` for the type constructor whereas *on* stands for its interpretation on boolean streams.

As opposed to sampling, the expression `merge` ce e_1 e_2 allows to combine two streams e_1 and e_2 on complementary clocks. When ce is true, the output of the merge is the current value of e_1 while e_2 is not consumed; otherwise, the current value of e_2 is produced and e_1 does not progress. For example, the following node splits `x` in two subsequences, instantiates `sum` on each and finally merges them.

```
let node sum_odd_even x = o where
  rec x_odd = x when (10)
  and x_even = x whenot (10)
  and o =
    merge (10) (sum x_odd) (sum x_even)

val sum_odd_even :: forall 'a. 'a -> 'a
```



We get the following chronogram:[1]

flow	values							clock
`x`	5	7	3	6	2	8	...	`(1)`
`(10)`	1	0	1	0	1	0	...	`(1)`
`x_odd`	5		3		2		...	`(1)` *on* `(10)` = `(10)`
`x_even`		7		6		8	...	`(1)` *on not* `(10)` = `(01)`
`sum x_odd`	5		8		10		...	`(10)`
`sum x_even`		7		13		21	...	`(01)`
`o`	5	7	8	13	10	21	...	`(1)`

n-Synchronous Communication. As in Lustre, communication is synchronous. As a consequence, the following program is rejected:

```
let node bad x = x + (x when (10))

File "bad.ls", line 1, characters 17-33:
Cannot unify clock 'a2 on (10) with clock 'a2
```

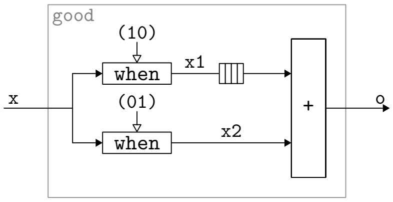
Indeed, `x` and `x when (10)` have respectively type α and α `on (10)` whereas `+` expects its two arguments to have the same type.

When the `buffer` primitive is used, communication is n-synchronous. That means that it can be made synchronous through the insertion of a bounded

[1] *not* ce is the point-wise application of the negation operator to ce.

buffer which size is computed automatically. Even using this `buffer` construct, the previous example cannot be accepted since it would need an infinite buffer to synchronize `x` and `x when (10)`. Here is an example of a perfectly valid n-synchronous program.

```
let node good x = o where
  rec x1 = x when (10)
  and x2 = x when (01)
  and o = (buffer x1) + x2
```



The compiler outputs the type and the buffer size needed:

```
val good :: forall 'a. 'a -> 'a on (01)
Buffer line 4, characters 10-20: size = 1
```

As an example, we get:

flow	values							clock
`x`	5	7	3	6	2	8	...	`(1)`
`x1`	5		3		2		...	`(10)`
`buffer(x1)`		5		3		2	...	`(01)`
`x2`		7		6		8	...	`(01)`
`buffer(x1) + x2`		12		9		10	...	`(01)`

Semantically, `buffer` is the identity function, it only delays its input. The use of a buffer is accepted provided the input clock of the buffer is *adaptable* to the output clock. The next section defines the adaptability relation between clocks.

3 Clock Adaptability

Here is the intuition of adaptability: *a clock w_1 is adaptable to clock w_2 if any stream with clock w_1 can be consumed with clock w_2 up to the insertion of a bounded buffer.*

To properly define this relation, we introduce the *cumulative function* of a binary word: for any binary word w, $\mathcal{O}_w(i)$ counts the number of `1`s up to the index i. Figure 2 shows the cumulative functions of $w_1 =$ `(11010)` and $w_2 =$ `0(00111)`.

Definition 2 (Elements and Cumulative Function of w). ✿
Let $w = b.w'$ with $b \in \{0, 1\}$. We write $w[i]$ for the i-th element of w:

$$\begin{array}{rcl} w[1] & \overset{def}{=} & b \\ \forall i > 1,\ w[i] & \overset{def}{=} & w'[i-1] \end{array}$$

We write $\mathcal{O}_w$ for the cumulative function of w:

$$\begin{array}{rcl} \mathcal{O}_w(0) & \overset{def}{=} & 0 \\ \forall i \geq 1,\ \mathcal{O}_w(i) & \overset{def}{=} & \begin{cases} \mathcal{O}_w(i-1) & \text{if } w[i] = 0 \\ \mathcal{O}_w(i-1) + 1 & \text{if } w[i] = 1 \end{cases} \end{array}$$

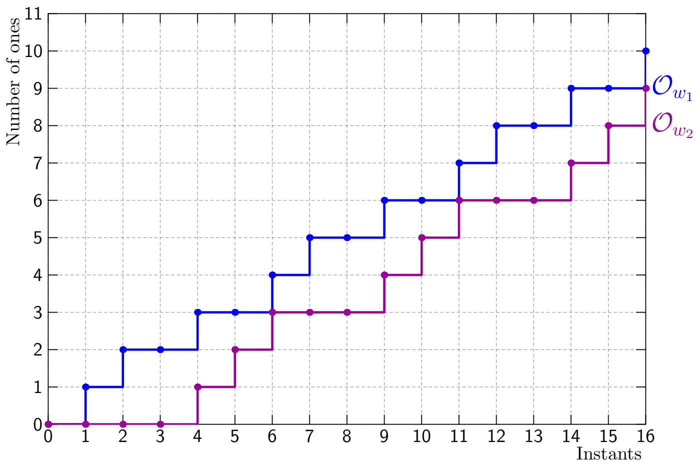


Fig. 2. Cumulative functions for $w_1 =$ `(11010)` and $w_2 =$ `0(00111)`

Adaptability is the conjunction of two relations: *precedence* and *synchronizability*. Precedence ensures that there is no read in an empty buffer, that is at each instant, more values have been written than read in the buffer. Synchronizability ensures that the number of values present in the buffer during the execution is bounded.

Definition 3 (Synchronizability $\bowtie$, Precedence $\preceq$, Adaptability $<:$)

$$\begin{array}{lcl} w_1 \bowtie w_2 & \overset{def}{\Leftrightarrow} & \exists b_1, b_2 \in \mathbb{Z}, \forall i \geq 0,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2 \\ w_1 \preceq w_2 & \overset{def}{\Leftrightarrow} & \forall i > 0,\ \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i) \\ w_1 <: w_2 & \overset{def}{\Leftrightarrow} & w_1 \preceq w_2\ \wedge\ w_1 \bowtie w_2 \end{array}$$ ✿

In Figure 2, $w_1 \bowtie w_2$ since the vertical distance between the two curves is bounded and $w_1 \preceq w_2$ since the curve $\mathcal{O}_{w_1}$ is always above the one of $\mathcal{O}_{w_2}$.

Buffer Size. Consider a buffer with an input clock w_1 and output clock w_2. For every instant i, the number of elements present in the buffer is:

$$size_i(w_1, w_2) = \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)$$ ✿

A negative value means that there were more reads than writes and this case should not appear. A sufficient size for the buffer is the maximal number of values present in the buffer during the execution:

$$size(w_1, w_2) = \max_{i \geq 1}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$$ ✿

Thus, if w_1 is adaptable to w_2, a stream with clock w_1 can be safely consumed on the clock w_2 by insertion of a bounded buffer. Otherwise, the size of the buffer may be infinite.

Theorem 1 (Communication Through a Buffer). ✿

$$w_1 <: w_2 \;\Rightarrow\; \exists b, \forall i,\ 0 \leq size_i(w_1, w_2) \leq b$$

Proof: By definition of the synchronizability relation and the formula giving the number of elements present in the buffer at each instant, we know that there exists a value b such that $\forall i,\ size_i(w_1, w_2) \leq b$. By definition of the precedence relation, we have $\forall i,\ size_i(w_1, w_2) \geq 0$. □

As ultimately periodic words have a repetitive behavior after a certain index, checking adaptability relation and computing buffer sizes can be done statically [4].

4 Relaxed Clock Calculus

The purpose here is to check that programs can be evaluated using bounded buffers and to find those bounds. As seen in Section 1, it is done using typing techniques where types represents clocks. In this paper, we consider the following type language:

$$\begin{array}{lcl} \sigma & ::= & \forall \alpha_1, \ldots, \alpha_n. (ck \times \ldots \times ck) \rightarrow (ck \times \ldots \times ck) \\ ck & ::= & \alpha \mid ck \ \texttt{on}\ ce \mid ck\ \texttt{on not}\ ce \end{array}$$

A type scheme (σ) represents the type of a node. It describes types of output streams with respect to types of input streams. Types of streams (ck) can be either a type variable (α) or the type of a sampled stream (ck `on` ce, ck `on not` ce).

Typing judgments are of the form: $H \vdash e : ck \,|\, C$ meaning that under type environment H, the expression e has the type ck provided type constraint C are satisfied. An environment H associates type schemes, types, and clock expressions respectively to node names, stream variables and clock variables:

$$H ::= ([\, f_1 : \sigma_1, \ldots, f_m : \sigma_m \,],\ [\, x_1 : ck_1, \ldots, x_p : ck_p \,],\ [\, c_1 : ce_1, \ldots, c_n : ce_n \,])$$

A set C of typing constraints contains equality constraints and subtyping constraints. It is of the form:

$$C ::= [ck_1 \ = \ ck'_1, ..., ck_n \ = \ ck'_n, \quad ck_{n+1} <: ck'_{n+1}, ..., ck_m <: ck'_m]$$

Typing constraints are gathered progressively during typing and this raises no particular difficulty. We simply illustrate what is done on the example `good` given in the previous section.

An example. The node `good` is rewritten as follows to make the typing derivation smaller.

```
let node good x = buffer (x when (10)) + x when (01)
```

We need to use the following typing rules in addition to the ones given in Section 1:

$$\frac{H(x) = ck}{H \vdash x : ck \,|\, \emptyset} \qquad \frac{H \vdash e : ck \;|\; C}{H \vdash e \texttt{ when } ce : ck \texttt{ on } ce \;|\; C}$$

Thereby, if we associate the type variable α_x to the input of node `good`, the typing of the left and the right branches of the `+` operator gives the following two derivations:

$$A: \quad \frac{\dfrac{\texttt{x} : \alpha_x \vdash \texttt{x} : \alpha_x \,|\, \emptyset}{\texttt{x} : \alpha_x \vdash \texttt{x when (10)} : \alpha_x \; \textit{on} \; \texttt{(10)} \,|\, \emptyset}}{\texttt{x} : \alpha_x \vdash \texttt{buffer (x when (10))} : \alpha_b \,|\, \{\alpha_x \; \textit{on} \; \texttt{(10)} \texttt{ <: } \alpha_b\}}$$

$$B: \quad \frac{\texttt{x} : \alpha_x \vdash \texttt{x} : \alpha_x \,|\, \emptyset}{\texttt{x} : \alpha_x \vdash \texttt{x when (01)} : \alpha_x \; \textit{on} \; \texttt{(01)} \,|\, \emptyset}$$

Now, using the typing rule of `+`, the body of `good` gives the following derivation:

$$\frac{A \qquad\qquad B}{\texttt{x} : \alpha_x \vdash \texttt{buffer (x when (10)) + x when (01)} : \alpha_o \,|\, \{\alpha_b = \alpha_x \; \textit{on} \; \texttt{(01)} = \alpha_o, \; \alpha_x \; \textit{on} \; \texttt{(10)} \texttt{ <: } \alpha_b\}}$$

The type of `good` is $\alpha_x \rightarrow \alpha_o$ provided the following system of constraints is satisfied:

$$\{\alpha_b = \alpha_x \; \textit{on} \; \texttt{(01)} = \alpha_o, \alpha_x \; \textit{on} \; \texttt{(10)} \texttt{ <: } \alpha_b\}$$

Constraint solving. There are two kinds of constraints in the system: equality and subtyping constraints. In order to solve equality constraints, synchronous languages such as Lustre or Lucid Synchrone use structural unification over clock types. It means that two types of the form ck_1 `on` ce_1 and ck_2 `on` ce_2 can be unified if and only if ce_1 is equal to ce_2 and if ck_1 can be unified with ck_2. Here, since we use only ultimately periodic clocks, the `on` operator can be interpreted and we can use a unification algorithm such as the one presented in [4]. None of these two unification techniques is complete, and they may fail on different cases. So, to be conservative over Lustre but more expressive, it is possible to use interpreted unification only after structural simplification of the constraints. We call this technique semi-interpreted unification.

In the example, if we choose the instantiation $\alpha_x = \alpha$, $\alpha_b = \alpha \; \textit{on} \; \texttt{(10)}$ and $\alpha_o = \alpha \; \textit{on} \; \texttt{(10)}$, then the constraint $\alpha_b = \alpha_x \; \textit{on} \; \texttt{(01)} = \alpha_o$ is always satisfied and the set of constraints reduces to $\{\alpha \; \textit{on} \; \texttt{(10)} \texttt{ <: } \alpha \; \textit{on} \; \texttt{(10)}\}$.

In order to solve subtyping constraints, we must find instantiations of type variables such that constraints take the form α `on` w_1 `<:` α `on` w_2 with $w_1 <: w_2$.

This is simple to achieve with the example `good` since the constraint is α *on* `(10)` `<:` α *on* `(10)` and then, already in this form. If this is not the case, finding such a solution is computationally expensive. A first solution was experimented so as to convert adaptability constraints into a system of linear inequations. Nonetheless, the number of linear inequations is proportional to the number of `1`s for each adaptability constraint. In order to overcome this complexity, we propose in this paper not to consider exact periodic clocks but their abstraction. The basic principles and algebraic properties of clock abstractions have been introduced in [5]. In this paper, we present an improved version. Moreover, its algebraic properties have been proved in Coq.

5 Abstraction of Binary Words

The idea behind abstraction is to reason on sets of binary words. An abstraction bounds the cumulative function of a set of words by two linear curves with the same slope. Thus, the abstraction of an infinite binary word w keeps only the asymptotic proportion r of `1`s in w and two values b^0 and b^1 which give the minimum and maximum shift of `1`s in w compared to r. This abstract information is called an *envelope* and noted $\langle b^0, b^1\rangle(r)$.

Definition 4 (Concretization). ✿

$$concr\left(\langle b^0, b^1\rangle(r)\right) \overset{def}{=} \left\{ w \mid \forall i \geq 1, \quad \wedge \begin{array}{lcl} w[i] = \mathtt{1} & \Rightarrow & \mathcal{O}_w(i) \leq r \times i + b^1 \\ w[i] = \mathtt{0} & \Rightarrow & \mathcal{O}_w(i) \geq r \times i + b^0 \end{array} \right\}$$

with $b^0, b^1, r \in \mathbb{Q}$ *and* $0 \leq r \leq 1$.

The words $w_1 = \mathtt{(11010)}$ and $w_2 = \mathtt{0(00111)}$ seen previously are respectively in envelopes $a_1 = \langle 0, \frac{4}{5}\rangle\left(\frac{3}{5}\right)$ and $a_2 = \langle -\frac{9}{5}, -\frac{3}{5}\rangle\left(\frac{3}{5}\right)$ shown in Figure 3. In chronograms, an abstract value $\langle b^0, b^1\rangle(r)$ is represented by two lines $\Delta^1 :\ r \times i + b^1$ and $\Delta^0 :\ r \times i + b^0$ that bound the cumulative functions of a set of binary words. The definition states that any rising edge must be below the line Δ^1 (solid line) and any absence of a rising edge must be above the line Δ^0 (dashed line).

For the set of words defined by an envelope to be non-empty, the line Δ^1 must be above the line Δ^0. At each instant, there must be a discrete value between the two lines. It is the case if the distance between them respects the following constraint.

Proposition 1 (Non-empty envelope). ✿ ✿

$$\forall a = \left\langle \frac{k^0}{\ell}, \frac{k^1}{\ell} \right\rangle \left(\frac{n}{\ell}\right),\ \frac{k^1}{\ell} - \frac{k^0}{\ell} \geq 1 - \frac{1}{\ell} \Rightarrow concr\,(a) \neq \varnothing$$

Proof: The proof is based on the use of the words $early_a$ and $late_a$ related to the envelope $a = \langle b^0, b^1\rangle(r)$ and defined by:

$$early_a[i] \overset{def}{=} \begin{cases} \mathtt{1} & \text{if } \mathcal{O}_{early_a}(i-1) + 1 \leq r \times i + b^1 \\ \mathtt{0} & \text{otherwise} \end{cases} \quad ✿$$

$$late_a[i] \overset{def}{=} \begin{cases} \mathtt{0} & \text{if } \mathcal{O}_{late_a}(i-1) + 0 \geq r \times i + b^0 \\ \mathtt{1} & \text{otherwise} \end{cases} \quad ✿$$

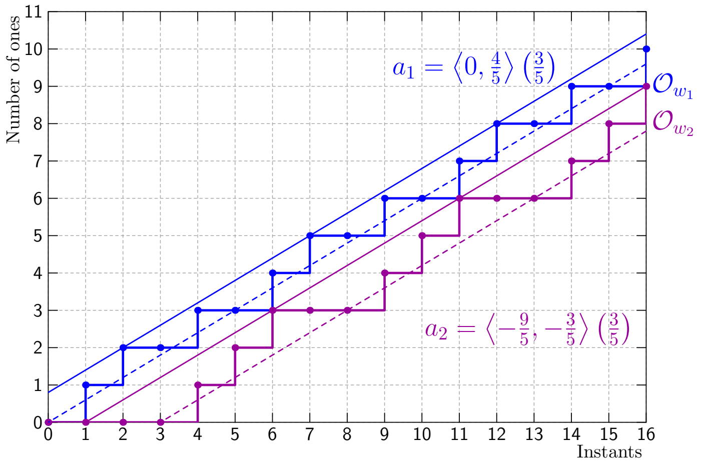


Fig. 3. Envelopes of w_1 and w_2

These words are such that $concr\,(a) = \{w \mid early_a \preceq w \preceq late_a\}$. They are used to prove most of the properties.

To prove that an envelope is non-empty, we have to prove that $early_a \preceq late_a$. ✿ This can be done by arithmetic manipulation once we have noticed that:

$$\begin{array}{lrcl} \forall i, & \mathcal{O}_{early_a}(i) & = & \max(0, \min(i, \lfloor r \times i + b^1 \rfloor)) \quad ✿✿ \\ & \mathcal{O}_{late_a}(i) & = & \max(0, \min(i, \lceil r \times i + b^0 \rceil)) \quad ✿✿ \end{array}$$

□

The abstraction of a periodic binary word can be computed automatically.

Definition 5 (Abstraction of a Periodic Word)
Let $p = u(v)$ a periodic binary word. We define $abs\,(p) \overset{def}{=} \langle b^0, b^1 \rangle\,(r)$ with:

$$\begin{array}{rcl} r & = & rate(p) \;=\; \frac{|v|_1}{|v|} \\ b^0 & = & \min_{i=1..|u|+|v| \text{ with } p[i]=0} (\mathcal{O}_p(i) - r \times i) \\ b^1 & = & \max_{i=1..|u|+|v| \text{ with } p[i]=1} (\mathcal{O}_p(i) - r \times i) \end{array}$$

where $|u|$ is the length of u and $|u|_1$ its number of 1*s.*

The asymptotic rate r corresponds to the ratio between the number of 1s in the periodic pattern and its length. To compute b^0 and b^1, the word must be traversed. The shift b^0 is the minimum difference when a 0 occurs between the number of 1s seen at instant i and the ideal value $r \times i$. The shift b^1 is the maximal difference between these values when a 1 occurs.

5.1 Abstract Operations and Relations

The interest of the abstraction is to reduce the complexity of exact computations and decisions on binary words by transforming them into arithmetic manipulations on rational numbers. For example, the computation of $\texttt{on}$ on envelopes only needs three multiplications and two additions:

Definition 6 ($\texttt{on}^\sim$ Operator). ✿ *Let $b^0{}_1 \leq 0$ and $b^0{}_2 \leq 0$.*[2] *We define:*

$$\begin{array}{llclcl} & \langle & b^0{}_1 & , & b^1{}_1 & \rangle\,(\;r_1\;) \\ \texttt{on}^\sim & \langle & b^0{}_2 & , & b^1{}_2 & \rangle\,(\;r_2\;) \\ \overset{def}{=} & \langle & b^0{}_1 \times r_2 + b^0{}_2 & , & b^1{}_1 \times r_2 + b^1{}_2 & \rangle\,(\,r_1 \times r_2\,) \end{array}$$

The elements of w_1 $\texttt{on}$ w_2 are the elements of w_1 filtered by the elements of w_2. The rate of $\texttt{1}$ in w_1 $\texttt{on}$ w_2 is thus the product of the rate of w_1 and the one of w_2. When w_1 is sampled by w_2, its shifts are multiplied by r_2. The shifts of w_2 are added to those of w_1. Consequently, this $\texttt{on}^\sim$ operator is correct:

Proposition 2 (Correctness of $\texttt{on}^\sim$). ✿ *The following property holds:*

$$\forall w_1 \in concr\,(a_1)\,,\ \forall w_2 \in concr\,(a_2)\,,\ w_1 \;\texttt{on}\; w_2 \in concr\,(a_1 \;\texttt{on}^\sim\; a_2)$$

Proof: Based on the computations of $early_{a_1}$ $\texttt{on}$ $early_{a_2}$, $late_{a_1}$ $\texttt{on}$ $late_{a_2}$ and $early_{(a_1\ \texttt{on}^\sim\ a_2)}$, $late_{(a_1\ \texttt{on}^\sim\ a_2)}$. □

The negation of binary words can also be computed on envelopes.

Definition 7 (The Operator $\texttt{not}^\sim$). ✿ *The following property holds:*

$$\texttt{not}^\sim \left\langle b^0, b^1 \right\rangle (r) \overset{def}{=} \left\langle -b^1, -b^0 \right\rangle (1 - r)$$

Proposition 3 (Correctness of $\texttt{not}^\sim$). ✿ *The following property holds:*

$$\forall w \in concr\,(a)\,,\ \texttt{not}\ w \in concr\,(\texttt{not}^\sim\ a)$$

Proof: By definition 4 and by noticing that $\mathcal{O}_{\texttt{not}\ w}(i) = i - \mathcal{O}_w(i)$. □

Now, we formulate definitions of Section 3 in the abstract domain. A relation is satisfied on envelopes if it is satisfied on all couple of words of their respective concretization sets.

Definition 8 (Abstract Synchronizability $\bowtie^\sim$, Precedence $\preceq^\sim$, Adaptability $<:^\sim$)

$$\begin{array}{lcll} a_1 \bowtie^\sim a_2 & \overset{def}{\Leftrightarrow} & \forall w_1 \in concr\,(a_1)\,, w_2 \in concr\,(a_2)\,,\ w_1 \bowtie w_2 & ✿ \\ a_1 \preceq^\sim a_2 & \overset{def}{\Leftrightarrow} & \forall w_1 \in concr\,(a_1)\,, w_2 \in concr\,(a_2)\,,\ w_1 \preceq w_2 & ✿ \\ a_1 <:^\sim a_2 & \overset{def}{\Leftrightarrow} & \forall w_1 \in concr\,(a_1)\,, w_2 \in concr\,(a_2)\,,\ w_1 <: w_2 & ✿ \end{array}$$

These relations can be tested by arithmetic comparisons on rates and shifting.

[2] We can always lose precision on the envelopes to satisfy this condition. More details are given in [12]. The chapter about clock abstraction will be translated in English.

Proposition 4 (Synchronizability, Precedence, Adaptability Tests)
Let the two envelopes $a_1 = \langle {b^0}_1, {b^1}_1 \rangle\,(r_1)$ *and* $a_2 = \langle {b^0}_2, {b^1}_2 \rangle\,(r_2)$*. We have:*

$$\begin{array}{lcll}
r_1 = r_2 & \Leftrightarrow & a_1 \bowtie^{\sim} a_2 & & ✿\ ✿ \\
{b^1}_2 - {b^0}_1 < 1 & \Rightarrow & a_1 \preceq^{\sim} a_2 & \textit{if } r_1 = r_2 & ✿\ ✿ \\
a_1 \bowtie^{\sim} a_2 \wedge a_1 \preceq^{\sim} a_2 & \Leftrightarrow & a_1 <:^{\sim} a_2 & & ✿\ ✿
\end{array}$$

Proof: By the use of early_{a_i} and late_{a_i}. □

As shown in Figure 3, words in envelopes a_1 and a_2 navigate between their two respective lines. If the lines have the same slope, all words stay at a bounded distance from each other. They are thus synchronizable. If moreover the overlapping between the envelopes is small enough, the words of the first envelope are always above the ones of the second one. Hence, the precedence relation is satisfied. In Figure 3, the envelope a_1 is adaptable to a_2.

Definition 9 (Buffer Size). ✿
Let $a_1 = \langle {b^0}_1, {b^1}_1 \rangle\,(r_1)$ *and* $a_2 = \langle {b^0}_2, {b^1}_2 \rangle\,(r_2)$ *two envelopes such that* $a_1 <:^{\sim} a_2$*.*

$$\mathit{size}^{\sim}(a_1, a_2) = \left\lfloor {b^1}_1 - {b^0}_2 \right\rfloor$$

The size of the buffer to communicate between an element of a_1 and an element of a_2 is the size necessary to communicate between the earlier element of a_1 and the latest element of a_2. That size can be over approximated by the distance between the upper line of a_1 and the lower line of a_2. The floor function is used since a buffer has an integral number of elements.

Proposition 5 (Correctness of $\mathit{size}^{\sim}$**).** ✿ *The following property holds:*

$$\forall w_1 \in \mathit{concr}\,(a_1), \forall w_1 \in \mathit{concr}\,(a_2), \mathit{size}(w_1, w_2) \leq \mathit{size}^{\sim}(a_1, a_2)$$

Proof: By the computation of $\mathit{size}(\mathit{early}_{a_1}, \mathit{late}_{a_2})$. □

5.2 Precision of Abstract Operators and Tests

We characterize here the precision of abstract operators and tests on envelopes that are in the normal form defined below.

Definition 10 (Normal Form)
Let $a = \langle b^0, b^1 \rangle\,(r)$*. Its normal form is* $\left\langle \frac{k^0}{\ell}, \frac{k^1}{\ell} \right\rangle \left(\frac{n}{\ell}\right)$ *with*

$$\frac{n}{\ell} = r \qquad \gcd(n, \ell) = 1 \qquad k^0 = \lceil b^0 \times \ell \rceil \qquad k^1 = \lfloor b^1 \times \ell \rfloor$$

Intuitively, putting an envelope in normal form consists in moving the lines as close as possible without changing the concretization set. To achieve this, if ℓ is the denominator of the reduced form of the rational slope, b^1 is decreased to the biggest rational number with denominator ℓ and b^0 is increased to the smallest rational number with denominator ℓ. Note that the movements are always strictly less than $\frac{1}{\ell}$.

If a is in normal form, the test given in proposition 1 to check whether a is empty or not is not only correct, but also complete. It means that it succeeds for all non-empty envelopes in normal form. The precedence test given in proposition 4 is also correct and complete on envelopes in normal form. Finally, as the synchronizability test is always correct and complete, the adaptability test is correct and complete on envelopes in normal form.

The most precise result for a_1 $\textit{\textbf{on}}^{\sim}$ a_2 is the smallest envelope that contains the following set of words:

$$W = \{w \mid \; w = w_1 \; \textit{\textbf{on}} \; w_2 \quad \wedge \quad w_1 \in \textit{concr}\,(a_1) \quad \wedge \quad w_2 \in \textit{concr}\,(a_2)\}$$

The formula that we proposed in definition 6 is not the most precise. Nonetheless, if a_1 and a_2 are not empty, and such that ${b^0}_1 \leq 0$, ${b^0}_2 \leq 0$, $r_1 \neq 0$ and a_1 is in normal form, we are able to quantify its imprecision: the line Δ^1 (resp. Δ^0) that we compute can be slightly above (resp. below) the most precise line, but at a distance of less than one.

Finally, the most precise correct buffer size computed on envelopes a_1 and a_2 is the minimal buffer size sufficient to communicate from every word of a_1 to every word of a_2. The formula given in definition 9 doesn't give the most precise result, but overestimates it by at most one.

5.3 Comparison with Previous Abstractions

There are several important differences between the abstraction presented here and the one introduced in [5]. Contrary to the previous abstraction, the new one is able to consider binary words with null rates (i.e., composed by a prefix followed by the infinite repetition of `0`). The new abstraction is more precise on binary words with a prefix starting by a bunch of `1`s. The formula for abstraction of the operator ***not*** is far simpler and also treats the case of words with rate 0 or 1. The precision of the abstraction has been established. Finally, the main properties have been formally proved in Coq.

6 Solving Subtyping Constraints

A subtyping constraint is introduced for every expression of the form `buffer` e. subtyping constraints are all gathered and solved for every node definition. For example, we have seen in Section 4 that the type of `good` is $\alpha \rightarrow \alpha$ `on (10)` with the constraints system $\{\alpha$ `on (10) <:` α `on (01)`$\}$ and that the constraint is verified if and only if `(10)` $<:$ `(01)`. To solve it, we can go into the abstract domain:

$$\texttt{(10)} <: \texttt{(01)} \quad \Leftarrow \quad \textit{abs}\,(\texttt{(10)}) <:^{\sim} \textit{abs}\,(\texttt{(01)})$$

The abstractions of `(10)` and `(01)` are respectively $\langle 0, \frac{1}{2}\rangle\left(\frac{1}{2}\right)$ and $\langle -\frac{1}{2}, 0\rangle\left(\frac{1}{2}\right)$. Thus, by application of proposition 4, we get:

$$\texttt{(10)} <: \texttt{(01)} \quad \Leftarrow \quad \left(\frac{1}{2} = \frac{1}{2} \quad \wedge \quad 0 - 0 < 1\right)$$

The subtyping constraint for the node `good` is always verified. Its type scheme is thus: $\forall\alpha.\alpha \rightarrow \alpha$ `on (10)`. On this example, the abstract method find the same solution as the method on ultimately periodic words.

This example was particularly simple because the constraint was on the very same variable α. This is not always the case as in:

```
let node f (x, y, z) =
  buffer (x when (11010))
    + y when 0(00111)
    + buffer (z when (01))
    + buffer (z when 0(1100))
```

The type of the node is $\alpha_1 \times \alpha_2 \times \alpha_3 \rightarrow \alpha_2$ `on 0(00111)` with the following constraints system C:

$$C = \left\{ \begin{array}{lll} \alpha_1 \texttt{ on (11010)} & <: & \alpha_2 \texttt{ on 0(00111)} \\ \alpha_3 \texttt{ on (01)} & <: & \alpha_2 \texttt{ on 0(00111)} \\ \alpha_3 \texttt{ on 0(1100)} & <: & \alpha_2 \texttt{ on 0(00111)} \end{array} \right\}$$

Depending on the instantiation of type variables α_1, α_2 and α_3, these constraints can be verified or not. Solving C consists in finding a substitution for those variables such that the constraints are always satisfied. For that, we have to express all related constraints according to the same type variable. Let $\alpha_1 = \alpha \texttt{ on } c_1$, $\alpha_2 = \alpha \texttt{ on } c_2$, $\alpha_3 = \alpha \texttt{ on } c_3$ where c_1, c_2 and c_3 fresh unknown variables such that the following system is satisfied:

$$C \Leftrightarrow \left\{ \begin{array}{lll} c_1 \textit{ on } \texttt{(11010)} & <: & c_2 \textit{ on } \texttt{0(00111)} \\ c_3 \textit{ on } \texttt{(01)} & <: & c_2 \textit{ on } \texttt{0(00111)} \\ c_3 \textit{ on } \texttt{0(1100)} & <: & c_2 \textit{ on } \texttt{0(00111)} \end{array} \right\}$$

We have translated the subtyping constraints into adaptability constraints. In order to solve them, we look for a solution in the abstract domain:

$$C \Leftarrow \left\{ \begin{array}{lll} \textit{abs}\,(c_1) \textit{ on}^{\sim} \textit{abs}\,(\texttt{(11010)}) & <:^{\sim} & \textit{abs}\,(c_2) \textit{ on}^{\sim} \textit{abs}\,(\texttt{0(00111)}) \\ \textit{abs}\,(c_3) \textit{ on}^{\sim} \textit{abs}\,(\texttt{(01)}) & <:^{\sim} & \textit{abs}\,(c_2) \textit{ on}^{\sim} \textit{abs}\,(\texttt{0(00111)}) \\ \textit{abs}\,(c_3) \textit{ on}^{\sim} \textit{abs}\,(\texttt{0(1100)}) & <:^{\sim} & \textit{abs}\,(c_2) \textit{ on}^{\sim} \textit{abs}\,(\texttt{0(00111)}) \end{array} \right\}$$

Consider the second constraint. Let $\textit{abs}\,(c_2) = \langle b^0{}_2, b^1{}_2\rangle\,(r_2)$ and $\textit{abs}\,(c_3) = \langle b^0{}_3, b^1{}_3\rangle\,(r_3)$. We compute values of the abstractions $\textit{abs}\,(\texttt{(01)}) = \left\langle -\frac{1}{2}, 0\right\rangle\left(\frac{1}{2}\right)$ and $\textit{abs}\,(\texttt{0(00111)}) = \left\langle -\frac{9}{5}, -\frac{3}{5}\right\rangle\left(\frac{3}{5}\right)$. Then, by definition of $\textit{on}^{\sim}$, the constraint can be rewritten into:[3]

$$\left\langle b^0{}_3 \times \tfrac{1}{2} - \tfrac{1}{2}, b^1{}_3 \times \tfrac{1}{2} + 0\right\rangle \left(r_3 \times \tfrac{1}{2}\right) <:^{\sim} \left\langle b^0{}_2 \times \tfrac{3}{5} - \tfrac{9}{5}, b^1{}_2 \times \tfrac{3}{5} - \tfrac{3}{5}\right\rangle \left(r_2 \times \tfrac{3}{5}\right)$$

Then, by proposition 4, it can be decomposed into a synchronizability constraint $(r_3 \times \frac{1}{2} = r_2 \times \frac{3}{5})$ and a precedence constraint $((b^1{}_2 \times \frac{3}{5} - \frac{3}{5}) - (b^0{}_3 \times \frac{1}{2} - \frac{1}{2}) < 1)$.

[3] To apply the definition, we have to suppose that $b^0{}_2 \leq 0$ and $b^0{}_3 \leq 0$.

If we apply these transformations to the other constraints, we can transform abstract adaptability constraints into a set of synchronizability constraints:

$$\left\{ \begin{array}{ccc} r_1 \times \frac{3}{5} & = & r_2 \times \frac{3}{5} \\ r_3 \times \frac{1}{2} & = & r_2 \times \frac{3}{5} \\ r_3 \times \frac{1}{2} & = & r_2 \times \frac{3}{5} \end{array} \right\}$$

and a set of precedence constraints:

$$\left\{ \begin{array}{ccc} ({b^1}_2 \times \frac{3}{5} - \frac{3}{5}) - ({b^0}_1 \times \frac{3}{5} + 0) & < & 1 \\ ({b^1}_2 \times \frac{3}{5} - \frac{3}{5}) - ({b^0}_3 \times \frac{1}{2} - \frac{1}{2}) & < & 1 \\ ({b^1}_2 \times \frac{3}{5} - \frac{3}{5}) - ({b^0}_3 \times \frac{1}{2} - \frac{1}{2}) & < & 1 \end{array} \right\}$$

In the synchronizability constraints, we look for correct rates. The r_i must be between 0 and 1 according to definition 4. To solve the system, we rewrite every constraint $r_i \times q_i = r_j \times q_j$ into $r_i = \frac{q_j}{q_i} \times r_j$ with $\frac{q_j}{q_i} \leq 1$. Then, we saturate the system by expressing all constraints in terms of a common variable. Finally, we choose the rate 1 for this variable so as to maximize the throughput of the system. In the example above, we obtain $r_1 = \frac{5}{6}$, $r_2 = \frac{5}{6}$ and $r_3 = 1$.

The precedence constraints allow to determine values of ${b^0}_i$ and ${b^1}_i$. In order to find non-empty envelopes, we add non-vacuity constraints as defined in proposition 1. Since we have computed the $\mathit{on}^{\sim}$ operator, we also have to add the constraints that ${b^0}_i \leq 0$. Thus, the system reduces to a set of linear inequations which can be solved using a standard tool such as Glpk [7]. We find the following solution to the abstract adaptability constraints:

$$\mathit{abs}\,(c_1) = \left\langle -\tfrac{5}{6}, 0 \right\rangle \left(\tfrac{5}{6}\right) \qquad \mathit{abs}\,(c_2) = \left\langle 0, \tfrac{10}{6} \right\rangle \left(\tfrac{5}{6}\right) \qquad \mathit{abs}\,(c_3) = \langle 0, 0 \rangle\,(1)$$

By definition of the relation $<:^{\sim}$, every word in the computed envelope is a solution of the original adaptability constraints system. Thus, we can take, for example $c_1 =$ `(011111)`, $c_2 =$ `(111110)` and $c_3 =$ `(1)` as a solution. Applying the substitution: $\{\alpha_1 \leftarrow \alpha \text{ on } c_1, \alpha_2 \leftarrow \alpha \text{ on } c_2, \alpha_3 \leftarrow \alpha \text{ on } c_3\}$ in the type of `f`, we get a correct type for any instantiation of α. The final type signature for `f` is:

$$\texttt{f} : \forall \alpha.\ \alpha \texttt{ on (011111)} \times \alpha \texttt{ on (111110)} \times \alpha \rightarrow \alpha \texttt{ on (111110) on 0(00111)}$$

Once the system of constraints is solved, we know input and output clocks of the buffers. Hence, we can compute their size. Concerning node `f`, buffers on line 2, 4 and 5 are respectively of size 2, 1 and 2.

7 Implementation

All the presented material has been implemented in OCaml. A distinctive feature of the implementation is to be generic: the clock calculus is a functor parametrized by the basic clock language (the one defining *ce*). It can accept any clock language provided the following functions are given:

Equality: A function `equal` to test the equality of two clocks and a function `unify` that takes as input two clock expressions ce_1 and ce_2 and returns two expressions ce'_1 and ce'_2 such that ce'_1 *on* $ce_1 = ce'_2$ *on* ce_2.

Adaptability: A function `adaptable` to test the adaptability of two clocks and a function `solve` that returns an instantiation of variables that satisfies an adaptability constraints system.

Buffer size: A function `buffer_size` to compute the size of a buffer provided its input and output clocks.

In the Lucy-n compiler, the clock expressions, unification algorithm and constraints solving algorithm can be chosen using respectively the options `-ce`, `-unif` and `-solver`. The examples given in the present paper have been typed using the following command line:

```
  lucync -ce pbw -unif semii -solver abs file.ls
```

It means that clock expressions are made of periodic binary words (`pbw`), that unification is semi-interpreted (`semii`) and that the solver of subtyping constraints uses abstraction (`abs`).

8 Application: The *Picture in Picture*

We illustrate the language on the example of a *Picture in Picture*. It is depicted in Figure 4 and is programmed in the following way:

```
let clock encrust_end =
  (0^(1920 * (1080 - 480)) {0^1200 1^720}^480)

let node pip_end (p1, p2) = o where
   rec small = buffer(downscaler p1)
   and big = (p2 whenot encrust_end)
   and o = merge encrust_end small big
```

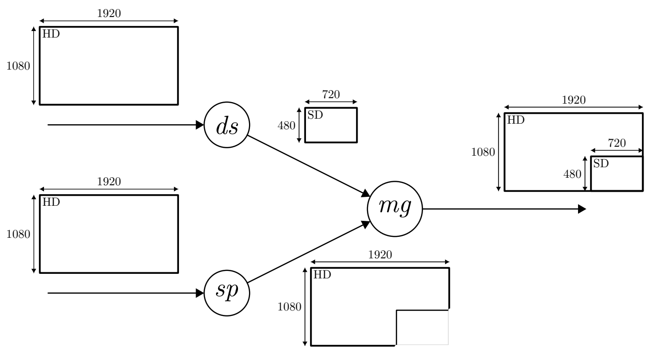


Fig. 4. *Picture in Picture.* The inputs are two flows of pixels representing High Definition videos. The size of the first video is reduced using a *Downscaler*. Some pixels from the second image are eliminated by sampling. Then the two images are merged.

The boolean sequence `encrust_end` controls the merge operation: when it is true, the small image is emitted, otherwise, the big one is emitted. The notation `{0^1200 1^720}^480` is a shortcut for repeating 480 times the pattern $0^{1200}1^{720}$. The small image is obtained by application of the node `downscaler`. It consists of an horizontal and a vertical filter. The horizontal filter applies a convolution and a sampling to reduce the size of lines from 1920 to 720. Similarly, the vertical filter applies a convolution and a sampling to reduce the size of columns from 1080 to 480. For the convolution, the vertical filter needs the pixels above and below every pixel. This means that the first output can be produced only after the consumption of one line of input (720 pixels). The signature inferred for node `downscaler` is:

```
val downscaler :: forall 'a. 'a -> 'a on hf on 0^720 (1) on vf
```

Since `downscaler` output is connected to the `merge` through a buffer, the type of node `pip_end` is $(\alpha_6 \times \alpha_{10}) \rightarrow \alpha_{10}$ with the following constraint (as emitted by the compiler):

```
'a6 on hf on 0^720 (1) on vf <: 'a10 on encrust_end
```

It remains to find envelopes $abs\,(c_6)$ and $abs\,(c_{10})$ which satisfy the following constraint:

$$abs\,(c_6)\ \mathit{on}^{\sim}\ \left\langle -720, \tfrac{481}{3} \right\rangle \left(\tfrac{1}{6}\right) <:^{\sim} abs\,(c_{10})\ \mathit{on}^{\sim}\ \langle -192200, 0\rangle \left(\tfrac{1}{6}\right)$$

The values computed for $abs\,(c_6)$ and $abs\,(c_{10})$ are the envelopes $\langle 0, 0\rangle\,(1)$ and $\langle -4315, -4315\rangle\,(1)$. As their concretization sets contain respectively the words `(1)` and 0^{4315}`(1)`, the inferred type is:

```
val pip_end ::
  forall 'a. ('a * 'a on 0^4315 (1)) -> 'a on 0^4315 (1)
Buffer line 56, characters 13-34: size = 192240
```

This means that if the input image to be reduced arrives at the very first instant, the big image should arrive after a delay of 4315 cycles and the resulting image is produced after this delay which is approximately two lines of high definition images.

We saw that the node `downscaler` introduces a delay while the `merge` doesn't. The inferred type is correct but overestimates by one line the necessary delay before the production starts. This is due to the abstraction. Nonetheless, we believe it to be reasonable considering that the existing resolution method with no abstraction needs one day of computation. Finally, the size of the buffer is also satisfactory: 192 240 instead of 191 970 for the value with no abstraction, that is, less than one extra line of the small image.

9 Conclusion

This paper has presented the implementation of the n-synchronous model inside a Lustre-like language. It is achieved by defining an extended clock calculus and introducing a subtyping rule. The implementation is modular in the sense that it can be instantiated for any clock language *ce* for which $ce_1 = ce_2$, $ce_1 <: ce_2$ and $size(ce_1, ce_2)$ are defined.

The language is conservative with respect to Lustre in the sense that if the program is synchronous and no `buffer` *e* construct is used, then the program is accepted by the clock calculus. Though the size of buffers is computed automatically, the places at which to insert them is still explicit.

Acknowledgments

We specially thank the reviewers for their careful reading and numerous comments which helped us improving the paper and Stéphane Lescuyer for his essential help in the Coq formalisation and proofs.

References

1. Buck, J.T.: Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model. Ph.D. thesis, EECS Department, University of California, Berkeley (1993)
2. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: Lustre: a declarative language for real-time programming. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 178–188. ACM, New York (1987)
3. Caspi, P., Pouzet, M.: Synchronous kahn networks. In: ACM SIGPLAN International Conference on Functional Programming, pp. 226–238. ACM Press, New York (1996)
4. Cohen, A., Duranton, M., Eisenbeis, C., Pagetti, C., Plateau, F., Pouzet, M.: *N*-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems. In: ACM International Conference on Principles of Programming Languages (POPL'06), pp. 180–193. ACM Press, New York (2006)
5. Cohen, A., Mandel, L., Plateau, F., Pouzet, M.: Abstraction of clocks in synchronous data-flow systems. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 237–254. Springer, Heidelberg (2008)
6. Colaço, J.L., Pouzet, M.: Clocks as First Class Abstract Types. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 134–155. Springer, Heidelberg (2003)
7. Glpk: Gnu linear programming kit
8. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information processing, pp. 471–475. North Holland, Amsterdam (August 1974)
9. Lee, E., Messerschmitt, D.: Synchronous dataflow. IEEE Trans. Comput. 75(9), 1235–1245 (1987)
10. Parks, T.M., Pino, J.L., Lee, E.A.: A comparison of synchronous and cycle-static dataflow. In: ASILOMAR '95: Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers, vol. 2, p. 204. IEEE Computer Society, Washington (1995)
11. Parks, T.M.: Bounded scheduling of process networks. Ph.D. thesis, EECS Department, University of California, Berkeley, Berkeley, CA, USA (1995)

12. Plateau, F.: Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée. Ph.D. thesis, Université Paris-Sud 11 (January 2010)
13. Pouzet, M.: Lucid Synchrone, version 3. Tutorial and reference manual. Université Paris-Sud, LRI (April 2006)
14. Thies, W., Karczmarek, M., Amarasinghe, S.P.: Streamit: A language for streaming applications. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)

A *Downscaler* Code

```
(* convolutions *)
let node convo (c0, c1, c2) = (c0 + c1 + c2) / 3

let node convolution (p0, p1, p2) = p where
  rec p = (r,g,b)
  and r = convo (p0r, p1r, p2r)
  and g = convo (p0g, p1g, p2g)
  and b = convo (p0b, p1b, p2b)
  and p0r, p0g, p0b = p0
  and p1r, p1g, p1b = p1
  and p2r, p2g, p2b = p2

(* horizontal filter *)
let clock hf = (10100100)

let node horizontal_filter p = o where
  rec p0 = p fby p1
  and p1 = p fby p2
  and p2 = p
  and o = (convolution (p0, p1, p2)) when hf

(* vertical sliding_window *)
let clock first_sd_line = 1^720 (0)
let clock first_line_of_img = (1^720 0^(720*1079))
let clock last_line_of_img = (0^(720*1079) 1^720)

let node my_fby_sd_line (p1,p2) =
  merge first_sd_line (p1 when first_sd_line) (buffer(p2))

let node reorder p = ((p0,p1,p2)::'a) where
  rec p0 =
    merge first_line_of_img
      (p1 when first_line_of_img)
      ((my_fby_sd_line (p1, p1)) whenot first_line_of_img)
  and p1 = buffer(p)
  and p2 =
    merge last_line_of_img
      (p1 when last_line_of_img)
      ((p whenot first_sd_line) whenot last_line_of_img)
```

```

(* vertical filter *)
let clock vf = (1^720 0^720 1^720 0^720 0^720 1^720 0^720 0^720 1^720)

let node vertical_filter p = o where
  rec (p0,p1,p2) = reorder p
  and o = (convolution (p0, p1, p2)) when vf

(* downscaler *)
let node downscaler p = vertical_filter (horizontal_filter p)
```

Sampling, Splitting and Merging in Coinductive Stream Calculus

Milad Niqui[1,⋆] and Jan Rutten[1,2]

[1] Centrum Wiskunde & Informatica (CWI), The Netherlands
{M.Niqui,janr}@cwi.nl
[2] Radboud University Nijmegen, The Netherlands

Abstract. We study various operations for partitioning, projecting and merging streams of data. These operations are motivated by their use in dataflow programming and the stream processing languages. We use the framework of *stream calculus* and *stream circuits* for defining and proving properties of such operations using behavioural differential equations and coinduction proof principles. We study the invariance of certain well patterned classes of streams, namely rational and algebraic streams, under splitting and merging. Finally we show that stream circuits extended with gates for dyadic split and merge are expressive enough to realise some non-rational algebraic streams, thereby going beyond ordinary stream circuits.

Keywords: stream calculus, dataflow programming, coinduction, rational stream, algebraic stream, stream circuit.

1 Introduction

In this paper, we study various operations for splitting, partitioning, projecting and merging streams (infinite sequences of data). These operations are motivated by their use in dataflow programming and stream processing languages (e.g., [BŞ01]).

Our perspective on streams and stream operations will be essentially coalgebraic. More specifically, we use the framework of *stream calculus* [Rut05a] and *stream circuits* [Rut05b] for defining and proving properties of such operations. Definitions are typically given using behavioural stream differential equations. Proofs will mostly be given by coinduction, with which two streams can be shown to be equal by the construction of a suitable stream bisimulation relation between them.

The use of stream calculus and coinduction leads to new and simpler definitions and proofs of several existing notions and properties, some of which are taken from [Mak08]. To mention already one example here (see Sections 3 and 4 for more): a periodic stream sampler S is a stream operation that produces a

⋆ Supported by a VENI grant from the Netherlands Organisation for Scientific Research (NWO).

substream of a given stream σ by taking out of each block of $l \geq 0$ elements a subset of $k \leq l$ elements (at fixed positions). Periodic stream samplers can be defined by the following stream differential equation:

$$S(\sigma)^{(k)} = S(\sigma^{(l)})$$

(plus the specification of k initial values). Here $(-)^{(i)}$ denotes the i-th stream derivative, which is defined as the operation *tail* applied i times. This differential equation is elementary, almost trivial. Yet it allows for proofs of basic facts (such as: composing two periodic steam samplers yields again a periodic stream sampler) that are much simpler than those in the literature.

Using stream calculus and stream circuits, we obtain also a number of new results. More specifically, we prove (in Sections 5 and 6) the invariance of certain well patterned classes of streams, namely rational and algebraic streams, under the operations of splitting and merging. Furthermore, we show (in Section 7) that stream circuits extended with gates for dyadic split and merge are expressive enough to realise some non-rational algebraic streams (such as the Prouhet–Thue–Morse stream), thereby going beyond ordinary stream circuits.

As mentioned above, this paper attempts to give a new perspective on existing notions and results, and also obtains some modest new results. The presented new outlook gives rise to a host of further questions and research directions. Section 8 discusses related work and future research.

2 Preliminaries

We define the set of *streams* over a set A by $A^\omega = \{\sigma \mid \sigma : \mathbb{N} \to A\}$. We denote elements $\sigma \in A^\omega$ by $\sigma = (\sigma(0), \sigma(1), \sigma(2), \ldots)$. The *stream derivative* of a stream σ is $\sigma' = (\sigma(1), \sigma(2), \sigma(3), \ldots)$ and the *initial value* of σ is $\sigma(0)$. For $n \geq 0$ and $\sigma \in A^\omega$, we define higher-order derivatives by $\sigma^{(0)} = \sigma$ and $\sigma^{(n+1)} = (\sigma^{(n)})'$. We have $\sigma(n) = \sigma^{(n)}(0)$.

A *stream bisimulation relation* is a set $R \subseteq A^\omega \times A^\omega$ such that, for all $(\sigma, \tau) \in R$,

$$\sigma(0) = \tau(0) \quad \text{and} \quad (\sigma', \tau') \in R \ .$$

We write $\sigma \sim \tau$ if there exists a bisimulation R with $(\sigma, \tau) \in R$. The *coinduction proof principle* allows us to prove the equality of two streams by establishing the existence of an appropriate bisimulation relation:

$$\sigma \sim \tau \ \Rightarrow \ \sigma = \tau \ .$$

If A has some algebraic structure, A^ω inherits (parts of) this structure. Assume $\langle A, +, \cdot, -, 0, 1 \rangle$ is a ring[1]. For $r \in A$, we define the constant stream $[r] = (r, 0, 0, 0, \ldots)$, which we often denote again by r. Another constant stream

[1] In fact many of the operations on A^ω only need a semiring structure on A [BR88, Rut08].

is $X = (0, 1, 0, 0, 0, \ldots)$. For $\sigma, \tau \in A^\omega$ and $n \geq 0$, the operations of *sum* and (convolution) *product* are given by

$$(\sigma + \tau)(n) = \sigma(n) + \tau(n) \ , \qquad (\sigma \times \tau)(n) = \sum_{i=0}^{n} \sigma(i) \cdot \tau(n-i)$$

(where $\cdot$ denotes ring multiplication).

We call a stream $\pi \in A^\omega$ *polynomial* if there are $k \geq 0$ and $a_i \in A_0$ such that

$$\pi = a_0 + a_1 X + a_2 X^2 + \cdots + a_k X^k \ = \ (a_0, a_1, a_2, \ldots, a_k, 0, 0, 0, \ldots)$$

where we write $a_i X^i$ for $[a_i] \times X^i$ with X^i the i-fold product of X with itself.

One can compute a stream from its initial value and derivative by the so-called *fundamental theorem* of stream calculus [Rut05a]: for all $\sigma \in A^\omega$,

$$\sigma = \sigma(0) + (X \times \sigma')$$

(writing $\sigma(0)$ for $[\sigma(0)]$).

Next assume A is a field, i.e., every nonzero element has a unique multiplicative inverse. Then this multiplicative inverse operation may be carried over to A^ω: if $\sigma(0) \neq 0$ then the stream σ has a (unique) multiplicative inverse σ^{-1} in A^ω, satisfying $\sigma^{-1} \times \sigma = [1]$. As usual, we shall often write $1/\sigma$ for σ^{-1} and σ/τ for $\sigma \times \tau^{-1}$. Note that the initial value of the sum, product and inverse of streams is given by the sum, product and inverse of their initial values.

If A is a field, a stream $\rho \in A^\omega$ is *rational* if it is the quotient $\rho = \sigma/\tau$ of two polynomial streams σ and τ with $\tau(0) \neq 0$.

The fundamental theorem of stream calculus allows us to solve *stream differential equations* such as $\sigma' = 2 \times \sigma$ with initial value $\sigma(0) = 1$ by computing $\sigma = \sigma(0) + (X \times \sigma') = 1 + (X \times 2 \times \sigma)$, which leads to the solution $\sigma = 1/(1 - 2X)$. Together with the basic fact that $(X \times \sigma)' = \sigma$, the fundamental theorem also leads to an easy calculation rule for the computation of derivatives: $\sigma' = (\sigma - \sigma(0))'$. This identity makes the computation of stream derivatives often surprisingly simple. For instance, for $\sigma = 1/(1 - X)^2$, we have

$$\sigma' = \left(\frac{1}{(1-X)^2} - 1\right)' = \left(\frac{2X - X^2}{(1-X)^2}\right)' = \left(X \times \frac{2 - X}{(1-X)^2}\right)' = \frac{2 - X}{(1-X)^2} \ .$$

For more stream calculations we refer the reader to [Rut05a].

In the remainder of the article we assume A is a field. Strictly speaking, this is not always necessary as some of the constructs, e.g. the stream samplers, do not presume any algebraic structure on A. Nevertheless, in order to be able to freely use the stream calculus we make this assumption. In Section 6 we work in the special case where $A := \mathbb{F}_q$ is a finite field.

3 Periodic Stream Samplers

Traditionally, a substream of an infinite stream $\sigma : \mathbb{N} \to A$ is defined by means of a (strictly) monotone function $f : \mathbb{N} \to \mathbb{N}$: if $n < m$ then $f(n) < f(m)$. Such an *index function* determines an (infinite) substream $S_f(\sigma)$ by

$$S_f(\sigma)(n) = \sigma(f(n))$$

and conversely, any substream of σ determines a unique such monotone function. Assigning to any stream the substream determined by a given monotone function f defines a *stream sampler*

$$S_f : A^\omega \to A^\omega \ , \qquad \sigma \mapsto S_f(\sigma) \ .$$

Periodic stream samplers are such that they produce a substream of a given input stream by repeatedly choosing certain elements and ignoring all others. For instance, the function $\mathit{even} : A^\omega \to A^\omega$ given by

$$\mathit{even}(\sigma) = (\sigma(0), \sigma(2), \sigma(4), \ldots)$$

takes of each incoming two elements the first and ignores the second. We say that *even* has *(input) period* 2 and *(output) block size* 1. Another example is the drop operator $D_4^2 : A^\omega \to A^\omega$ given by

$$D_4^2(\sigma) = (\sigma(0), \sigma(1), \sigma(3), \sigma(4), \sigma(5), \sigma(7), \ldots)$$

which drops from each four incoming elements the third and keeps all the others. Note we always start counting at zero hence $\sigma(2)$, $\sigma(6)$ etc. are dropped. The operator D_4^2 has period 4 and block size 3.

As it turns out, it is somewhat cumbersome to define these and similar such periodic stream samplers by means of monotone index functions. Moreover, it is surprisingly difficult to prove simple general facts such as: The composition of two periodic stream samplers is again a period stream sampler. Therefore, we prefer the following coinductive definition which uses a stream differential equation.

Definition 1. *Let $k, l \in \mathbb{N}$ with $l > 1$ and $1 \le k \le l$. Any sequence of k numbers $0 \le n_0 < n_1 < \cdots < n_{k-1} < l$ determines a* periodic stream sampler $S : A^\omega \to A^\omega$ *of (input) period l and (output) block size k defined by the following stream differential equation:*

$$S(\sigma)^{(k)} = S(\sigma^{(l)})$$

with initial values

$$S(\sigma)(j) = \sigma(n_j) \quad (0 \le j < k) \ .$$

We do not require period and block size to be minimal. If a stream sampler has period l and block size k then it also has period $2l$ with block size $2k$, etc.

The functions *even* and D_4^2 above are given by

$$\mathit{even}(\sigma)' = \mathit{even}(\sigma'') \ , \qquad \mathit{even}(\sigma)(0) = \sigma(0) \ ,$$

$$D_4^2(\sigma)^{(3)} = D_4^2(\sigma^{(4)}) \ , \quad D_4^2(\sigma)(0) = \sigma(0) \ , \quad D_4^2(\sigma)(1) = \sigma(1) \ , \quad D_4^2(\sigma)(2) = \sigma(3) \ .$$

Proposition 2. *If $S, T : A^\omega \to A^\omega$ are two periodic stream samplers then so is $T \circ S$.*

Proof. Let S and T satisfy

$$S(\sigma)^{(k)} = S(\sigma^{(l)}) \ , \qquad S(\sigma)(j) = \sigma(n_j) \quad (0 \leq j < k) \ ,$$

$$T(\sigma)^{(p)} = T(\sigma^{(q)}) \ , \qquad T(\sigma)(j) = \sigma(m_j) \quad (0 \leq j < p) \ .$$

We claim that $T \circ S$ is a periodic stream sampler with period $l \times q$ and block size $k \times p$. We define a sequence $i_0, i_1, \ldots, i_{q\times k-1}$ by

$$i_{(x\times k)+y} = \ (x \times l) + n_y \quad (\text{all} \ \ x, y \ \ \text{with} \ \ 0 \leq x < q, \ \ 0 \leq y < k) \ .$$

Next we define a sequence $0 \leq h_0 < h_1 < \cdots < h_{(k\times p)-1} < q \times k$ by

$$h_{(x\times p)+y} = \ (x \times q) + m_y \quad (\text{all} \ \ x, y \ \ \text{with} \ \ 0 \leq x < k, \ \ 0 \leq y < p) \ .$$

One readily shows that $T \circ S$ satisfies

$$T \circ S(\sigma)^{(k\times p)} = T \circ S(\sigma^{(l\times q)}) \ , \qquad T \circ S(\sigma)(j) = \sigma(i_{h_j}) \quad (0 \leq j < (k \times p) - 1) \ .$$

□

Next we provide some examples by introducing the family of all drop operators.

Definition 3. *For $l \geq 2$ and $0 \leq i < l$ we define the drop operator*

$$D_l^i : A^\omega \to A^\omega$$

which drops from each input block of size l the i-th element, by the following system of stream differential equations:

$$D_l^{i+1}(\sigma)' = D_l^i(\sigma') \ , \qquad D_l^{i+1}(\sigma)(0) = \sigma(0) \quad (\textit{all} \ \ l \geq 2, \ \ 0 \leq i < l-1) \ ,$$

$$D_l^0(\sigma)' = D_l^{l-2}(\sigma'') \ , \qquad D_l^0(\sigma)(0) = \sigma(1) \quad (\textit{all} \ \ l \geq 2) \ .$$

Note that for D_4^2, this definition is equivalent with our earlier definition above; also note that $\mathit{even} = D_2^1$.

One of the benefits of coinductive definitions is that they support coinductive proofs. As an example, we prove the so-called *Drop exchange rule* from [Mak08]: for all $l \geq 1$, $0 \leq k \leq h \leq l$,

$$D_{l+1}^h \circ D_{l+2}^k = \ D_{l+1}^k \circ D_{l+2}^{h+1} \ .$$

In order to prove this equality, we define a relation $R \subseteq A^\omega \times A^\omega$ by

$$R = \{ \langle D_{l+1}^h \circ D_{l+2}^k(\sigma), \ D_{l+1}^k \circ D_{l+2}^{h+1}(\sigma) \rangle \mid \ \sigma \in A^\omega \ \} \ .$$

The equality now follows by coinduction from the fact that $R \cup R^{-1}$ is a stream bisimulation.

Here is another example. It is a basic instance of a *Drop expansion rule* in [Mak08]:

$$D_2^0 = D_4^0 \circ D_5^2 \circ D_6^4 \ .$$

For a proof, we define a relation $R \subseteq A^\omega \times A^\omega$ by

$$\begin{aligned} R = &\{ \langle D_2^0(\sigma),\ D_4^0 \circ D_5^2 \circ D_6^4(\sigma) \rangle \mid \sigma \in A^\omega \} \\ \cup &\{ \langle D_2^0(\sigma),\ D_4^2 \circ D_5^0 \circ D_6^2(\sigma) \rangle \mid \sigma \in A^\omega \} \\ \cup &\{ \langle D_2^0(\sigma),\ D_4^1 \circ D_5^3 \circ D_6^0(\sigma) \rangle \mid \sigma \in A^\omega \} \ . \end{aligned}$$

The equality follows by coinduction from the fact that R is a stream bisimulation.

Returning to the general question of how to define substreams out of a given stream, we present yet another alternative to the use of monotone index functions, which is also well suited for a coinductive approach. Let $2 = \{0, 1\}$ and let 2^ω be the set of bitstreams. Note that there is a trivial field structure on 2 and hence we can apply stream calculus to 2^ω. Consider a bitstream $\alpha \in 2^\omega$ that is not eventually constant 0, i.e., there is no n such that $\alpha^{(n)} = [0]$. Then for any stream $\sigma \in A^\omega$, α defines a substream $S_\alpha(\sigma)$ consisting of those elements $\sigma(n)$ for which $\alpha(n) = 1$. (Note that the condition on α ensures that $S_\alpha(\sigma)$ is again an infinite stream.) Such a stream α acts as an oracle that tells us of any element of σ whether or not it should be included in the substream we are defining.

More formally, we first note that a stream $\alpha \in 2^\omega$ is eventually constant 0 if it is a polynomial. If α is non-polynomial, it is of the form

$$\alpha = X^n \times (1 + X \times \beta)$$

for some $n \geq 0$ and some $\beta \in 2^\omega$ that is again non-polynomial. Now we define $S_\alpha(\sigma)$ by the following system of differential equations, for arbitrary $\sigma \in A^\omega$ and non-polynomials $\alpha \in 2^\omega$:

$$S_\alpha(\sigma)' = S_\beta(\sigma^{(n+1)}) \ , \quad S_\alpha(\sigma)(0) = \sigma(n) \qquad (\alpha = X^n \times (1 + X \times \beta)) \ .$$

In this manner, any non-polynomial bitstream determines a substream and, conversely, any substream determines a non-polynomial bitstream.

It is now extremely simple to characterise *periodic* stream samplers:

$$S_\alpha \text{ is periodic with period } l \textit{ iff } \quad \alpha^{(l)} = \alpha \ .$$

The (output) block size is determined by the number of 1's in the set $\{\alpha(0), \ldots, \alpha(l-1)\}$.

Composition of stream samplers can be described in terms of composition of the corresponding oracle bitstreams, which we define as follows.

Definition 4. *For all $\alpha, \beta \in 2^\omega$, we define $\beta * \alpha \in 2^\omega$ by the following system of differential equations:*

$$(\beta * \alpha)' = \begin{cases} \beta' * \alpha' & \textit{if } \alpha(0) = 1 \\ \beta * \alpha' & \textit{if } \alpha(0) = 0 \end{cases} \qquad (\beta * \alpha)(0) = \beta(0) \cdot \alpha(0)$$

This composition operator is associative but not commutative and has $1/(1-X)$ as a neutral element: $\sigma * 1/(1-X) = 1/(1-X) * \sigma = \sigma$. It is not difficult to show that

$$S_\beta \circ S_\alpha = S_{\beta * \alpha} \ .$$

An alternative proof of Proposition 2 is now extremely easy: it follows from the fact that $\alpha^{(n)} = \alpha$ and $\beta^{(m)} = \beta$ imply $(\beta * \alpha)^{(n \times m)} = \beta * \alpha$.

Let us conclude this section with an example illustrating how one can reason about stream sampler composition in terms of stream calculus applied to the corresponding oracle streams. Periodic oracle bitstreams are always of the form

$$\frac{a_0 + a_1 X + a_2 X^2 + \cdots + a_{l-1} X^{l-1}}{1 - X^l}$$

for $a_0, a_1, a_2, \ldots, a_{l-1} \in 2$, not all 0. For our drop operators, for instance, one has

$$D_l^i = S_{\alpha_l^i} \quad \text{with} \quad \alpha_l^i = (1 + X + \cdots + X^{i-1} + X^{i+1} + \cdots + X^{l-1})/(1 - X^l)$$

The equality $D_2^0 = D_4^0 \circ D_5^2 \circ D_6^4$, which we proved above by coinduction, can also be deduced from the following computation in stream calculus on the corresponding oracle bitstreams:

$$\begin{aligned}
\alpha_4^0 * \alpha_5^2 * \alpha_6^4 &= \frac{X + X^2 + X^3}{1 - X^4} * \frac{1 + X + X^3 + X^4}{1 - X^5} * \frac{1 + X + X^2 + X^3 + X^5}{1 - X^6} \\
&= \frac{X + X^3 + X^4}{1 - X^5} * \frac{1 + X + X^2 + X^3 + X^5}{1 - X^6} \\
&= \frac{X}{1 - X^2} = \alpha_2^0 \ .
\end{aligned}$$

The work goes in the computation of the stream compositions, using the differential equation of Definition 4. This may be bothersome by hand but can easily be automated.

4 Splitting and Merging

All periodic stream samplers and, more generally, many periodic stream transformers that not necessarily preserve the order of the elements in a stream, can be obtained by splitting and merging streams. In this section, we introduce the operators of take and zip, with which streams can be split and merged, and we present a few basic laws about them.

Definition 5. *i) For $l \geq 2$ and $0 \leq i < l$, the* take operator $T_l^i : A^\omega \to A^\omega$ *is defined by the following stream differential equation:*

$$T_l^i(\sigma)' = T_l^i(\sigma^{(l)}) \ , \qquad T_l^i(\sigma)(0) = \sigma(i) \ .$$

ii) For $k \geq 1$ and streams $\sigma_0, \ldots \sigma_{k-1} \in A^\omega$, the zip operator $Z_k : (A^\omega)^k \to A^\omega$ *is defined by the stream differential equation*

$$Z_k(\sigma_0, \ldots, \sigma_{k-1})' = Z_k(\sigma_1, \ldots, \sigma_{k-1}, \sigma_0') \ , \quad Z_k(\sigma_0, \ldots, \sigma_{k-1})(0) = \sigma_0(0) \ .$$

(Note that $\sigma_0, \ldots, \sigma_{k-1}$ above are *streams*, not *elements* of streams, which for a stream σ we denote by $\sigma(0)$, $\sigma(1)$, etc.) Examples are

$$T_3^2(\sigma) = (\sigma(2), \sigma(5), \sigma(8), \ldots) \ ,$$

$$Z_2(\sigma, \tau) = (\sigma(0), \tau(0), \sigma(1), \tau(1), \sigma(2), \tau(2), \ldots) \ .$$

As suggested by the latter, it is easy to see (by induction) that in general if $0 \leq r \leq k{-}1$ then

$$Z_k(\sigma_0, \ldots, \sigma_{k-1})(kn + r) = \sigma_r(n) \ . \tag{4.1}$$

Any periodic stream sampler can be expressed in terms of take and zip. With S as in Definition 1, we have

$$S(\sigma) = Z_k(T_l^{n_0}(\sigma),\, T_l^{n_1}(\sigma),\, \ldots,\, T_l^{n_{k-1}}(\sigma)) \ .$$

More generally, we can define with take and zip periodic stream transformers that not merely produce substreams but that can change also the order of the elements. For instance, we can define the operation $Rev_k : A^\omega \to A^\omega$ of stream *reverse*, for any $k \geq 1$, by

$$Rev_k(\sigma) = \ Z_k(T_k^{k-1}(\sigma),\, T_k^{k-2}(\sigma),\, \ldots,\, T_k^0(\sigma)) \ .$$

For instance,

$$Rev_3(\sigma) = (\sigma(2), \sigma(1), \sigma(0), \sigma(5), \sigma(4), \sigma(3), \ldots) \ .$$

Next we present a few basic laws for take and zip that will allow us to prove elementary properties on stream transformers by equational reasoning. All of the identities below can easily be proved by coinduction.

Proposition 6. *For all $k \geq 1$, $l \geq 2$, $0 \leq i < l$,*

$$Z_k(T_k^0(\sigma),\, \ldots,\, T_k^{k-1}(\sigma)) = \ \sigma \ ,$$

$$T_l^i(Z_l(\sigma_0, \ldots, \sigma_{l-1})) = \ \sigma_i \ ,$$

$$T_l^i(\sigma) = \ Z_k(T_{k\times l}^i(\sigma),\, T_{k\times l}^{l+i}(\sigma),\, \ldots,\, T_{k\times l}^{(k-1)\times l+i}(\sigma)) \ .$$

Let us illustrate these identities with an equational proof of our earlier example, the Drop expansion rule: for all $\sigma \in A^\omega$,

$$D_2^0(\sigma) = D_4^0 \circ D_5^2 \circ D_6^4(\sigma) \ .$$

Let $\tau = D_6^4(\sigma)$. We have

$$\tau = \ D_6^4(\sigma) = \ Z_5(\, T_6^0(\sigma),\, T_6^1(\sigma),\, T_6^2(\sigma),\, T_6^3(\sigma),\, T_6^5(\sigma)) \ .$$

Next let $\rho = D_5^2 \circ D_6^4(\sigma)$; it satisfies

$$\begin{aligned} \rho = D_5^2(\tau) &= \ Z_4(\, T_5^0(\tau),\, T_5^1(\tau),\, T_5^3(\tau),\, T_5^4(\tau)) \\ &= Z_4(\, T_6^0(\sigma),\, T_6^1(\sigma),\, T_6^3(\sigma),\, T_6^5(\sigma)) \ . \end{aligned}$$

Finally, we compute

$$\begin{aligned} D_4^0 \circ D_5^2 \circ D_6^4(\sigma) = D_4^0(\rho) &= Z_3(\, T_4^1(\rho),\, T_4^2(\rho),\, T_4^3(\rho)\,) \\ &= Z_3(\, T_6^1(\sigma),\, T_6^3(\sigma),\, T_6^5(\sigma)\,) \\ &= T_2^1(\sigma) = D_2^0(\sigma)\ . \end{aligned}$$

As a second example, we prove $Rev_3 \circ Rev_3(\sigma) = \sigma$. Putting $\tau = Rev_3(\sigma)$,

$$\tau = Rev_3(\sigma) = Z_3(\, T_3^2(\sigma),\, T_3^1(\sigma),\, T_3^0(\sigma)\,)\ .$$

It follows that

$$\begin{aligned} Rev_3 \circ Rev_3(\sigma) = Rev_3(\tau) &= Z_3(\, T_3^2(\tau),\, T_3^1(\tau),\, T_3^0(\tau)\,) \\ &= Z_3(\, T_3^0(\sigma),\, T_3^1(\sigma),\, T_3^2(\sigma)\,) = \sigma\ . \end{aligned}$$

In the above, we have illustrated that the operators of take and zip are interesting because they can express all periodic stream samplers and because they can moreover be used to define stream transformers that have a periodic behaviour but that are not stream samplers. We have not given a general definition of periodic stream transformer. We shall come back to this point later.

5 Preserving Rationality

In this section, we show that the result of applying the operators of take and zip to rational streams in A^ω is again rational. We shall use the following definition from [Rut05a, p.109].

Definition 7. *For $\sigma \in A^\omega$ and $\rho \in A^\omega$ with $\rho(0) = 0$, we define the stream σ* applied to ρ, *written as $\sigma(\rho)$, by the following system of differential equations:*

$$\sigma(\rho)' = \sigma'(\rho) \times \rho'\ , \qquad \sigma(\rho)(0) = \sigma(0)\ .$$

Recall from [Rut05a] that every stream $\sigma \in A^\omega$ can be written as an infinite sum

$$\sigma = \sigma(0) + (\sigma(1) \times X) + (\sigma(2) \times X^2) + \cdots\ .$$

We may now think of $\sigma(\rho)$ as the stream that results from the above infinite sum by replacing every X by ρ (the condition $\rho(0) = 0$ will ensure that the resulting infinite sum is well-defined). In fact, there is the following identity:

$$\sigma(\rho) = \sigma(0) + (\sigma(1) \times \rho) + (\sigma(2) \times \rho^2) + \cdots\ .$$

This reminds one of formal power series and (generating) function application (cf. [GKP94]); note that the definition and identities above all live in stream calculus, where X is a constant stream and not a function variable.

If σ is polynomial and ρ is rational (with $\rho(0) = 0$) then $\sigma(\rho)$ is rational. Since for polynomials π and τ with $\tau(0) \neq 0$, one can easily show that

$$\frac{\pi}{\tau}(\rho) = \frac{\pi(\rho)}{\tau(\rho)}\ ,$$

it follows that if σ and ρ are rational then so is $\sigma(\rho)$. We shall be using the above mostly for the case that $\rho = X^n$, for some $n \geq 1$. For instance, we have

$$\frac{X}{(1-X)^2}(X^3) = \frac{X^3}{(1-X^3)^2} \ .$$

Since $X/(1-X)^2 = (0, 1, 2, \ldots)$ it follows that

$$\frac{X^3}{(1-X^3)^2} = (0, 0, 0, 1, 0, 0, 2, 0, 0, \ldots) \ .$$

We are now ready to formulate our first preservation result. We remark that Propositions 8 and 10 below can be found in [BR88]. Our proofs are different: in Proposition 8 the novelty lies in our use of coinduction proof principle; regarding Proposition 10 we give a rather elementary proof while the proof in [BR88] is based on Kleene–Schützenberger theorem.

Proposition 8. *The function zip preserves rationality: if $\sigma_0, \ldots, \sigma_{k-1} \in A^\omega$ are rational, for $k \geq 1$, then so is $Z_k(\sigma_0, \ldots, \sigma_{k-1})$.*

Proof. The proposition follows from the identity

$$Z_k(\sigma_0, \ldots, \sigma_{k-1}) = \sigma_0(X^k) + (X \times \sigma_1(X^k)) + \cdots + (X^{k-1} \times \sigma_{k-1}(X^k))$$

which can easily be proved by coinduction. □

Next we show that the take operators preserve rationality as well. We shall use the following lemma; it has an easy proof by coinduction which we omit here.

Lemma 9. *Let $l \geq 2$ and $0 \leq i < l$.*

(a) T_l^i is linear: for all $r, s \in A$, $\sigma, \tau \in A^\omega$,

$$T_l^i(\,(s \times \sigma) + (t \times \tau)\,) = (s \times T_l^i(\sigma)\,) + (t \times T_l^i(\tau)\,) \ .$$

(b) For $1 \leq i \leq l$ and $\sigma \in A^\omega$,

$$T_l^i(X \times \sigma) = T_l^{i-1}(\sigma) \ , \qquad T_l^0(X \times \sigma) = X \times T_l^{l-1}(\sigma) \ .$$

Proposition 10. *The function take preserves rationality: if $\sigma \in A^\omega$ is rational then so is $T_l^i(\sigma)$, for all $l \geq 2$ and $0 \leq i < l$.*

Proof. By Lemma 9, it is sufficient to prove the proposition for streams of the form $1/\sigma$, with σ polynomial and $\sigma(0) \neq 0$. So let $\sigma = s_0 + s_1 X + \cdots + s_d X^d$ be a polynomial stream, for $d \geq 0$ and $s_0, s_1, \ldots, s_d \in A$ with $s_0 \neq 0$. One can prove by induction that for any $l \geq 0$, the l-th stream derivative of $1/\sigma$ is of the form

$$(1/\sigma)^{(l)} = (r_0 + r_1 X + \cdots + r_{d-1} X^{d-1}\,) \times 1/\sigma$$

for certain $r_0, \ldots, r_{d-1} \in A$. Now for $l \geq 2$ and $0 \leq i < l$, we have

$$\begin{aligned} T_l^i(1/\sigma)' &= T_l^i(\,(1/\sigma)^{(l)}\,) \quad \text{[by definition]} \\ &= T_l^i(\,(r_0 + \cdots + r_{d-1}X^{d-1}\,) \times 1/\sigma\,) \quad \text{[by the equality above]} \\ &= (\rho_0 \times T_l^0(1/\sigma)\,) \,+\, \cdots \,+\, (\rho_{l-1} \times T_l^{l-1}(1/\sigma)\,) \end{aligned}$$

for certain rational streams $\rho_0, \ldots, \rho_{l-1} \in A^\omega$, where the last equality follows from Lemma 9. Multiplying the equation by X and adding $(1/\sigma)(i)$ to both sides gives

$$\begin{aligned} &T_l^i(1/\sigma) \\ &= T_l^i(1/\sigma)(0) + \,(X \times T_l^i(1/\sigma)'\,) \quad \text{[by the fundamental theorem, Section 2]} \\ &= (1/\sigma)(i) + \big(X \times \big((\rho_0 \times T_l^0(1/\sigma)\,) \,+\, \cdots \,+\, (\rho_{l-1} \times T_l^{l-1}(1/\sigma))\,\big)\big) \\ &= (1/\sigma)(i) + \,(X \times \rho_0 \times T_l^0(1/\sigma)\,) \,+\, \cdots \,+\, (X \times \rho_{l-1} \times T_l^{l-1}(1/\sigma)\,)\ . \end{aligned}$$

We have an equation of this form for all i with $0 \leq i < l$. Thus we have obtained a system of l equations in l unknowns: $T_l^0(1/\sigma), \ldots, T_l^{l-1}(1/\sigma)$, where all the occurrences of the unknowns on the right are multiplied by a rational stream of the form $X \times \rho$. Such a system of what could be called *guarded* equations can easily be seen to have rational streams as solutions, essentially by standard linear algebraic reasoning. □

Corollary 11. *If an operator is built by function composition from: constant streams* $[r]$ *(for* $r \in A$*),* X*, sum* $+$*, convolution product* $\times$*, convolution inverse* $(-)^{-1}$*, and the zip and take operators* Z_k *and* T_l^i*, then it preserves rationality.*

Proof. For the constants, sum, product and inverse, this is trivial and for zip and take, we have Propositions 8 and 10. □

Here are some examples. Let $\sigma = 1/(1-X)^2 = (1, 2, 3, \ldots)$. We will compute

$$\alpha = T_3^0(\sigma)\ , \qquad \beta = T_3^1(\sigma)\ , \qquad \gamma = T_3^2(\sigma)\ .$$

In the computation below, we shall be using the following equalities:

$$\sigma^{(3)} = \frac{4-3X}{(1-X)^2}\ , \qquad T_3^0(X \times \sigma) = X \times \gamma\ , \qquad T_3^1(X \times \sigma) = \alpha\ , \qquad T_3^2(X \times \sigma) = \beta\ .$$

For α, we compute as follows:

$$\alpha' = \, T_3^0(\sigma^{(3)}) = \, T_3^0(\frac{4-3X}{(1-X)^2}) = \, 4\alpha - (3X \times \gamma)\ .$$

Using the fundamental theorem and $\alpha(0) = 1$ gives

$$\alpha = \, 1 + \,(4X \times \alpha) - \,(3X^2 \times \gamma)\ .$$

Similar computations lead to equations for β and γ:

$$\beta = \, 2 + \,(4X \times \beta) - \,(3X \times \alpha)\ ,$$

$$\gamma = 3 + (4X \times \gamma) - (3X \times \beta) \ .$$

Solving this system of three equations gives

$$\alpha = \frac{1+2X}{(1-X)^2} \ , \qquad \beta = \frac{2+X}{(1-X)^2} \ , \qquad \gamma = \frac{3}{(1-X)^2} \ .$$

As a next example, we will compute $\mathit{Rev}_3(\sigma)$, as follows:

$$\begin{aligned}
\mathit{Rev}_3(\sigma) &= Z_3(T_3^2(\sigma),\, T_3^1(\sigma),\, T_3^0(\sigma)) \quad \text{[definition } \mathit{Rev}_3\text{]} \\
&= Z_3\left(\frac{3}{(1-X)^2}, \frac{2+X}{(1-X)^2}, \frac{1+2X}{(1-X)^2}\right) \\
&= \frac{3}{(1-X^3)^2} + X \times \frac{2+X^3}{(1-X^3)^2} + X^2 \times \frac{1+2X^3}{(1-X^3)^2} \quad \text{[Proposition 8]} \\
&= \frac{3 - X - X^2 + 2X^3}{(1-X)^2(1+X+X^2)} \ .
\end{aligned}$$

6 Preserving Algebraicity

Corollary 11 shows that starting with a rational stream and applying some ‘basic’ operations we stay in the realm of rational streams. But there is a somewhat larger class of streams that is preserved under some of these operations, namely the class of algebraic streams defined below.

Algebraicity is a notion that should be defined over other algebraic structures. In this section we study algebraicity over finite fields. For $q \geq 1$ let $\mathbb{F}_q$ be the finite field with q elements (note that $\mathbb{F}_q$ has cardinality p^n for some prime p [Hun80]). A univariate polynomial in X is a polynomial of the form $a_0 + a_1X + \cdots + a_kX^k$ where $a_i \in \mathbb{F}_q, a_k \neq 0$. Subsequently by $\mathbb{F}_q(X)$ we denote the *field of fractions of polynomials in* X, i.e., $\pi(X) \in \mathbb{F}_q(X)$ means there are univariate polynomials $\pi_1(X), \pi_2(X)$ with coefficients in $\mathbb{F}_q$ such that $\pi(X) = \pi_1(X)/\pi_2(X)$.

Definition 12. *A stream* $\sigma \in \mathbb{F}_q^\omega$ *is* algebraic *over* $\mathbb{F}_q(X)$ *if there are* $A_i \in \mathbb{F}_q(X), A_k \neq 0$ *such that* $A_0 + A_1\sigma + \ldots + A_k\sigma^k = 0$.[2]

As an example, the stream $\sigma \in \mathbb{F}_2^\omega$ for which

$$X^3 + \frac{1}{1-X}\sigma + \frac{X+1}{1-X^2}\sigma^2 = 0 \ ,$$

is algebraic over $\mathbb{F}_2(X)$.

This definition is borrowed from the theory of formal power series [Fog02] and is motivated by the fact that σ can be considered as the sequence of coefficients of a formal power series. Following Section 2, by taking $A := \mathbb{F}_q$ we can obtain the stream calculus on $\mathbb{F}_q^\omega$. As a consequence the left hand side of expression

[2] In fact we can restrict the coefficients A_i to *univariate polynomials* instead of fractions.

above can be interpreted in two ways: as a stream in the stream calculus where $X = (0, 1, 0, \ldots)$ as in Section 2 or as a formal power series in the ring of formal power series with one variable X. It can easily be observed that each rational stream in $\mathbb{F}_q^\omega$ is algebraic. The converse does not always hold. In next section we give an example of an algebraic stream that is not rational, namely the *Prouhet–Thue–Morse* sequence. There are also streams that are not algebraic, a simple example being the Fibonacci sequence [Fog02, § 1.2.2]. But in general, the so called *automatic* streams, i.e., streams that are 'computable' by a class of transducers similar to Mealy machines[3], can be shown to be algebraic [Fog02].

We state a useful criterion, originally from [Chr79], that is usually used as an intermediate step in relating algebraic and automatic sequences but here we will use it on its own. Our formulation follows [Fog02, Theorem 3.2.1].

Definition 13. *Let $\sigma \in \mathbb{F}_q^\omega$. Then the q-*kernel *of σ is the set of subsequences of σ defined as*

$$N_q(\sigma) = \{\lambda n.\sigma(q^s n + r) \mid s \geq 0 \ , \ 0 \leq r \leq q^s - 1\} \ . \tag{6.1}$$

Here $\lambda n.f(n)$ is the notation for the sequence whose nth element is $f(n)$.

Theorem 14 (Christol). *A stream $\sigma \in \mathbb{F}_q^\omega$ is algebraic over $\mathbb{F}_q(X)$ if and only if the q-kernel $N_q(\sigma)$ of σ is finite.*

By applying this theorem we can obtain what can be considered as counterparts of Propositions 8 and 10 above. First, we have the following which resembles Proposition 10. This one is an easy consequence and is also mentioned in [Fog02], so we skip the proof.

Proposition 15. *The function take preserves algebraicity for streams over a finite alphabet: if $\sigma \in \mathbb{F}_q^\omega$ is algebraic over $\mathbb{F}_q(X)$ then so is $T_l^i(\sigma)$, for all $l \geq 2$ and $0 \leq i < l$.*

For *zip* we first need to define a notion based on q-kernels.

Definition 16. *Let $\sigma_0, \ldots, \sigma_{h-1} \in \mathbb{F}_q^\omega$ (where $h > 0$). Then h-*fold q-kernel *of $\sigma_0, \ldots, \sigma_{h-1}$ is the set of sequences defined as*

$$N_q^{(h)}(\sigma_0, \ldots, \sigma_{h-1}) = \{Z_h(\tau_0, \ldots, \tau_{h-1}) \mid \forall i \exists j, \ \tau_i \in N_q(\sigma_i)\} \ . \tag{6.2}$$

Note that we have the following trivial properties.

Proposition 17

i) If $\varsigma_0, \ldots, \varsigma_{h-1}$ is a possibly repetitive sequence such that $\varsigma_i \in \{\sigma_0, \ldots, \sigma_{h-1}\}$, then $N_q^{(h)}(\varsigma_0, \ldots, \varsigma_{h-1}) \subseteq N_q^{(h)}(\sigma_0, \ldots, \sigma_{h-1})$.
ii) If q-kernel of each of $\sigma_0, \ldots, \sigma_{h-1}$ is finite then the h-fold q-kernel of them is finite.

[3] This is a very informal description. The precise definition of automatic sequences can be found in [AS03].

We use these facts for proving that *zip* preserves algebraicity. To the best of our knowledge this result is new.

Proposition 18. *The function zip preserves algebraicity for streams over a finite alphabet: if $\sigma_0, \ldots, \sigma_{h-1} \in \mathbb{F}_q^\omega$ (where $h > 0$) are algebraic over $\mathbb{F}_q(X)$, then so is $Z_h(\sigma_0, \ldots, \sigma_{h-1})$.*

Proof. Let $\tau := Z_h(\sigma_0, \ldots, \sigma_{h-1})$. We show that

$$N_q(\tau) \subset N_q^{(h)}(\sigma_0, \ldots, \sigma_{h-1}) \ . \tag{6.3}$$

The result then will follow from Theorem 14, since the right hand side is finite.

To prove (6.3) assume $\alpha \in N_q(\tau)$. Then $\alpha \equiv \lambda n.\tau(q^s n + r)$ for some s, r as in (6.1). Assume, using division algorithm, that $q = d_0 h + r_0$ and $r = d_1 h + r_1$. Furthermore by applying (4.1) it can easily be seen that

$$\alpha \equiv Z_h(\lambda n.\tau(hnq^s + r), \lambda n.\tau((hn+1)q^s + r), \cdots, \lambda n.\tau((hn + (h-1))q^s + r)) \ .$$

So α is the *zip* of h streams each of which of the form $\tau((hn+k)q^s + r)$ where $k \leq h-1$. Again using the division algorithm assume $kr_0^s + r_1 = d_k h + r_k$. Then

$$\begin{aligned}(hn+k)q^s + r &= hnq^s + k(d_0 h + r_0)^s + d_1 h + r_1 \\ &= hnq^s + k(d_0^s h^s + sd_0^{s-1} h^{s-1} r_0 + \cdots + sd_0 h r_0^{s-1} + r_0^s) + d_1 h + r_1 \\ &= h(nq^s + kd_0^s h^{s-1} + skd_0^{s-1} h^{s-2} r_0 + \cdots + skd_0 r^{s-1} + d_1) + d_k h + r_k \\ &= h(nq^s + U_k) + r_k \ ,\end{aligned}$$

where

$$U_k = kd_0^s h^{s-1} + skd_0^{s-1} h^{s-2} r_0 + \cdots + skd_0 r^{s-1} + d_1 + d_k \ .$$

From this and using the property of zip in (4.1) we get

$$\lambda n.\tau((hn+k)q^s + r) \equiv \lambda n.\tau(h(nq^s + U_k) + r_k) \equiv \lambda n.\sigma_{r_k}(nq^s + U_k) \ .$$

It remains to be checked whether $U_k < q^s$. But this is evident because

$$\begin{aligned}hU_k &= k(q^s - r_0^s) + d_1 h + d_k h \\ &= kq^s + r - r_k \\ &\leq (h-1)q^s + r \\ &< hq^s \ .\end{aligned}$$

Therefore defining $\upsilon_k := \lambda n.\sigma_{r_k}(nq^s + U)$ we obtain $\upsilon_0 \in N_q(\sigma_{r_0}), \ldots, \upsilon_{h-1} \in N_q(\sigma_{r_{h-1}})$ such that

$$\alpha \equiv Z_h(\upsilon_0, \ldots, \upsilon_{h-1}) \ .$$

Hence, by (6.2) and Proposition 17 we have $\alpha \in N_q^{(h)}(\sigma_0, \ldots, \sigma_{h-1})$. □

In general, the *zip* of algebraic sequence need not be algebraic over a field whose cardinality is the number of arguments of *zip*. This is a consequence of the following result in [Cob69] where it is stated in terms of automatic sequences. Here we rephrase it in terms of algebraicity over finite fields.

Theorem 19 (Cobham). *Let $\sigma \in \mathbb{F}_{q_0}^{\omega} \cap \mathbb{F}_{q_1}^{\omega}$ be algebraic over* two *fields $\mathbb{F}_{q_0}(X)$ and $\mathbb{F}_{q_1}(X)$. Then either σ is rational or q_0 and q_1 are powers of the same prime number.*

According to this theorem if $\sigma_0, \sigma_1, \sigma_2 \in \mathbb{F}_2^{\omega}$ are non-rational binary streams that are algebraic over $\mathbb{F}_2(X)$ (e.g. the sequence Ψ defined in next section) then $Z_3(\sigma_0, \sigma_1, \sigma_2)$ cannot be algebraic over $\mathbb{F}_3(X)$.

Finally, we remark that the sum of two algebraic streams is algebraic. The proof is a straightforward application of Theorem 14, together with a similar construct to the one in (6.2).

7 Stream Circuits

We briefly recall the correspondence between rational streams (of real numbers) and so-called stream circuits built from adder, copier, register and multiplier gates. Then we propose to look at stream circuits built from this set of gates extended with basic gates for splitting and merging. We study their behaviour by describing how they act on input streams of real numbers. For circuits without feedback, it will be immediate that they preserve rationality. For feedback circuits, the situation turns out to be more complicated.

Stream circuits [Rut05b] are data flow networks that act on streams of inputs (here real numbers) and produce streams of outputs. They are built out of four types of basic gates by means of composition, which amounts simply to connecting (single) output ends to (single) input ends. Below we describe the basic gates and their input-output behaviour. An *r-multiplier*, for $r \in A$, transforms an input stream $\sigma \in A^{\omega}$ into $[r]\times\sigma$:

$$\begin{tikzcd} \sigma \arrow[r, mapsto, "r"] & {[r]\times\sigma} \end{tikzcd}$$

which amounts to the element-wise multiplication of the input values with r. A *register* (with initial value 0) takes an input stream σ

$$\begin{tikzcd} \sigma \arrow[r, mapsto] & \boxed{R} \arrow[r] & (X \times \sigma) \end{tikzcd}$$

and outputs it with one step delay, after having output the initial value 0 first. An *adder* takes two input streams σ and τ and outputs the stream consisting of their element-wise addition; and a *copier* simply copies input streams into output streams:

$$\begin{tikzcd} \sigma \arrow[dr, mapsto, bend left] & & & & & \sigma \\ & + \arrow[r] & \sigma+\tau & \sigma \arrow[r, mapsto] & C \arrow[ur, bend left] \arrow[dr, bend right] & \\ \tau \arrow[ur, mapsto, bend right] & & & & & \sigma \end{tikzcd}$$

Stream circuits are then built by composing various basic gates. Here is a simple example of a circuit with feedback:

$$\begin{tikzcd}
& \circ \arrow[d, mapsto] & \boxed{R} \arrow[l] & \circ \arrow[l, mapsto] & \\
{} \arrow[r, mapsto] & + \arrow[r] & \circ \arrow[r, mapsto] & C \arrow[u] \arrow[r] & {}
\end{tikzcd}$$

For an input stream $\sigma \in A^\omega$, we can compute the output stream as a function $f(\sigma)$ of σ as follows. With the three internal composition nodes of the circuit, we associate streams $\rho_1, \rho_2, \rho_3 \in A^\omega$:

$$\begin{tikzcd}
& \rho_1 \arrow[d, mapsto] & \boxed{R} \arrow[l] & \rho_2 \arrow[l, mapsto] & \\
\sigma \arrow[r, mapsto] & + \arrow[r] & \rho_3 \arrow[r, mapsto] & C \arrow[u] \arrow[r] & f(\sigma)
\end{tikzcd}$$

For each of the three basic gates used in this circuit, we have an equation:

$$\rho_1 = X \times \rho_2 \ , \qquad \rho_3 = \sigma + \rho_1 \ , \qquad \rho_2 = \rho_3 = f(\sigma) \ .$$

Eliminating the streams ρ_1, ρ_2 and ρ_3 from this system of equations, we find

$$f(\sigma) = \frac{1}{1-X} \times \sigma \ .$$

In [Rut05b, Theorem 4.25], it is shown that every (finite) circuit possibly with feedback loops (which always have to pass through at least one register), compute stream functions $f : A^\omega \to A^\omega$ of the form: $f(\sigma) = \rho \times \sigma$, for all σ and some fixed rational stream ρ; conversely, every such function is implemented by some finite circuit.

Next we introduce new basic gates for the splitting and merging of streams.

A *splitter* gate in our setting is a gate with one input and two output ends:

$$\begin{tikzcd}
& \tau & \\
\sigma \arrow[r, mapsto] & \boxed{S} \arrow[u, "\bullet"] \arrow[r] & \upsilon
\end{tikzcd}$$

It transforms an input stream $\sigma \in \mathbb{R}^\omega$ to streams τ, υ such that

$$\tau = D_2^1(\sigma) = T_2^0(\sigma) \ , \qquad \upsilon = D_2^0(\sigma) = T_2^1(\sigma) \ .$$

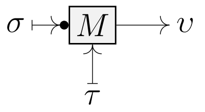
Note that $\tau = \mathit{even}(\sigma)$ and $\upsilon = \mathit{even}(\sigma')$ (where *even* is defined in Section 3). We define

$$\mathit{odd}(\sigma) := \mathit{even}(\sigma') \ .$$

Hence the splitters transforms σ to $\mathit{even}(\sigma)$ and $\mathit{odd}(\sigma)$.

The splitter is different from the previous ports (in particular copier) in that only one of its outgoing ports is active at any time. This means when a data element belonging to τ is being output, the port outputting υ is pending. Moreover, the active output port alternates with each data consumed from σ. The bullet on one of the output ports denotes the port that activates in the very beginning. This confirms the fact that $\tau = \mathit{even}(\sigma)$.

A *merger* gate is a gate with two inputs and one output end.



It transforms two input streams $\sigma, \tau \in \mathbb{R}^\omega$ to a stream υ such that

$$\upsilon = Z_2(\sigma, \tau) \enspace .$$

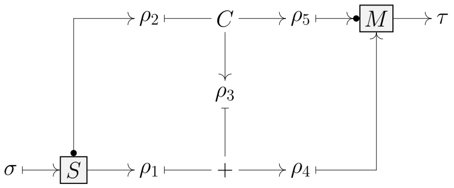
In contrast with the splitter gate, in a merger only one of the inputs is activated at a time. The active input port alternates with each data output. Again the bullet denotes the port that is activated in the very beginning, i.e., the one that contributes to υ_0.

It is clear that merger and splitter can be composed with each other and with the previously defined gates to form compound circuits. We call such a circuit an *extended stream circuit.* The functions $f(\sigma) = \rho \times \sigma$, for constant stream ρ, that are realisable by well-formed stream circuits are instances of *causal* functions on streams [Rut05b]. These are functions that output a data item after each input. Since each gate of stream circuit is causal their composition is causal too. However, introducing splitter and merger into the extended stream circuits leads to *overconsumption* (splitter) or *overproduction* (merger). So there will be data queues behind causal gates. Hence we need to assume the following important rule:

The connecting lines in extended stream circuits behave like unbounded FIFO buffers.

This is similar to the framework of Kahn Networks [Kah74].

Simple feed-forward extended stream circuits can easily be analysed using the same method used for stream circuit. As an example consider the following circuit [Mak08, § 4].



First note that,

$$\begin{aligned} \rho_1 &= \mathit{odd}(\sigma) \enspace , \\ \rho_2 = \rho_3 = \rho_5 &= \mathit{even}(\sigma) \enspace , \\ \rho_4 &= \mathit{odd}(\sigma) + \mathit{even}(\sigma) \enspace , \\ \tau &= Z_2(\mathit{even}(\sigma), \mathit{odd}(\sigma) + \mathit{even}(\sigma)) \enspace . \end{aligned}$$

Assume we input the stream $\sigma = X/(1-X)^2 = (0, 1, 2, \cdots)$ to the above circuit. It can easily be shown that (cf. the example at the end of Section 5),

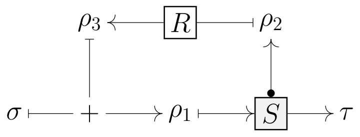
$$\mathit{even}(\sigma) = \frac{2X}{(1-X)^2} \enspace , \qquad \mathit{odd}(\sigma) = \frac{1+X}{(1-X)^2} \enspace .$$

Subsequently we derive

$$\begin{aligned}\tau &= Z_2(\frac{2X}{(1-X)^2}, \frac{1+3X}{(1-X)^2}) \\ &= \frac{2X^2}{(1-X^2)^2} + \frac{X+3X^3}{(1-X^2)^2} = \frac{X(1+2X+3X^2)}{(1-X)^2(1+X)^2} \ .\end{aligned}$$

Evidently, by sequencing splitters and mergers one can synthesise feed-forward circuits for calculating dyadic (2^n-ary) *take* and *zip* and functions. I.e., we can build circuits for calculating $T^l_{2^n}$ Z_{2^n}. This suggests that by adding new splitter and merger gates with p input and output ports, where p is a prime number, we can synthesise circuits for calculating general *take* and *zip* functions T^l_n and Z_n. We do not consider this issue in the present paper.

While feed-forward extended stream circuits are relatively easy to analyse, allowing feed-back will complicate the matter. First of all we need to formulate well-formedness rules with respect to the topology of the circuit, whose purpose would be to prevent overconsumption from happening (overproduction is not a problem, since we assume that connecting lines are buffers). Intuitively this means that for any possible path in the circuit, splitters should be directly connected to the global input or be preceded by appropriate number of mergers. In future work we plan to make such rules more formal. For now we give an example a non well-formed circuit demonstrating the problem of overconsumption.



In the circuit above, assuming there is a flow, one can take the second derivative of the behavioural equations for ρ_1 and obtain the contradiction in the form of following identity.

$$\rho_1(2) = \sigma(2) + \rho_1(2) \ .$$

We conclude this section by giving an example of a non-rational stream that can be calculated using the extended stream circuits. This will demonstrate that adding splitter and merger will indeed extend the class of definable streams with respect to those of the ordinary stream calculus. Our example is the *Prouhet–Thue–Morse* sequence which is an algebraic non-rational[4] stream over $\mathbb{F}_2(X)$. The stream, which we denote by Ψ is given by the following behavioural differential equations.

$$\begin{aligned}&\Psi(0) = 0 \ , \qquad \Psi'(0) = 1 \ , \\ &\Psi'' = Z_2(\Psi', \overline{\Psi'}) \ ;\end{aligned}$$

where $\overline{\sigma}$ is the bit-wise negation of σ itself defined as

$$\overline{\sigma}(0) = \neg\sigma(0) \ , \qquad \overline{\sigma}' = \overline{\sigma'} \ .$$

[4] Proof of this fact can be found in [Fog02].

Consider following extended circuit which contains only one merger.

$$\begin{tikzcd}
 & \rho_3 \arrow[d] & C \arrow[l] \arrow[rrr] & & & \rho_5 \arrow[d] & & & \\
 & \boxed{R} \arrow[d] & \rho_2 \arrow[u] & & \rho_{11} \arrow[dd] & + \arrow[l] & & & \\
 & \rho_4 \arrow[r] & + \arrow[u] & & & \rho_{10} \arrow[u] & & & \\
 & & & & \boxed{M} \arrow[dl] & & & & \\
 & & \rho_1 \arrow[uu] & \rho_{13} \arrow[d] & & \rho_9 \arrow[uu, "-1"'] & & & \\
 & & & \boxed{R} \arrow[dd] & \rho_{12} \arrow[uu] & C \arrow[l] \arrow[u] & & & \\
\sigma \arrow[rr] & & C \arrow[uu] \arrow[dd] & & & & & & \\
 & & & \rho_{14} \arrow[d] & & \rho_8 \arrow[uu] & & & \\
 & & \rho_6 \arrow[r] & + \arrow[r] & \rho_7 \arrow[r] & C \arrow[u] \arrow[r] & \rho_{15} \arrow[r] & \boxed{R} \arrow[r] & \tau
\end{tikzcd}$$

Note that the -1-multiplier is meaningful since we are working in a field. Then by calculating the intermediate values ρ_i one observes that:

$$\begin{aligned}
&\rho_1 = \rho_6 = \sigma \ , \\
&\rho_2 = \rho_3 = \rho_5 = \sigma + X \times \rho_2 = \frac{1}{1-X} \times \sigma \ , \\
&\rho_4 = \frac{X}{1-X} \times \sigma \ , \\
&\rho_7 = \rho_8 = \rho_9 = \rho_{12} = \rho_{15} = \sigma + \rho_{14} \ , \\
&\rho_{10} = -\rho_7 \ , \\
&\rho_{11} = \frac{1}{1-X} \times \sigma - \rho_7 \ , \\
&\rho_{13} = Z_2(\rho_7, \frac{1}{1-X} \times \sigma - \rho_7) \ , \\
&\rho_{14} = X \times Z_2(\rho_7, \frac{1}{1-X} \times \sigma - \rho_7) \ , \\
&\tau = X \times \rho_7 \ .
\end{aligned}$$

Form here we can obtain

$$\rho_7 = \sigma + X \times Z_2(\rho_7, \frac{1}{1-X} \times \sigma - \rho_7) \ .$$

Hence if $\sigma = [1] = (1, 0, 0, \cdots)$ is input to this circuit then $\tau = \Psi$.

8 Discussion and Future Work

We have studied various data independent operations for partitioning, projecting or merging streams. These operations are usually studied in the context of dataflow programming, while we showed that the operations and many of their properties can be defined using elements of *stream calculus*, namely behavioural differential equations for definitions and coinduction proof principle for proofs. Furthermore we focused on *take* and *zip* operations, for merging and splitting of data that are widely used elements in dataflow programming [BŞ01, Mak08] and models of concurrency [Arb04]. We dealt with the fact that splitting and merging preserves well behaved and well patterned class of streams namely rational and algebraic streams. While some of those results were known in the literature, we present them in the framework of stream calculus. Finally we showed how adding two new gates, namely dyadic merger and splitter will enlarge the class of streams that are realisable using stream circuits to beyond rational streams and into the realm of algebraic streams.

There are several issues and directions for future work.

Automated coinduction proofs. In Section 3 we showed how to use coinduction to prove the Drop exchange rule by finding a bisimulation. There are in fact tools for automatically finding bisimulation, e.g. the CIRC tool [LR07]. We applied CIRC and it could drive the rule $D_2^0 = D_4^0 \circ D_5^2 \circ D_6^4$. The CIRC tool uses a special technique called *circular coinduction*, a partial decision procedure, whose success depends on the type of bisimulation to be found. Our goal is to further investigate the different types of bisimulation that will arise in *Periodic Drop Take Calculus (PDTCS)* of Mak [Mak08] and examine the applicability of circular coinduction to them.

Extended stream circuits. We plan to investigate precisely which class of streams are realisable using extended stream circuits of Section 7. For this we will also study extended circuits with p-adic merger and splitter where p is a prime number. Moreover the question of well-formedness with respect to the topological properties of the circuits needs to be investigated. As a related problem we are interested in finding a closed formula for *even* and *odd* (and their n-ary counterparts). Intuitively these functions correspond to the roots of unity (cf. [Wil94, § 2.4], and Lemma 9 on periodicity of *take*). This implies that one could use hyperbolic functions (e.g. cosh) to represent the effect of *even* in the stream calculus. We plan to make this connection more formal.

Coalgebraic semantics. Earlier work on stream calculus has led to a coalgebraic treatment of rational power series [Rut08]. Advantage of the coalgebraic modelling is that it present a unified way for dealing with stream circuits, stream functions and transducers. Above all it helps in dealing with various types of bisimulations. We intend to study the material of Section 7 in a coalgebraic setting, by looking into the systems based on causal functions and beyond [Rut06, UV08, Kim08].

Acknowledgements. We thank the anonymous referees for their comments.

References

[Arb04] Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14, 329–366 (2004)

[AS03] Allouche, J.-P., Shallit, J.: Automatic sequences: theory, applications, generalizations. Cambridge University Press, Cambridge (2003)

[BR88] Berstel, J., Reutenauer, C.: Rational series and their languages. EATCS Monographs on Theoretical Computer Science, vol. 12. Springer, Heidelberg (1988)

[BŞ01] Broy, M., Ştefănescu, G.: The algebra of stream processing functions. Theoret. Comput. Sci. 258(1-2), 99–129 (2001)

[Chr79] Christol, G.: Ensembles presque periodiques k-reconnaissables. Theoret. Comput. Sci. 9(1), 141–145 (1979)

[Cob69] Cobham, A.: On the base-dependence of sets of numbers recognizable by finite automata. Math. Systems Theory 3, 186–192 (1969)

[Fog02] Pytheas Fogg, N.: Substitutions in dynamics, arithmetics and combinatorics. In: Berthé, V., Ferenczi, S., Mauduit, C., Siegel, A. (eds.). Lecture Notes in Math., vol. 1794. Springer, Berlin (2002)

[GKP94] Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete mathematics, 2nd edn. Addison-Wesley, Reading (1994)

[Hun80] Hungerford, T.W.: Algebra. Graduate Texts in Mathematics, vol. 73, Springer, New York (1980); Reprint of the 1974 original

[Kah74] Kahn, G.: The semantics of a simple language for parallel programming. In: Information Processing 74: Proceedings of IFIP Congress 74, Stockholm, August 1974, vol. 74, pp. 471–475. North Holland Publishing Co., Amsterdam (1974)

[Kim08] Kim, J.: Coinductive properties of causal maps. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 253–267. Springer, Heidelberg (2008)

[LR07] Lucanu, D., Roşu, G.: CIRC: A circular coinductive prover. In: Mossakowski, T., Montanari, U., Haveraaen, M. (eds.) CALCO 2007. LNCS, vol. 4624, pp. 372–378. Springer, Heidelberg (2007)

[Mak08] Mak, R.H.: Design and Performance Analysis of Data-independent Stream Processing Systems. PhD thesis, Technische Universiteit Eindhoven (2008)

[Rut05a] Rutten, J.J.M.M.: A coinductive calculus of streams. Mathematical Structures in Computer Science 15, 93–147 (2005)

[Rut05b] Rutten, J.J.M.M.: A tutorial on coinductive stream calculus and signal flow graphs. Theoretical Computer Science 343(3), 443–481 (2005)

[Rut06] Rutten, J.J.M.M.: Algebraic specification and coalgebraic synthesis of Mealy automata. In: Proceedings of FACS 2005. ENTCS, vol. 160, pp. 305–319. Elsevier Science Publishers, Amsterdam (2006)

[Rut08] Rutten, J.J.M.M.: Rational streams coalgebraically. Logic. Methods in Comput. Sci. 4(3:9), 1–22 (2008)

[UV08] Uustalu, T., Vene, V.: Comonadic notions of computation. In: Adámek, J., Kupke, C. (eds.) Proc. of CMCS 2008. ENTCS, vol. 203(5), pp. 263–284. Elsevier, Amsterdam (June 2008)

[Wil94] Wilf, H.S.: Generatingfunctionology. Academic Press, London (1994)

Generic Point-free Lenses

Hugo Pacheco and Alcino Cunha

DI-CCTC, Universidade do Minho, Braga, Portugal
{hpacheco,alcino}@di.uminho.pt

Abstract. Lenses are one the most popular approaches to define bidirectional transformations between data models. A bidirectional transformation with *view-update*, denoted a *lens*, encompasses the definition of a forward transformation projecting *concrete models* into *abstract views*, together with a backward transformation instructing how to translate an abstract view to an update over concrete models. In this paper we show that most of the standard point-free combinators can be lifted to lenses with suitable backward semantics, allowing us to use the point-free style to define powerful bidirectional transformations by composition. We also demonstrate how to define generic lenses over arbitrary inductive data types by lifting standard recursion patterns, like folds or unfolds. To exemplify the power of this approach, we "lensify" some standard functions over naturals and lists, which are tricky to define directly "by-hand" using explicit recursion.

Keywords: point-free, bidirectional transformations, lenses, recursion patterns, inductive types.

1 Introduction

With the ever growing list of programming languages and application development frameworks, transforming a data format into a different format is essential to "bridge the gap" between technology layers and ensure sharing of information among software applications. Moreover, users generally expect transformations to be bidirectional, in the sense that changes made to one of the models can be safely propagated to its connected pair (imagine the synchronization of a laptop's and a cellphone's contact list).

The naive way to create a *bidirectional transformation* is to engineer two unidirectional transformations together and manually prove that they are somehow consistent. This is likely to cause a maintenance problem, besides being a notoriously expensive and error-prone task. Any change in a data format implies a redefinition of both transformations, and a new consistency proof.

A better approach is to design a domain-specific language in which one expression denotes both transformations, which are then guaranteed to be consistent by construction in the respective semantic space. Following this notion, approaches to bidirectional transformations have emerged in the most diverse computing domains, including heterogeneous data synchronization [15,6], software model transformation [26], schema evolution [9,4], constraint maintenance for graphical user interfaces [21], interactive structure editing [19] and relational databases [7]. By restricting the domain-specific language to particular domains, these approaches overcome the difficulty of designing bidirectional transformations, and achieve a neat balance between expressiveness and the robustness imposed by the consistency constraints.

One of the most successful approaches to bidirectional transformations are the so-called *lenses*, proposed by Foster *et al.* [15] to solve the classical *view-update problem* originating from database theory [3]: when a concrete data model is abstracted into a view, how can changes made to the view be propagated back as updates to the original model? According to the following diagram, a *lens* comprises the definition of three functions involving a concrete data model C and its abstract counterpart A:

$$\begin{tikzcd}
C \arrow[rr, bend left, "get"] \arrow[rr, phantom, "\trianglerighteq"] & & A \arrow[ll, bend left, "create"] \\
& A \times C \arrow[ul, "put"] \arrow[ur, "\pi_1"'] &
\end{tikzcd}$$

The first ingredient of a lens consists of the definition of a view: function $get : C \rightarrow A$ abstracts away details from the concrete model that are irrelevant for a specific purpose. Since this abstraction implies loss of information, the backwards transformation $put : A \times C \rightarrow C$ is augmented with knowledge of the original concrete instance, rendering it capable to restore some information no longer present in the view. As this is not always possible, a default concrete model is sometimes reconstructed by applying $create : A \rightarrow C$ to the view.

Of course, these three functions should be somehow consistent in order to define a *well-behaved* lens. First, *get* must be an abstraction function, i.e, A shall contain at most as much information as C. In a sense, a lens is a dual concept of refinement [23,25]. Second, the lens should be *acceptable*, i.e, updates to a view cannot be ignored and must be translated exactly. Finally, the lens should be *stable*, i.e, if the view does not change, then neither should the source. These properties will be formally defined in the next section.

As an example, let's consider as a concrete data model lists of natural numbers. A possible lens over this data type is determined by the length of the list. In Haskell we could define the *get* function trivially as follows:

```
data Nat = Zero | Succ Nat
get :: [Nat] → Nat
get []       = Zero
get (x : xs) = Succ (get xs)
```

Given a natural number, *create* must generate a default list of that length:

```
create :: Nat → [Nat]
create Zero     = []
create (Succ n) = Zero : create n
```

For the lens to be well-behaved, *create* could use other defaults but cannot create a list with a different length. The *put* function is a bit more tricky. A possible definition that guarantees well-behavedness is:

$$\begin{array}{ll} put :: (Nat, [Nat]) \rightarrow [Nat] & \\ put\ (Zero, \_) & = [\,] \\ put\ (Succ\ n, [\,]) & = Zero : put\ (n, create\ n) \\ put\ (Succ\ n, x : xs) & = x : put\ (n, xs) \end{array}$$

If the view (i.e, the length of the list) remains the same or decreases, elements of the original list must be used in the new concrete value. If the length increases, defaults are invented for the new elements.

For more complex data formats and abstractions, the definition of a *put* function that guarantees well-behavedness becomes highly complex. As such, Foster *et al.* [15] propose a combinatorial approach to the definition of lenses over generalized trees: complex lenses are defined by composition of more simpler lenses using a standard set of combinators and recursion. This combinatorial approach is also central to the so-called *point-free* style of programming, popularized by John Backus in his 1977 Turing award lecture [2]. In this variable-free style, functions are defined by composition using a standard set of higher-order combinators, characterized by a rich set of algebraic laws that make this style particularly amenable for program calculation.

In this paper we explore precisely this connection and develop a library of point-free lens combinators. In the next section, we establish that most of the standard point-free combinators define well-behaved lenses. This opens interesting perspectives towards a lens calculus, with practical applications for the optimization of complex lenses defined by composition. In the point-free style of programming, general recursion is usually deterred in favor of more calculation-friendly recursion patterns. In Section 3 we show how to define generic lenses over arbitrary inductive data types by lifting standard recursion patterns, namely folds and unfolds. In principle, this makes it simpler to establish that a lens is well-behaved, when compared to the general recursion approach followed by Foster *et al.* [15], where non-trivial conditions must be proved every time a lens is defined by recursion. In Section 4 we discuss some relevant related work, and we conclude in Section 5 with a synthesis of the main contributions and pointers for future work.

2 Point-free Combinators as Lenses

The rather standard set of point-free combinators that we will use in this paper is shown in Figure 1. Although we give our examples in Haskell, the semantic domain will be the SET category, where objects are sets (types) and arrows are total functions. The most fundamental combinators are the *composition* of $f : B \rightarrow C$ after $g : A \rightarrow B$, denoted by $f \circ g : A \rightarrow C$, and the *identity* function, denoted by $id : A \rightarrow A$. The *projections* $\pi_1 : A \times B \rightarrow A$ and $\pi_2 : A \times B \rightarrow B$ project out the left and right components of a pair, respectively, and the *split*

$$\begin{array}{lll}
id & : & A \to A \\
\circ & : & (B \to C) \to (A \to B) \to (A \to C) \\
\pi_1 & : & A \times B \to A \\
\pi_2 & : & A \times B \to B \\
\triangle & :: & (A \to B) \to (A \to C) \to (A \to B \times C) \\
\times & : & (A \to B) \to (C \to D) \to (A \times C \to B \times D) \\
i_1 & : & A \to A + B \\
i_2 & : & B \to A + B \\
\nabla & : & (A \to C) \to (B \to C) \to (A + B \to C) \\
+ & : & (A \to B) \to (C \to D) \to (A + C \to B + D) \\
! & : & A \to 1 \\
\underline{\cdot} & : & B \to (A \to B)
\end{array}$$

Fig. 1. Point-free combinators

combinator $f \triangle g : A \to B \times C$ builds a pair by applying $f : A \to B$ and $g : A \to C$ to the same input value. The derived *product* combinator $f \times g : A \times C \to B \times D$ applies $f : A \to B$ and $g : C \to D$ to the left and right elements, respectively, of the input pair in order to build a new pair. The *injections* $i_1 : A \to A + B$ and $i_2 : B \to A + B$ build left and right alternatives of a disjoint sum, respectively, and the *either* combinator $f \nabla g : A + B \to C$ applies $f : A \to C$ if the input is a left alternative or $g : B \to C$ otherwise. The derived *sum* combinator $f + g : A + C \to B + D$ uses $f : A \to B$ to build a left alternative from a left alternative or $g : C \to D$ otherwise. The *bang* combinator $! : A \to 1$ returns the single element of the terminal object, and, given a constant $b : B$, $\underline{b}$ always returns b. Some of the laws governing these combinators are presented in Appendix A, and can easily be derived from their uniqueness laws. For more information on point-free program calculation in general see [5,17,22,11].

Using these combinators, we can give a precise point-free characterization of well-behaved lenses.

Definition 1 (Lens). *A well-behaved lens l, denoted by $l : C \trianglerighteq A$, is a bidirectional transformation that comprises three total functions $get : C \to A$, $put : A \times C \to C$ and $create : A \to C$, satisfying the following properties:*

$$get \circ create = id \qquad \text{CreateGet}$$

$$get \circ put = \pi_1 \qquad \text{PutGet}$$

$$put \circ (get \triangle id) = id \qquad \text{GetPut}$$

Property CreateGet [6] guarantees that the lens is an abstraction, ensuring that *get* is a surjection and *create* an injection. PutGet and GetPut [15] guarantee that the lens is acceptable and stable, respectively.

We will now show how to lift most of the point-free combinators of Figure 1 to lenses. To avoid introducing new notation, we will denote the lens corresponding to a particular combinator using the same syntax. From the context it should be clear if we are referring to the lens or the point-free combinator. For some lenses there is some freedom in the design of backwards transformations (namely, *create* and *put*). As such, they can receive extra parameters to plugin in contexts were such freedom exists.

The simplest cases of bidirectional transformations are isomorphisms. Given a bijective function $f : A \to B$ with inverse $f^{-1} : B \to A$, there exists a lens $f : A \trianglerighteq B$ with:

$$\begin{aligned} get &= f \\ put &= f^{-1} \circ \pi_1 \\ create &= f^{-1} \end{aligned}$$

A primitive combinator that falls under this category is the identity function $id : A \trianglerighteq A$. Similarly, all usual isomorphisms involving sums and products are lenses. Here are some that we will use throughout the paper:

$$\begin{aligned} swap &: A \times B \trianglerighteq B \times A \\ assocr &: (A \times B) \times C \trianglerighteq A \times (B \times C) \\ assocl &: (A \times (B \times C)) \trianglerighteq ((A \times B) \times C) \\ coassocl &: A + (B + C) \trianglerighteq (A + B) + C \\ distr &: A \times (B + C) \trianglerighteq (A \times B) + (A \times C) \\ distl &: (A + B) \times C \trianglerighteq (A \times C) + (B \times C) \end{aligned}$$

One of the most important properties of lenses is composability. In point-free, the composition of two lenses, first defined in [15], can be restated as follows:

$$\begin{aligned} &\forall f : B \trianglerighteq A,\ g : C \trianglerighteq B.\ (f \circ g) : C \trianglerighteq A \\ &\quad get = get_f \circ get_g \\ &\quad put = put_g \circ (put_f \circ (id \times get_g) \bigtriangleup \pi_2) \\ &\quad create = create_g \circ create_f \end{aligned}$$

If the concrete domain of f and the abstract domain of g have the same type, then f and g are composable and $f \circ g$ is a lens with the concrete domain of g and the abstract domain of f. In the *get* and *create* directions, the composed transformation is just the composition of the respective transformations from f and g. In the *put* direction, in order to apply the *put* functions in sequence, the original concrete value is duplicated. Note that, while put_g consumes the original concrete value with type C, the concrete value passed to put_f, with type B, is calculated by applying the function get_g to the original concrete value.

The projections π_1 and π_2 will be the main ingredients in defining more complex lenses that project away components of a concrete data type:

$$\begin{aligned} &\forall b \in B.\ {\pi_1}^b : A \times B \trianglerighteq A \\ &\quad get = \pi_1 \\ &\quad put = id \times \pi_2 \\ &\quad create = id \bigtriangleup \underline{b} \end{aligned} \qquad \begin{aligned} &\forall a \in A.\ {\pi_2}^a : A \times B \trianglerighteq B \\ &\quad get = \pi_2 \\ &\quad put = swap \circ (id \times \pi_1) \\ &\quad create = \underline{a} \bigtriangleup id \end{aligned}$$

Since π_1 and π_2 project the corresponding elements of the product, the backward transformations have to reconstruct the projected out elements. In *create*, a default value is inserted for the "lost" value of the pair, while *put* copies it from the original pair. Therefore, the derived lenses accept additional parameters (the constants a and b), represented using superscript.

In general, the split of two lenses $f : C \trianglerighteq A$ and $g : C \trianglerighteq B$ sharing the same domain is not a well-behaved lens $f \vartriangle g : C \trianglerighteq A \times B$. For example, the duplication combinator $id \vartriangle id : A \trianglerighteq A \times A$ would be a valid lens iff the invariant $\pi_1 = \pi_2$ was imposed on the codomain $A \times A$, stating that both components of the pair are always equal. However, if f and g project distinct concrete information from C then it is possible to define a well-behaved lens. An example of such a lens is the *swap* isomorphism $\pi_2 \vartriangle \pi_1 : A \times B \trianglerighteq B \times A$. When this *non-interference* between f and g exists, updates to the view can be propagated back to the concrete model by independent inspection of both components of the pair. In practice this means that, when defining $put : (A \times B) \times C \to C$ the order of application of $put_f : A \times C \to C$ and $put_g : B \times C \to C$ should be irrelevant. Formally, this non-interference condition can be expressed by the following equality:

$$put_f \circ (id \times put_g) \circ assocr = put_g \circ (id \times put_f) \circ assocr \circ (swap \times id)$$

Given a split $f \vartriangle g$ where this non-interference condition is valid, it should be possible to lift it into a well-behaved lens by defining *put* as any of the above expressions (for example, $put_f \circ (id \times put_g) \circ assocr$). Unfortunately, we are unaware of a general definition for *create* that obeys the CreateGet law, which prevents us from giving a generic definition of split as a well-behaved lens. For *swap* it is rather easy to show that the non-interference condition is valid, that the suggested definition for *put* is equal to the expected $swap \circ \pi_1$ (according to the previous generic definition of a bijection as a well-behaved lens), and that *create* can be done using *swap* itself.

Another instance of split that satisfies the non-interference condition is the product combinator $f \times g = f \circ \pi_1 \vartriangle g \circ \pi_2$. Again, it is easy to show that any of the above alternative definitions for *put* is equivalent to $(put_f \times put_g) \circ distp$, where *distp* is the isomorphism given by:

$$\begin{aligned} &distp : (C \times D) \times (A \times B) \to (C \times A) \times (D \times B) \\ &distp = (\pi_1 \times \pi_1) \vartriangle (\pi_2 \times \pi_2) \qquad \textsc{Distp-Def} \end{aligned}$$

For this particular split, creating a concrete value from an abstract one can be done by independently creating both components of the pair, leading to the following definition:

$$\begin{aligned} &\forall f : C \trianglerighteq A,\ g : D \trianglerighteq B.\ f \times g : C \times D \trianglerighteq A \times B \\ &\quad get \quad\ = get_f \times get_g \\ &\quad put \quad\ = (put_f \times put_g) \circ distp \\ &\quad create = create_f \times create_g \end{aligned}$$

In practice, most expressions involving split that satisfy non-interference can be transformed into point-free expressions using other valid lens product combinators and isomorphisms (like $\times$ or *swap*).

Moving to sums, we have two alternative ways to generically lift the either combinator into a well-behaved lens:

$$\begin{array}{l}\forall f : A \to C,\ g : B \to C.\ f \mathbin{{}_\bullet\nabla} g : A + B \trianglerighteq C \\ \quad \mathit{get} \quad\ \ = \mathit{get}_f \nabla \mathit{get}_g \\ \quad \mathit{put} \quad\ \ = (\mathit{put}_f + \mathit{put}_g) \circ \mathit{distr} \\ \quad \mathit{create} = i_1 \circ \mathit{create}_f \\ \forall f : A \to C,\ g : B \to C.\ f \mathbin{\nabla_\bullet} g : A + B \trianglerighteq C \\ \quad \mathit{get} \quad\ \ = \mathit{get}_f \nabla \mathit{get}_g \\ \quad \mathit{put} \quad\ \ = (\mathit{put}_f + \mathit{put}_g) \circ \mathit{distr} \\ \quad \mathit{create} = i_2 \circ \mathit{create}_g \end{array}$$

When putting back, put_f is used if the concrete value is a left alternative and put_g otherwise. For *create* we have two alternatives – either output a left or a right alternative – originating left-biased (${}_\bullet\nabla$) and right-biased ($\nabla_\bullet$) versions of this lens. Assuming that predicates are represented using sums (for example, using $p : A \to A + A$ instead of $p : A \to \mathit{Bool}$), this lens corresponds to a point-free formulation of the concrete conditional combinator *ccond* from [15].

The sum injections $i_1 : A \to A + B$ and $i_2 : B \to A + B$ are non-surjective functions and classic examples of refinements [9]. The only way to lift them into lenses would be by imposing an invariant on the codomain $A + B$, constraining its values to be all left or all right alternatives, respectively. Since this semantic constraint is not supported by standard type systems, unrestricted usage of the injections will be disallowed for well-behaved lenses. Notwithstanding, if injections are used inside an expression that is jointly surjective, they can sometimes build up well-behaved lenses. Two particular useful cases are the lenses $i_1 \nabla f : A + C \trianglerighteq A + B$ and $f \nabla i_2 : C + B \trianglerighteq A + B$, where $f : C \to A + B$ is any lens. Notice that these eithers are necessarily surjective because f, being a well-behaved lens, is already surjective. For example, the first lens can be defined as follows:

$$\begin{array}{l}\forall f : C \trianglerighteq A + B.\ i_1 \nabla f : A + C \trianglerighteq A + B \\ \quad \mathit{get} \quad\ \ = i_1 \nabla \mathit{get}_f \\ \quad \mathit{put} \quad\ \ = ((\mathit{id} + \mathit{create}_f \circ i_2) \circ \pi_1 \nabla i_2 \circ \mathit{put}_f) \circ \mathit{distr} \\ \quad \mathit{create} = \mathit{id} + \mathit{create}_f \circ i_2 \end{array}$$

For the sum combinator we can have the following lifting into a lens:

$$\begin{array}{l}\forall f : C \trianglerighteq A,\ g : D \trianglerighteq B.\ f + g : C + D \trianglerighteq A + B \\ \quad \mathit{get} \quad\ \ = \mathit{get}_f + \mathit{get}_g \\ \quad \mathit{put} \quad\ \ = (\mathit{put}_f \nabla \mathit{create}_f \circ \pi_1 + \mathit{create}_g \circ \pi_1 \nabla \mathit{put}_g) \circ \mathit{dists} \\ \quad \mathit{create} = \mathit{create}_f + \mathit{create}_g \end{array}$$

where *dists* is the following distribution combinator over sums:

$$\begin{array}{ll}\mathit{dists} : (A + B) \times (C + D) \to (A \times C + A \times D) + (B \times C + B \times D) & \\ \mathit{dists} = (\mathit{distr} + \mathit{distr}) \circ \mathit{distl} & \textsc{Dists-Def}\end{array}$$

In the definition of *put*, *dists* is first used to span the four possible cases. If the abstract and concrete values match (cases $A \times C$ and $B \times D$), then put_f and put_g are applied as expected. Otherwise (cases $A \times D$ and $B \times C$), we ignore the "out of sync" concrete values and use $create_f$ and $create_g$ to generate concrete values of the correct type. The definition of *create* is trivial and is merely the sum of the *create* functions of f and g. This sum combinator is essentially the point-free homologous of the abstract conditional combinator *acond* from [15].

Actually, $+$ can be lifted into a well-behaved lens in many different ways. Another alternative is the following, for arbitrary functions $h : A \times D \to C$ and $i : B \times C \to D$, although the first definition gives more natural results in most cases and will be used by default:

$$\begin{array}{l}
\forall f : C \trianglerighteq A,\ g : D \trianglerighteq B.\ \ (f+g)^{h,i} : C + D \trianglerighteq A + B \\
\quad \begin{array}{ll}
get & = get_f + get_g \\
put & = (put_f + put_g) \circ (id \nabla (\pi_1 \triangle h) + (\pi_1 \triangle i) \nabla id) \circ dists \\
create & = create_f + create_g
\end{array}
\end{array}$$

As with projections, in order to lift $! : C \to 1$ into a lens, a default value must be provided to be returned by the *create* function. The definition is trivial:

$$\begin{array}{l}
\forall c \in C.\ !^c : C \trianglerighteq 1 \\
\quad \begin{array}{ll}
get & = \,! \\
put & = \pi_2 \\
create & = \underline{c}
\end{array}
\end{array}$$

Likewise to sum injections, the constant combinator $\underline{\cdot} : B \to (A \to B)$, that given a value $b \in B$ returns b for all input values, cannot be lifted into a well-behaved lens unless an invariant is imposed on the abstract type stating that all its values are equal to b. Again, there exist particular expressions in which this combinator forms a well-behaved lens, such as $\underline{b} \nabla f : A + C \trianglerighteq B$ or $f \nabla \underline{b} : C + A \trianglerighteq B$, where $f : C \trianglerighteq B$ is any other well-behaved lens.

3 Recursion Patterns as Lenses

In this section, we investigate recursive lenses over inductive data types. Most user defined data types can be defined as the fixpoint of a polynomial functor. Given a *base functor* F, the inductive type generated by its least fixpoint will be denoted by μF. A polynomial functor is either the identity functor *Id* (denoting recursive invocation), the constant functor $\underline{A}$ or the lifting of the sum $\oplus$ and product bifunctors $\otimes$. For example, for lists we have $[A] = \mu L_A$, where $L_A = \underline{1} \oplus \underline{A} \otimes Id$, and for naturals $Nat = \mu N$, where $N = \underline{1} \oplus Id$. Associated with each data type μF we also have two unique functions $in_F : F\ \mu F \to \mu F$ and $out_F : \mu F \to F\ \mu F$, that are each other's inverse. They allow us to encode and inspect values of the given type, respectively. The application of *out* to a type results on a one-level unfolding to a sum-of-products representation capable of being processed with point-free combinators.

Given a functor F and a function $f : A \to B$, the functor mapping $F\ f : F\ A \to F\ B$ is a function that preserves the functorial structure and modifies all the instances of the type argument A into instances of type B. It can be defined inductively on the functor F, such that the argument f is applied to the recursive occurrences inside the sums-of-products structure and constants are left unchanged:

$$\begin{array}{ll} F\ f : F\ A \to F\ B & \\ \mathit{Id}\ f & = f \\ \underline{T}\ f & = \mathit{id} \\ (F \otimes G)\ f & = F\ f \times G\ f \\ (F \oplus G)\ f & = F\ f + G\ f \end{array}$$

On the other side, a *natural transformation* η between functors F and G, denoted by $\eta : F \dot{\to} G$, is a function that transforms instances of F into instances of G while preserving the inner instances of the polymorphic type argument. It assigns to each type A an arrow $\eta_A : F\ A \to G\ A$ such that, for any function $f : A \to B$, the following naturality condition holds:

$$G\ f \circ \eta_A = \eta_B \circ F\ f \qquad \textsc{Nat-Swap}$$

Instead of defining lenses by general recursion, we resort to well-known recursion patterns, and use their powerful algebraic laws (see Appendix A) to prove that the resulting lenses are well-behaved. The most fundamental combinator is the fold or *catamorphism* that encodes the recursion pattern of iteration. Given an algebra $g : F\ A \to A$, the catamorphism $(\![g]\!)_F : \mu F \to A$ is the unique function that makes the hereunder diagram commute:

$$\begin{tikzcd} \mu F \arrow[r, "out_F"] \arrow[d, "(\![g]\!)_F"'] & F\ \mu F \arrow[d, "F\ (\![g]\!)_F"] \\ A & F\ A \arrow[l, "g"] \end{tikzcd}$$

A catamorphism recursively consumes a data type μF by replacing its constructors with the given algebra g. A well-known example is the $\mathit{length} : [A] \to \mathit{Nat}$ function presented in the introduction, that can be defined as the following catamorphism:

$$\mathit{length} = (\![\mathit{in}_N \circ (\mathit{id} + \pi_2)]\!)_{L_A}$$

Another example of a catamorphism is the function $\mathit{filter\_left} : [A + B] \to [A]$ that filters all the left alternatives from a list of optional elements:

$$\begin{array}{ll} \mathit{filter\_left} :: [\mathit{Either}\ a\ b] \to [a] & \\ \mathit{filter\_left}\ [\,] & = [\,] \\ \mathit{filter\_left}\ (\mathit{Left}\ x : xs) & = x : \mathit{filter\_left}\ xs \\ \mathit{filter\_left}\ (\mathit{Right}\ x : xs) & = \mathit{filter\_left}\ xs \end{array}$$

Using the basic isomorphisms presented before, it is not difficult to put together a point-free algebra with the intended behaviour:

$$\mathit{filter\_left} = (\!|(in_{L_A} \nabla \pi_2) \circ \mathit{coassocl} \circ (id + \mathit{distl})|\!)_{L_{A+B}}$$

The dual recursion pattern of catamorphism is the unfold or *anamorphism.* Given a coalgebra $h : A \to F\ A$, the anamorphism $[\!(h)\!]_F : A \to \nu F$ is a function that, given an element of A, builds a (possibly infinite) element of the coinductive datatype νF (the greatest fixpoint of F). The coalgebra h is used to decide when generation stops and, in case it proceeds, which "seeds" should be used to generate the recursive occurrences of νF. Here we will only be interested in a specific kind of unfolds, namely those that always terminate. Not only we want all our lenses to terminate, but we also want to be able to freely compose them with catamorphisms - this composition is not always well-defined because anamorphisms can generate infinite values that are not part of the least fixed point consumed by catamorphisms. If we restrict ourselves to *recursive* [8] (or *reductive* [1]) coalgebras, the resulting morphism (to be denoted by *recursive anamorphism*) is guaranteed to halt in finitely many steps. A recursive coalgebra $h : A \to F\ A$ is essentially one that guarantees that all As contained in the resulting $F\ A$ are somehow smaller than its input. Capretta *et al.* [8] give a nice formal definition and provide a set of constructions for building recursive coalgebras out of simpler ones. Since $out_F : \mu F \to F\ \mu F$ is a final recursive coalgebra, we can safely compose catamorphisms with recursive anamorphisms. Given a recursive coalgebra $h : A \to F\ A$, the recursive anamorphism $[\![h]\!]_F : A \to \mu F$ is the unique function that makes the hereunder diagram commute:

$$\begin{tikzcd}
\mu F & F\ \mu F \arrow[l, "in_F"'] \\
A \arrow[u, "{[\![h]\!]_F}"] \arrow[r, "h"'] & F\ A \arrow[u, "{F\ [\![h]\!]_F}"']
\end{tikzcd}$$

Given this uniqueness property, the recursive anamorphism obeys the same laws as the normal anamorphism, namely fusion. A trivial example of a recursive anamorphism to naturals is again the length function:

$$\mathit{length} = [\![(id + \pi_2) \circ out_{L_A}]\!]_N$$

Notice that $id + \pi_2 : L_A \stackrel{.}{\to} N$ and that a composition of a natural transformation with a recursive coalgebra is again a recursive coalgebra [8, Proposition 3.9], so this is clearly a recursive anamorphism. In fact, every catamorphism $(\!|in_G \circ \eta|\!)_F$, where $\eta : F \stackrel{.}{\to} G$ is a natural transformation can also be defined by a recursive anamorphism $[\![\eta \circ out_F]\!]_G$, and vice-versa. A classical example of a recursive anamorphism that cannot be defined using a catamorphism is the function $\mathit{zip} : [A] \times [B] \to [A \times B]$, that zips two lists together into a list of pairs:

$$\begin{array}{ll} \mathit{zip} :: ([a],[b]) \to [(a,b)] & \\ \mathit{zip}\ (x : \mathit{xs}, y : \mathit{ys}) & = (x,y) : \mathit{zip}\ (\mathit{xs},\mathit{ys}) \\ \mathit{zip}\ \_ & = [\,] \end{array}$$

In the point-free style it can be redefined as follows:

$$\mathit{zip} = [\![(! + \mathit{distp}) \circ \mathit{coassocl} \circ \mathit{dists} \circ (\mathit{out}_{L_A} \times \mathit{out}_{L_B})]\!]_{L_{A \times B}}$$

The coalgebra guarantees that the output list stops being generated when at least one of the inputs is empty. Otherwise, both tails are used as "seed" to recursively generate the tail of the output list.

The composition of a catamorphism after an anamorphism is known as *hylomorphism*, but as mentioned above, this composition is not always well-defined in Set. Here, we will be interested in hylomorphisms that are guaranteed to terminate, namely those where the cata is composed with a recursive anamorphism:

$$[\![g, h]\!]_F = (\![g]\!)_F \circ [\![h]\!]_F \qquad \text{HYLO-SPLIT}$$

These *recursive hylomorphisms* (the unique *coalgebra-to-algebra morphisms* of [8]) are quite amenable to program calculation because they enjoy a uniqueness law similar to the other recursion patterns:

$$[\![g, h]\!]_F = f \Leftrightarrow g \circ F\ f \circ h = f \qquad \text{HYLO-UNIQ}$$

3.1 Functor Mapping

We can lift functor mapping into a lens combinator by applying regular functor mapping to each component of a lens, as follows:

$$\begin{array}{ll} \forall f : C \trianglerighteq A.\ \ F\ f : F\ C \trianglerighteq F\ A & \\ \mathit{get} & = F\ \mathit{get}_f \\ \mathit{put} & = F\ \mathit{put}_f \circ \mathit{fzip}_F\ \mathit{create}_f \\ \mathit{create} & = F\ \mathit{create}_f \end{array}$$

The interesting snippet is the fzip_F combinator, responsible for zipping abstract and concrete instances of the same F-structure, and that is defined below:

$$\begin{array}{l} \mathit{fzip}_F : (A \to C) \to F\ A \times F\ C \to F\ (A \times C) \\ \mathit{fzip}_{Id}\ f = id \\ \mathit{fzip}_{\underline{T}}\ f = \pi_1 \\ \mathit{fzip}_{(F \otimes G)}\ f = (\mathit{fzip}_F\ f \times \mathit{fzip}_G\ f) \circ \mathit{distp} \\ \mathit{fzip}_{(F \oplus G)}\ f = (\mathit{fzip}_F\ f \nabla F\ (id \triangle f) \circ \pi_1 + G\ (id \triangle f) \circ \pi_1 \nabla \mathit{fzip}_G\ f) \circ \mathit{dists} \end{array}$$

As usual, *fzip* gives preference to the values from the abstract data type. In the case of sums (similarly to the definition of the + lens), *fzip* is applied recursively to the sub-functors F and G, and whenever the abstract and concrete values are "out of sync", the abstract value is preserved and a new concrete value is created from the abstract value, by invoking the argument function.

We can polytypically prove (in the style of [18]) the following laws about *fzip*:

$$F\ \pi_1 \circ \mathit{fzip}_F\ f = \pi_1 \qquad \textsc{Fzip-Cancel}$$
$$\mathit{fzip}_F\ f \circ (F\ g \bigtriangleup F\ h) = F\ (g \bigtriangleup h) \qquad \textsc{Fzip-Split}$$

The first states that fzip_F cannot modify the shape of the abstract type, nor the data contained in it. The second states that zipping two "in sync" values can be trivially done just by mapping. The proof of the first property can be found in Appendix B. The other proof is similar and is omitted.

3.2 Anamorphism

At this point, we have enough ingredients to "lensify" anamorphisms. For the resulting lens to be well-behaved, the coalgebra must be recursive and itself a well-behaved lens. The generic definition is as follows:

$$\begin{array}{l}
\forall f : A \trianglerighteq G\ A.\ [\![f]\!]_G : A \trianglerighteq \mu G \\
\quad \mathit{get} \quad\ = [\![\mathit{get}_f]\!]_G \\
\quad \mathit{put} \quad\ = [\![\mathit{put}_f, \mathit{fzip}_G\ \mathit{create} \circ (\mathit{out}_G \times \mathit{get}_f) \bigtriangleup \pi_2]\!]_{G \otimes \underline{A}} \\
\quad \mathit{create} = (\![\mathit{create}_f]\!)_G
\end{array}$$

Knowing that create_f is an algebra with type $G\ A \rightarrow A$, *create* is trivially defined using a catamorphism. The generic definition of *put* uses an accumulation technique implemented as a recursive hylomorphism: it proceeds inductively over the abstract value, using the concrete value as an accumulator. The function that propagates the accumulator to recursive calls is $fzip_G\ \mathit{create} \circ (\mathit{out}_G \times \mathit{get}_f)$. The diagram for this hylomorphism is the following:

$$\begin{tikzcd}
\mu G \times A \arrow[r, "out_G \times get_f"] \arrow[rr, bend right, "\pi_2"', "\bigtriangleup"] \arrow[d, "put"'] & G\ \mu G \times G\ A \arrow[r, "fzip_G\ create"] & G\ (\mu G \times A) \times A \arrow[d, "G\ put \times id"] \\
A & & G\ A \times A \arrow[ll, "put_f"]
\end{tikzcd}$$

The proof that this lens is well-behaved is given in Appendix B. The proof of laws CreateGet and PutGet can be done using the fusion law for anamorphisms. The proof of law GetPut uses hylomorphism fusion and Hylo-Uniq.

By applying this definition to the *zip* function, we get the expected definitions for *create* and *put*. For better understanding, we present them using Haskell syntax and explicit recursion (easily derivable from the original point-free definition):

$$\begin{array}{l}
\mathit{create} :: [(a, b)] \rightarrow ([a], [b]) \\
\mathit{create}\ [\,] \qquad\qquad = ([\,], [\,]) \\
\mathit{create}\ ((x, y) : t) = \mathbf{let}\ (\mathit{xs}, \mathit{ys}) = \mathit{create}\ t\ \mathbf{in}\ (x : \mathit{xs}, y : \mathit{ys})
\end{array}$$

```
put :: ([(a, b)], ([a], [b])) → ([a], [b])
put ([], ([], r))             = ([], r)
put ([], (l, []))             = (l, [])
put ((x, y) : t, (_ : l, _ : r)) = let (xs, ys) = put (t, (l, r)) in (x : xs, y : ys)
put (l, _)                    = create l
```

The *create* induced by this lens is just the *unzip* function, a fold that recursively splits a list of pairs into two lists. The *put* has a more intricate behaviour: it only recovers elements of one of the original concrete lists when the updated abstract list is smaller than it but with the exact same length of the other concrete list. This guarantees that zipping the result again yields the same view. For example, *put* $([(1,2),(3,4)],([4,5],[6,7,8,9]))$ returns $([1,3],[2,4,8,9])$. Notice how the elements 8 and 9 of the bigger list are recovered.

3.3 Catamorphism

Catamorphisms can be lifted to well-behaved lenses as follows:

$$\begin{aligned}
&\forall f : F\ A \trianglerighteq A.\ \ (\!| f |\!)_F : \mu F \trianglerighteq A \\
&\quad get \;\;\;\, = (\!| get_f |\!)_F \\
&\quad put \;\;\;\, = [\![fzip_F\ create \circ (put_f \circ (id \times G\ get) \bigtriangleup \pi_2) \circ (id \times out_F)]\!]_F \\
&\quad create = [\![create_f]\!]_F
\end{aligned}$$

Notice how *put*, encoded as an anamorphism, still gives preference to abstract values by using *fzip*, as depicted in the following diagram:

$$\begin{tikzcd}
\mu F & & & & F\ \mu F \arrow[llll, "in_F"'] \\
A \times \mu F \arrow[u, "put"] \arrow[r, "id \times out_F"'] & A \times F\ \mu F \arrow[r, "id \times F\ get"'] \arrow[rr, bend left, dashed, "\pi_2"] & A \times F\ A \arrow[r, "put_f"'] \arrow[r, phantom, shift left=4, "\bigtriangleup"] & F\ A \times F\ \mu F \arrow[r, "fzip_F\ create"'] & F\,(A \times \mu F) \arrow[u, "F\ put"']
\end{tikzcd}$$

To prove that $(\!| f |\!)$ is a well-behaved lens, we must prove that both *put* and *create* are recursive anamorphisms. Given these conditions, the proof is very similar to the proof for anamorphisms and is omitted.

The *filter_left* function, using the left-biased version of the either combinator, is an example of a fold lens where it is not difficult to prove that both the *create* and *put* are indeed recursive. Its corresponding *create* is a simple unfold that maps a list into a list of left alternatives:

```
create :: [a] → [Either a b]
create []       = []
create (x : xs) = Left x : create xs
```

The *put* function restores right alternatives from the original concrete list:

$$\begin{array}{ll} put :: ([a], [Either\ a\ b]) \to [Either\ a\ b] & \\ put\ (xs, Right\ y : ys) & = Right\ y : put\ (xs, ys) \\ put\ ([\,], \_) & = [\,] \\ put\ (x : xs, [\,]) & = Left\ x : put\ (xs, create\ xs) \\ put\ (x : xs, Left\ y : ys) & = Left\ x : put\ (xs, ys) \end{array}$$

For an example of a catamorphism lens whose *create* function is not recursive, just replace the left-biased either combinator by the right-biased version in the point-free definition presented above. This change yields the following reshaping of *create*, assuming b to be the default constant that parameterizes π_2:

$$\begin{array}{l} create :: [a] \to [Either\ a\ b] \\ create\ xs = Right\ b : create\ xs \end{array}$$

3.4 Natural Transformations

A special case of the previous lenses occurs when the forward transformation is both expressible as a catamorphism and an anamorphism with the same natural transformation in the recursive gene. We name a lens f describing a bidirectional natural transformation between functors F and G a *natural lens* and type it with the signature $f : F \stackrel{\cdot}{\trianglerighteq} G$, where $get : F \stackrel{\cdot}{\to} G$, $put : G \otimes F \stackrel{\cdot}{\to} F$ and $create : G \stackrel{\cdot}{\to} F$. Unlike the previous cases, where we still have to check that the coalgebras are recursive, given a natural lens $\eta : F \stackrel{\cdot}{\trianglerighteq} G$, both $(\!|in_G \circ \eta|\!)_F$ and $[\![\eta \circ out_F]\!]_G$ immediately determine well-behaved lenses between μF and μG because, as mentioned before, termination is guaranteed.

There are several examples of these lenses. As seen before, the *length* function is a well-known example that can be expressed either as a catamorphism on lists or an anamorphism to naturals. Instantiating the input type to lists of naturals and the default constant that parameterizes π_2 to *Zero*, the forward and backward functions induced by this lens are exactly the same as the ones presented in the introduction. Another lens that establishes a natural transformation between base functors is mapping over lists:

$$\forall f : C \trianglerighteq A.\ \ map\ f = (\!|in_{L_A} \circ (id + f \times id)|\!)_{L_C} : [C] \trianglerighteq [A]$$

This definition can be generalized for any parametric type D defined inductively over a bifunctor B:

$$\forall f : C \trianglerighteq A.\ \ gmap\ f = (\!|in_{B\ A} \circ B\ f\ id|\!)_{B\ C} : D\ C \trianglerighteq D\ A$$

3.5 Hylomorphisms

It is well known that most recursive functions can be encoded using hylomorphisms over polynomial functors. Given that HYLO-SPLIT allows us to factorize a hylomorphism into the composition of a catamorphism after an anamorphism, the range of recursive functions that we can lift to well-behaved lenses is considerably enlarged. Of course the algebras and coalgebras of the hylomorphism must themselves be lenses (for example, built using the combinators presented in Section 2), and the coalgebras must be recursive.

Take as an example the natural number addition function $plus : Nat \times Nat \rightarrow Nat$:

$$\begin{aligned}
&plus :: (Nat, Nat) \rightarrow Nat\\
&plus\ (Zero, m) \quad = m\\
&plus\ (Succ\ n, m) = Succ\ (plus\ (n, m))
\end{aligned}$$

Although it is not a fold neither an unfold, it can be defined as the following hylomorphism, where both the algebra and the recursive coalgebra are lenses:

$$plus = [\![in_N \circ (out_N \nabla i_2), (\pi_2 + id) \circ distl \circ (out_N \times id)]\!]_{\underline{Nat} \oplus Id}$$

In order to lift this function into a well-behaved lens, the *create* and *put* functions should guarantee that the sum of the generated pair of numbers equals the abstract value. The *create* automatically induced by the techniques presented above simply creates a pair with the abstract value and a *Zero* as the second element:

$$\begin{aligned}
&create :: Nat \rightarrow (Nat, Nat)\\
&create\ n = (n, Zero)
\end{aligned}$$

As usual, the induced *put* function is a bit more tricky: if the abstract value is greater than the first element of the concrete pair, that element is preserved and the second element becomes the difference between both; if the abstract value is smaller it just pairs, it with zero likewise to *create*:

$$\begin{aligned}
&put :: (Nat, (Nat, Nat)) \rightarrow (Nat, Nat)\\
&put\ (Zero, \_) &&= (Zero, Zero)\\
&put\ (n, (Zero, o)) &&= \mathbf{let}\ (a, b) = create\ n\ \mathbf{in}\ (b, a)\\
&put\ (Succ\ n, (Succ\ m, o)) &&= \mathbf{let}\ (a, b) = put\ (n, (m, o))\ \mathbf{in}\ (Succ\ a, b)
\end{aligned}$$

4 Related Work

Our point-free framework can be seen as a domain-specific language similar to the language for lenses over trees first developed by Foster *et al.* [15]. While our generic combinators rely on the sum-of-products representation of inductive types, they represent all recursive types as generalized trees. They also devise a complex set-based type system with invariants to precisely define the domains for which their combinators are well-behaved. Using such *semantic* types [16], they are able to define well-behaved lenses like $\underline{c}$, for an arbitrary constant c. We rely in a *syntactic* and more standard (and decidable) type system that is implemented in functional programming languages like Haskell or ML, with the counterpart that we have a more limited set of well-behaved lenses. Additionally, we identify precise termination conditions to verify in order to guarantee that recursive lenses like folds and unfolds constitute well-behaved lenses. We believe that using the techniques presented in [1,8] these conditions are easier to verify than the conditions stated in [15] concerning general recursion.

On a more algebraic tone, researchers from the University of Tokyo have studied the automatic inversion of forward transformations defined in a point-free language of injective functions, to be used in a structure XML editor supporting views [24]. The idea is to move the burden of information preservation from *put* to *get* as to make $put : A \to C$ stateless and $get : C \to A$ injective. In practice this setting resembles that of data refinement, as attested by the required $put \circ get = id$ property. In order to deal with duplication and structural changes, editing tags are introduced in the domain of *put*. In the case of data duplication, if only one element of the resulting pair is updated (and thus marked with an edit tag), a view-to-view roundtrip is then able to propagate the modification to the other element and restore the invariant in the view. In addition, a weaker version of PutGet, baptized PutGetPut ($put \circ get \circ put \sqsubseteq put$), is required, to ensure that, when editing a view, applying *put* to update the source and computing the new view with *get* is sufficient to synchronize all the changes. The preorder $\sqsubseteq$ reflects the partiality of *put*, given that the domain of *get* may be larger than the range of *put*.

A follow-up work approached the automatic derivation of backward transformations based on a notion of view-update under *constant complement* [20]. Instead of assuming forward injective functions, they now take any $get : C \to A$ and derive an explicit complement function $get^c : C \to H$, such that the tupled function $get \vartriangle get^c : C \to A \times H$ is injective. The *put* function is then calculated from the specification $(get \vartriangle get^c)^{-1} \circ (id \times get^c)$. The bidirectional properties follow those of *closed* view-updating [3], where the source is hidden from the users when the view is updated. Besides the fundamental stability condition, there are undoability and composability conditions that, as shown by Diskin [14], yield precisely the *very well-behaved lenses* first presented in [15]. However, these additional properties are defined modulo partiality of *put* since, according to the constant complement approach, *put* should forbid any changes to the information that the complement has kept. For instance, inserting and removing elements are forbidden updates in their running example of a filtering lens.

To avoid restricting the syntax of the forward transformations, Voigtländer allows normal Haskell functions to be used in lens definitions [27]. In this scenario, reasonable backward transformations can be derived by observing the runtime behaviour of the forward transformations. A higher-order bidirectionalizer *bff* is defined that receives a polymorphic *get* function and returns the corresponding *put* function, ensuring bidirectional properties similar to [20]. Although, for example, the mapping lens is not definable in this framework, there are some lenses supported by *bff* that are not expressible with our combinators, namely polymorphic functions that duplicate information. However, likewise to [20], it is debatable how much bidirectionalization is truly achieved, concerning the degree of partiality of the backward transformation. For example, in this framework the *put* function of the *length* lens would not try to synchronize updates that change the shape of the abstract view, and thus would only be defined for the cases when the length of the original list remains constant.

Wang *et al.* [28] propose a language of right-invertible point-free combinators denoting total transformations, in order to define a *view* mechanism on datatypes that enables sound equational reasoning at the view level. However, they only consider pure abstractions (i.e, without a *put* function that takes into account old concrete values), and the chosen right-inverses of most of their combinators essentially coincide with the *create* functions of our lens combinators. Since their language also includes non-surjective datatype constructors as primitives, an additional compile-time check is required to test the joint surjectivity of programs involving constructors. The inclusion of a fold combinator also raises concerns regarding the termination of anamorphisms as right-inverses, and, likewise to our approach, additional constraints on the coalgebras must be checked.

In previous work, we have proposed a two-level bidirectional transformation framework (2LT) for *data refinement* [9,4], where forwards and backwards transformations were also specified in the point-free style, and type-safeness of the value migration functions was ensured with a deep embedding in Haskell. Later, we have shown how point-free program calculation could be used for the optimization of large compositions of bidirectional transformations and structure-shy query migration from the source to the target types [12,13]. In this paper, we tackle the dual problem of abstraction, using similar techniques to define generic point-free lenses: we intend to incorporate them into the 2LT framework, in order to enlarge the scope of model transformation scenarios to which it can be applied, and benefit from the optimization strategies previously implemented.

5 Conclusion

In this paper we have shown how to lift most of the standard point-free combinators and recursion patterns to well-behaved lenses. This enables the definition of elegant, generic, and, hopefully, intuitive lenses over inductive data types. Concerning recursion, we have identified precise termination conditions that allow folds and unfolds to be lifted to well-behaved lenses. Notice that we can also tackle arbitrary recursive lenses by expressing them as hylomorphisms, i.e the composition of a fold after an unfold. Using the techniques described in [10], we have also implemented a Haskell library, with the combinators presented in this paper and some more, to aid the construction of functional bidirectional programs by composition. The library is extensible, by supporting user-defined lens combinators, and is available through the Hackage package repository (`http://hackage.haskell.org`) under the name `pointless-lenses`, honoring a common joke about point-free programming.

Complex lens transformations suffer from degraded performance due to the cluttering of intermediate structures originated from the combination of smaller transformations. In the short run, we intend to apply the techniques developed for point-free refinement optimization [12,13] to the optimization of complex lenses defined by composition. Oliveira [25] already showed that a relational point-free calculus can be a more natural setting to formalize bidirectional transformations. This relational calculus can provide several advantages, such as reasoning about termination, computing inverses of arbitrary transformations, and expressing structural invariants over data-types. The latter is of utmost importance to statically calculate the domain on which a put function is well-defined, thus widening the set of potential well-behaved lenses to combinators like split or the injections.

Acknowledgments

We would like to thank the anonymous referees for the many insightful comments, and José Bacelar Almeida for the helpful discussions concerning termination.

References

1. Backhouse, R., Doornbos, H.: Mathematics of recursive program construction. Manuscript (2001), `http://www.cs.nott.ac.uk/rcb/MPC/papers`
2. Backus, J.: Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. Communications of the ACM 21(8), 613–641 (1978)
3. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Transactions on Database Systems 6(4), 557–575 (1981)
4. Berdaguer, P., Cunha, A., Pacheco, H., Visser, J.: Coupled schema transformation and data: Conversion for XML and SQL. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 290–304. Springer, Heidelberg (2006)
5. Bird, R., de Moor, O.: The Algebra of Programming. Prentice-Hall, Englewood Cliffs (1997)
6. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08), pp. 407–419. ACM, New York (2008)
7. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: a language for updatable views. In: Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'06), pp. 338–347. ACM, New York (2006)
8. Capretta, V., Uustalu, T., Vene, V.: Recursive coalgebras from comonads. Information and Computation 204(4), 437–468 (2006)
9. Cunha, A., Oliveira, J.N., Visser, J.: Type-safe two-level data transformation. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 284–299. Springer, Heidelberg (2006)
10. Cunha, A., Pacheco, H.: Algebraic specialization of generic functions for recursive types. In: Proceedings of the 2nd Workshop on Mathematically Structured Functional Programming (MSFP'08). ENTCS, Elsevier Science Publishers B. V., Amsterdam (2008)

11. Cunha, A., Pinto, J.S., Proença, J.: A framework for point-free program transformation. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 1–18. Springer, Heidelberg (2006)
12. Cunha, A., Visser, J.: Strongly typed rewriting for coupled software transformation. Electronic Notes in Theoretical Computer Science 174(1), 17–34 (2007)
13. Cunha, A., Visser, J.: Transformation of structure-shy programs with application to XPath queries and strategic functions. Science of Computer Programming (to appear, 2010)
14. Diskin, Z.: Algebraic Models for Bidirectional Model Synchronization. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 21–36. Springer, Heidelberg (2008)
15. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Transactions on Programming Languages and Systems 29(3), 17 (2007)
16. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. Journal of the ACM 55(4), 1–64 (2008)
17. Gibbons, J.: Calculating functional programs. In: Blackhouse, R., Crole, R.L., Gibbons, J. (eds.) Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. LNCS, vol. 2297, pp. 149–203. Springer, Heidelberg (2002)
18. Hinze, R.: Generic programs and proofs. Bonn University, Habilitation (2000)
19. Hu, Z., Mu, S.-C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. Higher Order and Symbolic Computation 21(1-2), 89–118 (2008)
20. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07), pp. 47–58. ACM, New York (2007)
21. Meertens, L.: Designing constraint maintainers for user interaction (1998), Manuscript available at, `http://www.kestrel.edu/home/people/meertens`
22. Meijer, E., Fokkinga, 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)
23. Morgan, C., Gardiner, P.H.B.: Data refinement by calculation. Acta Informatica 27(6), 481–503 (1990)
24. Mu, S.-C., Hu, Z., Takeichi, M.: An algebraic approach to bi-directional updating. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 2–20. Springer, Heidelberg (2004)
25. Oliveira, J.N.: Data transformation by calculation. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering II. LNCS, vol. 5235, pp. 134–195. Springer, Heidelberg (2008)
26. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
27. Voigtländer, J.: Bidirectionalization for free! (Pearl). In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09), pp. 165–176. ACM, New York (2009)
28. Wang, M., Gibbons, J., Matsuda, K., Hu, Z.: Translucent Abstraction: Safe Views through Invertible Programming. Draft (2010), `http://web.comlab.ox.ac.uk/files/2280/total.pdf`

A Point-free Laws

$f \circ id = f \wedge id \circ f = f$	Id-Nat
$f \circ (g \circ h) = (f \circ g) \circ h$	$\circ$-Assoc
$f \circ h = g \circ h \Leftarrow f = g$	Leibniz
$\pi_1 \vartriangle \pi_2 = id$	$\times$-Reflex
$\pi_1 \circ (f \vartriangle g) = f \wedge \pi_2 \circ (f \vartriangle g) = g$	$\times$-Cancel
$(f \vartriangle g) \circ h = f \circ h \vartriangle g \circ h$	$\times$-Fusion
$(f \times g) \circ (h \vartriangle i) = f \circ h \vartriangle g \circ i$	$\times$-Absor
$(\pi_1 \nabla \pi_1) \circ distr = \pi_1$	Distr-Fst
$(\pi_1 + \pi_1) \circ distl = \pi_1$	Distl-Fst
$(f \times g) \circ (h \times i) = f \circ h \times g \circ i$	$\times$-Functor-Comp
$f \circ (g \nabla h) = f \circ g \nabla f \circ h$	$+$-Absor
$(f + g) \circ (h + i) = f \circ h + g \circ i$	$+$-Functor-Comp
$in_F \circ out_F = id_{\mu F} \wedge out_F \circ in_F = id_{F\ \mu F}$	In-Out-Iso
$F\ f \circ F\ g = F\ (f \circ g)$	Functor-Comp
$F\ id = id$	Functor-Id
$(\!\lvert g \rvert\!)_F \circ in_F = g \circ F\ (\!\lvert g \rvert\!)_F$	Cata-Cancel
$f \circ (\!\lvert g \rvert\!)_F = (\!\lvert h \rvert\!)_F \Leftarrow f \circ g = h \circ F\ f$	Cata-Fusion
$[\![out_F]\!]_F = id$	Ana-Reflex
$out_F \circ [\![h]\!]_F = F\ [\![h]\!]_F \circ h$	Ana-Cancel
$[\![g]\!]_F \circ f = [\![h]\!]_{\mu F} \Leftarrow g \circ f = F\ f \circ h$	Ana-Fusion
$[\![g, h]\!]_F = g \circ F\ [\![g, h]\!]_F \circ h$	Hylo-Cancel
$g \circ [\![h, i]\!]_F \circ j = [\![k, l]\!]_F \Leftarrow g \circ h = k \circ F\ g\ \wedge\ i \circ j = F\ j \circ l$	Hylo-Fusion

B Proofs

Proof ($[\![f]\!]_G$ is a well-behaved lens).

$get \circ create = id$
$\Leftrightarrow$ {definition of *get*; Ana-Reflex}
$[\![get_f]\!]_G \circ create = [\![out_G]\!]_G$
$\Leftarrow$ {Ana-Fusion}
$get_f \circ create = G\ create \circ out_G$
$\Leftrightarrow$ {definition of *create*; Cata-Cancel}
$get_f \circ create_f \circ G\ create \circ out_G = G\ create \circ out_G$
$\Leftarrow$ {Leibniz}
$get_f \circ create_f = id$
$\Leftrightarrow$ {CreateGet}
TRUE

$get \circ put$
$=$ {definition of *get*}

$[\![get_f]\!]_G \circ put$
$= \{$ANA-FUSION; definition of $put\}$
$\quad get_f \circ [\![put_f, fzip_G\ create \circ (out_G \times get_f) \triangle \pi_2]\!]_{G \otimes \underline{A}} = G\ put \circ h$
$\quad \Leftrightarrow \{$HYLO-CANCEL$\}$
$\quad get_f \circ put_f \circ (G \otimes \underline{A})\ put \circ (fzip_G\ create \circ (out_G \times get_f) \triangle \pi_2) = G\ put \circ h$
$\quad \Leftrightarrow \{$PUTGET; $\times$-FUNCTOR-COMP$\}$
$\quad \pi_1 \circ (G\ put \circ fzip_G\ create \circ (out_G \times get_f) \triangle \pi_2)$
$\quad \Leftrightarrow \{\times$-CANCEL$\}$
$\quad G\ put \circ fzip_G\ create \circ (out_G \times get_f) = G\ put \circ fzip_G\ create \circ (out_G \times get_f)$
$[\![fzip_G\ create \circ (out_G \times get_f)]\!]_G$
$= \{$ANA-FUSION$\}$
$\quad G\ \pi_1 \circ fzip_G\ create \circ (out_G \times get_f) = out_G \circ \pi_1$
$\quad \Leftrightarrow \{$FZIP-CANCEL; $\times$-CANCEL$\}$
$\quad out_G \circ \pi_1 = out_G \circ \pi_1$
$[\![out_G]\!]_G \circ \pi_1$
$= \{$ANA-REFLEX$\}$
π_1

$put \circ (get \triangle id)$
$= \{$definition of $put\}$
$[\![put_f, (fzip_G\ create \circ (id \times get_f) \triangle \pi_2) \circ (out_G \times id)]\!]_{G \otimes \underline{A}} \circ (get \triangle id)$
$= \{$HYLO-FUSION$\}$
$\quad (fzip_G\ create \circ (id \times get_f) \triangle \pi_2) \circ (out_G \times id) \circ (get \triangle id)$
$\quad = (G \otimes \underline{A})\ (get \triangle id) \circ (get_f \triangle id)$
$\quad \Leftrightarrow \{\times$-ABSOR; $\times$-ABSOR; $\times$-CANCEL$\}$
$\quad (fzip_G\ create \circ (out_G \circ get \triangle get_f) \triangle id)$
$\quad = (G\ (get \triangle id) \times id) \circ (get_f \triangle id)$
$\quad \Leftrightarrow \{$ANA-CANCEL; $\times$-FUSION$\}$
$\quad (fzip_G\ create \circ (G\ get \triangle id) \circ get_f \triangle id)$
$\quad = (G\ (get \triangle id) \times id) \circ (get_f \triangle id)$
$\quad \Leftrightarrow \{$FZIP-SPLIT; $\times$-ABSOR$\}$
$\quad (G\ (get \triangle id) \circ get \triangle id) = (G\ (get \triangle id) \circ get_f \triangle id)$
$[\![put_f, get_f \triangle id]\!]_{G \otimes \underline{A}} = id$
$= \{$HYLO-UNIQ; GETPUT$\}$
id

*Proof (*FZIP-CANCEL*).*

$Id\ \pi_1 \circ fzip_{Id}\ f$
$= \{$FZIP-DEF; ID-NAT$\}$
π_1

$\underline{T}\ \pi_1 \circ fzip_{\underline{T}}\ f$
$= \{$FZIP-DEF; ID-NAT$\}$
π_1

$(F \otimes G)\ \pi_1 \circ fzip_{F \otimes G}\ f$
$= \{$FZIP-DEF$\}$

$(F\ \pi_1 \times G\ \pi_1) \circ (\mathit{fzip}_F\ f \times \mathit{fzip}_G\ f) \circ \mathit{distp}$
$= \{ \times\text{-Functor-Comp; Fzip-Cancel; Fzip-Cancel} \}$
$(\pi_1 \times \pi_1) \circ \mathit{distp}$
$= \{ \text{Distp-Def; } \times\text{-Absor; } \times\text{-Cancel; } \times\text{-Cancel} \}$
$\pi_1 \circ \pi_1 \bigtriangleup \pi_2 \circ \pi_1$
$= \{ \times\text{-Fusion; } \times\text{-Reflex; Id-Nat} \}$
π_1

$(F \oplus G)\ \pi_1 \circ \mathit{fzip}_{F \oplus G}\ f$
$= \{ \text{Fzip-Def} \}$
$(F\ \pi_1 + G\ \pi_1) \circ (\mathit{fzip}_F\ f \bigtriangledown F\ (\mathit{id} \bigtriangleup f) \circ \pi_1 + G\ (\mathit{id} \bigtriangleup f) \circ \pi_1 \bigtriangledown \mathit{fzip}_G\ f)$
$\circ\ \mathit{dists}$
$= \{ +\text{-Functor-Comp; } +\text{-Absor; } +\text{-Absor} \}$
$(F\ \pi_1 \circ \mathit{fzip}_F\ f \bigtriangledown F\ \pi_1 \circ F\ (\mathit{id} \bigtriangleup f) \circ \pi_1 + G\ \pi_1 \circ G\ (\mathit{id} \bigtriangleup f) \circ \pi_1 \bigtriangledown G\ \pi_1$
$\circ\ \mathit{fzip}_G) \circ \mathit{dists}$
$= \{ \text{Fzip-Cancel; Fzip-Cancel; Functor-Comp; Functor-Comp} \}$
$(\pi_1 \bigtriangledown F\ (\pi_1 \circ (\mathit{id} \bigtriangleup f)) \circ \pi_1 + G\ (\pi_1 \circ (\mathit{id} \bigtriangleup f)) \circ \pi_1 \bigtriangledown \pi_1) \circ \mathit{dists}$
$= \{ \text{Functor-Id; Functor-Id} \}$
$((\pi_1 \bigtriangledown \pi_1) + (\pi_1 \bigtriangledown \pi_1)) \circ \mathit{dists}$
$= \{ \text{Dists-Def; Distr-Fst; Distr-Fst} \}$
$(\pi_1 + \pi_1) \circ \mathit{distl}$
$= \{ \text{Distl-Fst} \}$
π_1

Formal Derivation of Concurrent Garbage Collectors

Dusko Pavlovic[1], Peter Pepper[2], and Douglas R. Smith[1]

[1] Kestrel Institute, Palo Alto, California
{dusko,smith}@kestrel.edu
[2] Technische Universität Berlin and Fraunhofer FIRST, Berlin
pepper@cs.tu-berlin.de

Abstract. Concurrent garbage collectors are notoriously difficult to implement correctly. Previous approaches to the issue of producing correct collectors have mainly been based on posit-and-prove verification or on the application of domain-specific templates and transformations. We show how to derive the upper reaches of a family of concurrent garbage collectors by refinement from a formal specification, emphasizing the application of domain-independent design theories and transformations. A key contribution is an extension to the classical lattice-theoretic fixpoint theorems to account for the dynamics of concurrent mutation and collection.

1 Introduction

Concurrent collectors are extremely complex and error-prone. Since such collectors now form part of the trusted computing base of a large portion of the world's mission-critical software infrastructure, such unreliability is unacceptable [21]. The challenge has been to find a way to provide mathematical assurance of the correctness of concurrent collectors without doing harm to the productivity of the programmers. The latter aspect still is a major obstacle in verification-oriented systems. Interactive theorem provers may need thousands of lines of proof scripts or hundreds of lemmas in order to cope with serious collectors (see e.g. [10,15,4]). But also fully automated verifiers exhibit problems. As can be seen in [6], even the verification of a simplified collector necessitates such a large amount of complex properties that the specification may easily become faulty itself. The problem of assurance also has to deal with the fact that garbage collectors come in many variations, each addressing specific quality or efficiency goals. Separate verification of each variation leads to a tremendous duplication of work. On the other hand it is extremely difficult to determine for a slightly modified algorithm, which properties and proofs can remain unchanged, which are superfluous, and which need to be added or redone.

We propose to apply the approach of *specification refinement* as illustrated in Figures 1 and 2. This approach has already been successfully applied to complex problems, such as planning and scheduling tasks [19]. Figure 1 describes the way in which we come from abstract problems to concrete solutions.

(1) Suppose we have an *abstract problem description*, that is, a collection of types, operations and properties that together describe a certain problem.
(2) For this abstract problem we then develop an *abstract solution*, that is, an abstract implementation that fulfills all the requested properties.
(3) When we now have a *concrete problem* that is an instance of our abstract problem (since it meets all its properties), then we can
(4) automatically derive a *concrete solution* by instantiating the abstract solution correspondingly.
The abstract problem/solution pairs can be organized into a taxonomic library [17] in formal development environments such as KIDS [16] and Specware [7]. We will consider the abstract problem of finding fixed points in lattices or cpos and several solutions for this problem. Then we will show that garbage collection is an instance of this abstract problem by considering the concrete graphs and sets as instances of the more abstract lattices. This way our abstract solutions carry over to concrete solutions for the garbage collection problem.

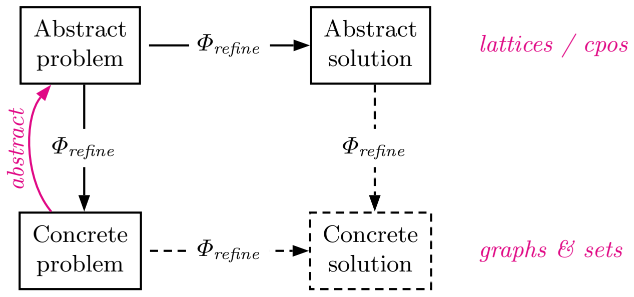


Fig. 1. Abstract and concrete problems and their solutions

Technically, all of our problem and solution descriptions are algebraic and coalgebraic specifications, which are usually underspecified and thus possess many models. "Solutions" are treated as borderline cases of such specifications, which are directly translatable into code of some given programming language. The formal connections between the various specifications are given by certain kinds of refinement morphisms, and the derivation of the concrete solution from the other parts is formally a pushout construction from category theory[1].

Figure 2 illustrates the second essential aspect of our method. We work with a family tree (or dag) of more and more refined problems, each giving rise to more and more refined solutions. On the problem side "refined" essentially means that we have additional properties, on the solution side "refined" essentially means that we have better algorithms, e.g. more efficient, more robust, more concurrent etc.

[1] A morphism Φ from specification S to specification T is given by a type-consistent mapping of the type, function, and predicate symbols of S to derived types, functions, and predicates in T. The mapping is a specification morphism if the axioms of S translate to theorems of T. A pushout construction is used to compose specifications. More detail on the category of specifications may be found in [18,7,12].

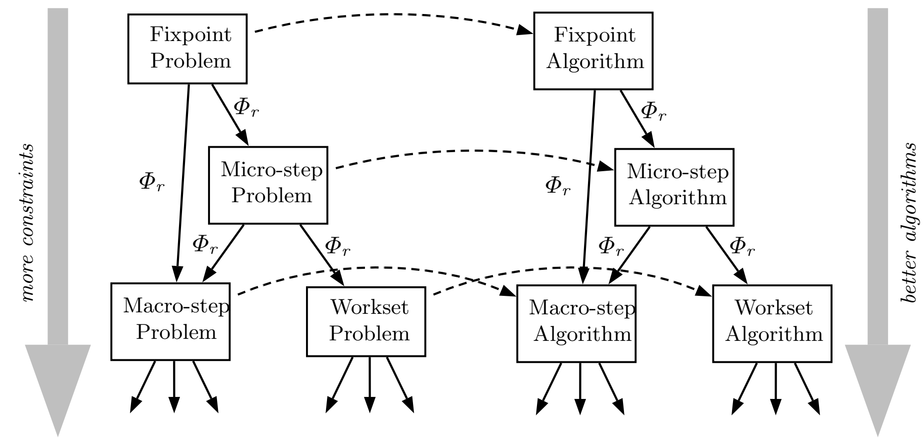


Fig. 2. Refinement of problems (and solutions)

This way of proceeding has the primary advantage that it allows us to reuse verification and development efforts. Suppose that at some point in the tree we want to design a new variation. This is reflected in a new refinement child of the current specification, to which certain properties are added. In the modified new solution we need only prove those properties that have been added; everything else is inherited.

Vechev et al. [21] have a similar goal of presenting a derivational treatment of a family of concurrent garbage collectors. They start from a generic algorithm, which is parameterized by an underspecified function, such that different instantiations of this function lead to different collection algorithms. A primary concern of [21] is the possibility to combine various “design dimensions” in a very flexible way. By contrast, we study the family tree of specifications and implementations that can be systematically derived using formal refinements that are largely problem-independent. We base the whole treatment of garbage collection on fundamental mathematical principles, namely lattices and fixed points. This means that the same design theories can be applied to a wide range of other problems. A key result of our studies was a generalization of classical fixed-point results of lattice theory to handle the dynamics of concurrency; i.e. iterating to a fixpoint with a monotone function that is changing over time.

The final efficiency of most practical garbage collection algorithms depends on the use of clever data representations. Standard techniques range from the classical stacks or queues to bit maps, overlayed pointers, so-called dirty bits, color toggling, concurrent local structures, and so forth. In our approach all these designs fall under the paradigm of data reification morphisms. This means that we can work throughout our developments with high-level abstract data structures such as graphs and sets in order to specify and verify the algorithmic aspects in the clearest possible way. It will be only at the end of the derivation that the high-level data structures are implemented by concrete data structures, which are chosen based on their efficiency in the given context. This step is automated in systems like Specware [7], together with many low-level optimizations. Since this is very technical and can be done almost automatically by advanced systems, we will only touch this part very briefly and sketchy here.

Full derivation of practical algorithms requires many more transformations than we can show here. We focus on the crucial initial refinement steps from a formal specification of concurrent garbage collection toward a variety of important and practical algorithms. An expanded version of this paper treats more aspects of contemporary concurrent garbage collectors [13].

2 Notes on Garbage Collection

The very first garbage collectors, which essentially go back to McCarthy's original design [9], were *stop-the-world* collectors. That is, the Mutator was completely laid to sleep, while the Collector did its recycling. This approach leads to potentially very long pauses, which are nowadays considered to be unacceptable.

The idea of having the Collector run concurrently with the Mutator goes back to the seminal papers of Dijkstra et al. [3] and Steele [20], which were followed by many other papers trying to improve the algorithm or its verification. The Doligez-Leroy-Gonthier algorithm (short: DLG) that was developed for the Concurrent CAML Light system [4,5], is considered an important milestone, since it not only takes many practical complications of real-world collectors into account, but also generalizes from a single Mutator to many Mutators. The transition to concurrent garbage collection necessitates a *trade-off between the precision of the Collector and the degree of concurrency it provides* [21]: the higher the degree of concurrency, the more garbage nodes will be overlooked. However, this is no major concern in practice, since the escaped garbage nodes will be found in the next collection cycle.

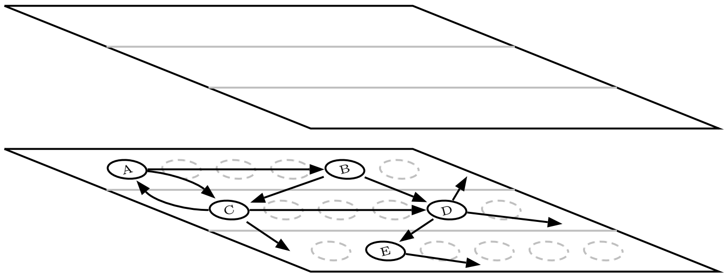


Fig. 3. At the start of the Collector

We now illustrate the key problem that can arise in concurrent garbage collection. Figure 3 illustrates the situation at the beginning of the collection process by showing a little fragment of the store; solid nodes are reachable from the root A, dashed circles represent dead garbage nodes (the arcs of which are not drawn here for the sake of readability). We use the metaphor of "planes" to illustrate both mark-and-sweep and copying collectors. In the former, "lifting" a node to the upper plane means marking, in the latter it means copying. The picture already hints at a later generalization, where the store is partitioned into "regions".

Figure 4 shows an intermediate snapshot of the algorithm. Some nodes and arcs are already lifted (i.e. marked or copied), others are still not considered. The gray nodes are in the the "workset" which means that they are marked/copied, but not all outgoing arcs have been handled yet.

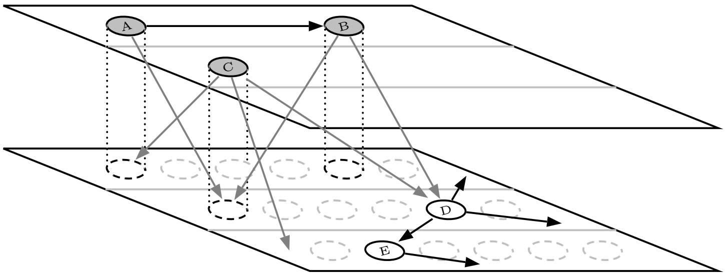


Fig. 4. A snapshot

Figure 5 shows the next snapshot. Now all direct successors of A have been treated. Therefore A is taken out of the workset, which we represent by the color black. Note that we have the invariant property that all downward arrows start in the workset. This corresponds to one of the two main invariants in the original paper of Dijkstra et al. [3].

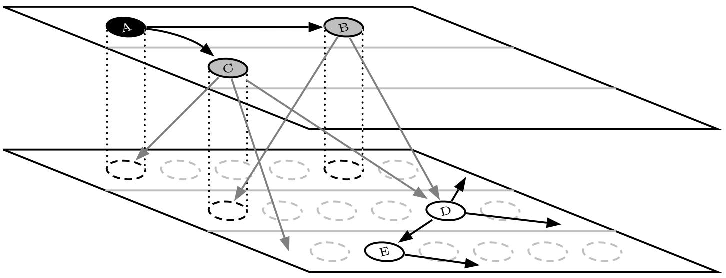


Fig. 5. The next snapshot

Now let us assume that in this moment the Mutator intervenes by adding an arc $A \rightarrow E$ and then deleting the arc $D \rightarrow E$. This leads to the situation in Figure 6. Since A is no longer in the workset, its connection to E will not be detected. Hence, E is hidden from the Collector and therefore will be treated, erroneously, as a dead garbage node.

There are three reasonable ways to cope with this problem (using suitable *write barriers*):

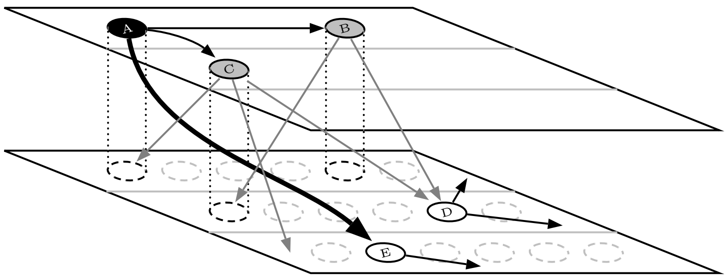


Fig. 6. A subtle error

- When performing $addArc(A, E)$, record E. (This is the approach of Dijkstra et al. [3].)
- When performing $addArc(A, E)$, record A. (This is the approach taken by Steele [20].)
- When performing $delArc(D, E)$, record E. (This is the approach taken by Yuasa [22].)

Note also that this bug may appear in an even subtler way *during* the handling of a node in the workset. Consider the node C in Figure 5 and suppose that it has lifted the first two of its three arcs. At this moment the Mutator redirects the first pointer field to, say, E. But a naive Collector will nevertheless take node C out of the workset (color it black) when its final arc has been treated.

2.1 Architecture and Basic Terminology

We modularize the problem by way of three kinds of components; see Figure 7. The *Mutators* represent the activities of all programs that use the heap. These activities base on primitive operations that are provided by the component *Store*, which represents the memory management system (as part of the runtime system or operating system). Finally the task of the garbage collection is performed by a component *Collector*.

The *Mutator* operates on a *graph*, which is a data structure of type $Graph(Node, Arc)$. It can essentially perform three primitive operations:[2]

- $addArc(a, b)$: add a new arc node a to node b.
- $delArc(a, b)$: delete the arc between a and b. This may have the effect that b and other nodes reachable from b become unreachable ("garbage").
- $addNew(a)$: allocate a new node b (from the freelist) and attach it by an arc from a. This reflects the fact that in reality *alloc* operations return a pointer, which is stored in some field (variable, register, heap cell) of the Mutator. Hence, the new node is immediately linked to the Mutator's graph.

[2] This considerably simplifies the memory model used in the DLG algorithm [4,5], where the Mutator has eight operations. However, the essence of these operations is captured by our three operations.

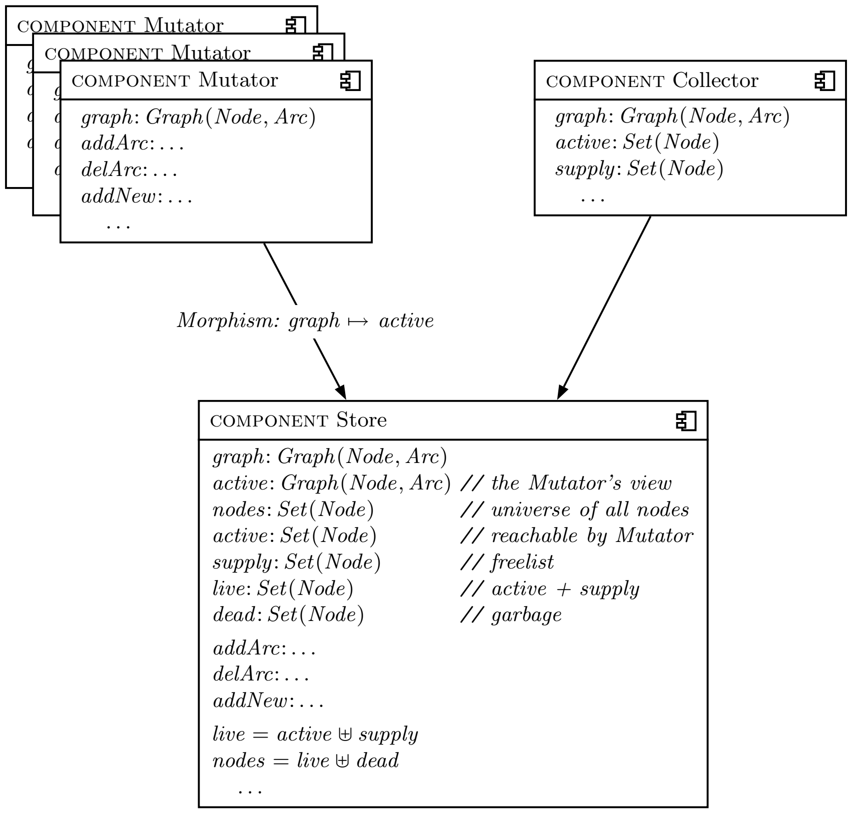


Fig. 7. The system architecture

The *Store* provides the low-level interface to the actual memory-access operations. We distinguish the following sets:

- *active* are those nodes that constitute the Mutator's graph.
- *supply* are the nodes in the freelist. (They become *active* through the operation *addNew*.)
- *live* is a shorthand for the union of the *active* and *supply* nodes.
- *dead* are the garbage nodes that are neither reachable from the Mutator nor in the freelist. (Nodes may become *dead* through the operation *delArc*.)

Note that the specifications in Figure 7 use $A = B \uplus C$ as a shorthand notation for the two properties $A = B \cup C$ and $B \cap C = \emptyset$. They also use overloading of operation names. For example *active* is used both for the subgraph that constitutes the Mutator's view and for the set of nodes in this subgraph. Such overloaded symbols must always be distinguishable from their context. Note also that we frequently refer to the "set" *Arcs* of the arcs of a graph and also to the "set" $sucs(a)$ of all successors of a node a; but these are actually *multisets*, since two nodes may be connected by several arcs. Technically, the cell has several slots that all point to the same cell.

The Mutator's operations *addArc*, *delArc*, *addNew* have an invariant property that is decisive for the working of any kind of garbage collector: *being garbage is a stable property* [1].

Proposition 1 (Antitonicity of Mutator). *A Mutator can only access the nodes in its graph and the freelist; that is, it can never access garbage nodes. In other words, the set of live nodes (graph + freelist) monotonically* decreases*.*

2.2 The Fundamental Specification of Garbage Collection

Surprisingly often papers on garbage collection refer to an intuitive understanding of what the Collector shall achieve. But in a formal treatment we cannot rely on intuition; rather we have to be absolutely precise about the goal that we want to achieve. Consider the architecture sketched in Figure 7. The Mutator continuously performs its basic operations *addArc*, *delArc* and *addNew*, which – from the Mutators viewpoint – are all considered to be total functions; i.e. they return a defined value on all inputs. This is trivially so for *addArc* and *delArc*, since their arguments exist in the mutators graph. The problematic operation is *addNew*, since this operation needs an element from the freelist. However, the freelist may be empty (i.e. $supply = \emptyset$). In this situation there are two possibilities:

1. $|\,active| = MemorySize$. That is, the Mutator has used all available memory in its graph. Then nothing can be done!
2. $|\,active| < MemorySize$. When $supply = \emptyset$, this means that $dead \neq \emptyset$. This is the situation in which we want to recycle garbage cells into the freelist. And this is the Collector's reason for existence!

Based on this reasoning, we obtain two basic principles for the Mutator/Collector paradigm.

Assumption 1 (Boundedness of Mutator's Graph). *We presume the following global property:* $|\,Mutator.graph| < MemorySize$

Under this global assumption the Collector has to ensure that the operation *addNew* is a total function (which may at most be delayed). This can be cast into a temporal-logic formula:

Goal 2 (Specification of Collector). $\Box\Diamond\, supply \neq \emptyset$ *(provided assumption 1 holds)*

This is a liveness property stating that "at any point in time the freelist (may be empty but) will eventually be nonempty." When this condition is violated, that is, $supply = \emptyset$, then it follows by the global Assumption 1 that $dead \neq \emptyset$. Hence the Collector has to find at least *some* dead nodes, which it can then transfer to the freelist. This can be cast into an operation *recycle* with the initial specification given in Figure 8.

SPEC Collector

$recycle\colon Graph(Node, Arc) \rightarrow Set(Node)$
$\emptyset \subset recycle(G) \subseteq dead \qquad if\ dead \neq \emptyset$
$\emptyset = recycle(G) \qquad if\ dead = \emptyset$

Fig. 8. The Collector's task

Hence we should design the system's working such that the following property holds (using an ad-hoc notation for transitions).

Goal 3 (Required Actions of Collector). $\Box \Diamond \big(supply \longrightarrow supply \uplus recycle(G)\big)$

When Goal 3 is met, then the original Goal 2 is also guaranteed to hold. In other words, the collector has to periodically call *recycle* and add the found subset of the garbage nodes to the freelist.

Note that the above operation can happen at any point in time; we need not wait until the freelist is indeed empty. This observation leaves considerable freedom for optimized implementations which are all correct.

2.3 How to Find Dead Nodes

Unfortunately, the specification of *recycle* in Figure 8 is not easily implementable since the dead nodes are not directly recognizable. Since the *dead* nodes are the complement of the *live* nodes; i.e. $live = \complement\, dead = nodes \backslash live$, the idea comes to mind to work with the complement of *recycle*. This leads to the simple calculation

$$\begin{aligned}
&\emptyset \subset recycle(G) \subseteq dead\\
&\Leftrightarrow \complement\,\emptyset \supset \complement\, recycle(G) \supseteq \complement\, dead\\
&\Leftrightarrow nodes \supset \complement\, recycle(G) \supseteq live\\
&\Leftrightarrow nodes \supset trace(G) \supseteq live
\end{aligned}$$

where we introduce a new function $trace(G) = \complement\, recycle(G)$. This leads to the refined version of the Collector's specification in Figure 9. Note that this specification, which will form the starting point for our more detailed derivation, is *formally derived* from the fundamental requirements for garbage collection as expressed in Assumption 1 and Goal 2 above!

3 Mathematical Foundation: Fixed Points

In garbage collection one can roughly distinguish two classes of collectors:

– *Stop-the-world collectors*: these are the classical non-concurrent collectors, where the mutators need to be stopped while the collector works.
– *Concurrent collectors*: these are the collectors that allow the mutators to keep working concurrently with the collector (except for very short pauses).

SPEC Collector	
$\mathit{recycle}\colon \mathit{Graph}(\mathit{Node}, \mathit{Arc}) \to \mathit{Set}(\mathit{Node})$	
$\mathit{trace}\colon \mathit{Graph}(\mathit{Node}, \mathit{Arc}) \to \mathit{Set}(\mathit{Node})$	
$\mathit{recycle}(G) = \complement\, \mathit{trace}(G)$	
$\mathit{live} \subseteq \mathit{trace}(G) \subset \mathit{nodes}$	$\mathit{if}\ \ \mathit{dead} \neq \emptyset$
$\mathit{trace}(G) = \mathit{nodes}$	$\mathit{if}\ \ \mathit{dead} = \emptyset$

Fig. 9. The Collector's task (first refinement)

3.1 Classical Fixed Points (Stop-the-World Collectors)

Computing the set of garbage nodes in a stop-the-world collector can be treated as a classical fixpoint computation in a finite powerset lattice. We briefly review the basic concepts and then show how to calculate the overall structure of a marking algorithm.

- For a set $s = \{\, x_0, x_1, x_2, \ldots\}$ of type $\mathit{Set}(A)$ and a function $f\colon A \to A$ we use the overloaded function $f\colon \mathit{Set}(A) \to \mathit{Set}(A)$ by writing $f(s)$ as a shorthand for $\{f(x_0), f(x_1), f(x_2), \ldots\}$.
- A function $f\colon A \to A$ is *monotone*, if $x \leq y \Rightarrow f(x) \leq f(y)$ holds.
- The function f is *continuous*, if $f(\sqcup\{x_0, x_1, x_2, \ldots\}) = \sqcup\{f(x_0), f(x_1), f(x_2), \ldots\}$ holds.
- The function f is *inflationary* in x, if $x \leq f(x)$ holds.
- The element x is called a *fixed point* of f, if $x = f(x)$ holds; x is the *least fixed point*, if $x \leq y$ for any other fixed point y of f.
- The element x is called a *fixed point* of f *relative* to r, if $x = f(x) \wedge r \leq x$ holds.
- By $\widehat{f}(x) = \text{LEAST}\ u.\ u = f(u) \wedge x \leq u$ we denote the reflexive-transitive *closure* of f (when it exists); i.e. the function that yields the least fixed point of f relative to x.

Lemma 4 (Properties of the Closure $\widehat{f}$). *The closure $\widehat{f}(x)$ has a number of properties that we will utilize frequently:*

- $x \leq \widehat{f}(x)$ *(inflationary);*
- $\widehat{f}(\widehat{f}(x)) = \widehat{f}(x)$ *(idempotent);*
- $f(\widehat{f}(x)) = \widehat{f}(x)$ *(fixpoint);*
- $\widehat{f}(f(x)) = \widehat{f}(x)$ *if* $x \leq f(x)$

Theorem 1 (Kleene[8]). *For a* continuous *function f the least fixed point x is obtained as the least upper bound of the Kleene chain:*

$$x = \sqcup\{\, \bot, f(\bot), f^2(\bot), f^3(\bot), \ldots\}$$

where $\bot$ is the bottom element of the lattice.

It has been shown that the essence of these theorems also holds in the simpler structure of *complete partial orders (cpos)*[3]. Cai and Paige [2] present a number of generalizations of Theorem 1 that are streamlined towards practical algorithmic implementations of fixpoint computations.

Theorem 2 (Cai-Paige). *Let A be a cpo and $f\colon A \to A$ be a monotone function that is inflationary in r. Let $\{s_0, s_1, s_2, \ldots, s_n\}$ be an arbitrary sequence obeying the conditions*

$$\begin{aligned} r &= s_0 \\ s_i < s_{i+1} &\le f(s_i) \quad \textit{for } i < n \\ s_n &= f(s_n) \end{aligned}$$

then s_n is the least fixed point *of f relative to r. Conversely, when the least fixed point is finitely computable, then the sequence will lead to such an s_n.*

Theorem 2 provides a natural abstraction from workset-based iterative algorithms, which maintain a workset of change items. At each iteration, a change item is selected and used to generate the next element of the iteration sequence. The incremental changes tend to be small and localized, hence this is called the *micro-step* approach, and the Kleene chain the *macro-step* approach [14]. All practical collectors use a workset that records nodes that await marking.

Corollary 1 (Invariance of Closure). *The elements of the set of approximations $\{ s_0 < s_1 < s_2 < \ldots < s_n \}$ all have the same closure: $\widehat{f}(s_i) = \widehat{f}(r)$.*

Using these basic results, we derive the overall structure of a marking algorithm for a stop-the-world collector. The essence of it is the iterative algorithm for finding garbage nodes to recycle.

Letting *roots* denote the roots of the active graph together with the head of the *supply* list, we have $live = \widehat{f}(roots)$ where $f(R) = \{b \mid b \in G.sucs(a) \;\&\; a \in R\}$; in words, the *active* nodes are the closure of the roots under the successor function in the current graph G.

To derive an algorithm for computing the *dead* nodes, we calculate as follows:

$$\begin{array}{lll} \multicolumn{2}{l}{dead} & \\ = & \complement\, live & \text{definition} \\ = & \complement\, \widehat{f}(roots) & \text{definition} \\ = & \check{g}(roots) & \text{using the law } \complement\, \widehat{h}(R) \;=\; \check{\imath}(R) \text{ where } i(x) = \complement h(\complement\, x) \end{array}$$

where $\check{g}(R)$ is the greatest fixpoint of the monotone function

$$g(x) = nodes \setminus (roots \cup \{b \mid b \in sucs(a) \;\&\; a \in nodes \setminus x\}).$$

This allows us to produce a correct, but naive iterative algorithm to compute dead nodes via a Kleene chain.

[3] A *cpo* is a partial order in which every directed subset has a supremum.

Program 1. Raw Fixpoint Iteration Algorithm

```
$W \leftarrow h.nodes$;
while $W \neq g(W)$ do $W \leftarrow g(W)$
return $W$
```

Following Cai and Paige [2], we can construct a more efficient fixpoint iteration algorithm using a workset defined by

$$WS \;=\; X \setminus g(X).$$

Although this workset definition is created by instantiating a problem-independent scheme, it has an intuitive meaning: the workset is the set of nodes whose parents have been "marked" as live, but who themselves have not yet been marked. The workset expression can be simplified as follows

$X \setminus g(X)$

$=$ $\{$ Definition $\}$

$X \setminus (nodes \setminus (roots \cup \{b \mid b \in sucs(a) \;\;\&\;\; a \in nodes \setminus X\}))$

$=$ $\{$ Using the law $A \setminus (B \cup C) \;=\; (A \setminus B) \setminus C$ $\}$

$X \setminus ((nodes \setminus roots) \setminus \{b \mid b \in sucs(a) \;\;\&\;\; a \in nodes \setminus X\})$

$=$ $\{$ Using the law $A \setminus (B \setminus C) \;=\; (A \setminus B) \bigcup (A \cap C)$ $\}$

$(X \setminus (nodes \setminus roots)) \bigcup (\{b \mid b \in sucs(a) \;\;\&\;\; a \in nodes \setminus X\} \cap X)$

$=$ $\{$ Using the law $\{x|P(x)\} \cap Q \;=\; \{x|P(x) \wedge x \in Q\}$ $\}$

$(X \setminus (nodes \setminus roots)) \bigcup \{b \mid b \in sucs(a) \;\;\&\;\; b \in X \;\;\&\;\; a \in nodes \setminus X\}$

$=$ $\{$ Again using the law $A \setminus (B \setminus C) = (A \setminus B) \bigcup (A \cap C)$ (on first term) $\}$

$(X \setminus nodes) \cup (X \cap roots) \bigcup \{b \mid b \in sucs(a) \,\&\, b \in X \;\;\&\;\; a \in nodes \setminus X\}$

$=$ $\{$ Simplifying $\}$

$\{\} \cup (X \cap roots) \bigcup \{b \mid b \in sucs(a) \;\;\&\;\; b \in X \;\;\&\;\; a \in nodes \setminus X\}$

$=$ $\{$ Simplifying$\}$

$(X \cap roots) \bigcup \{b \mid b \in sucs(a) \;\;\&\;\; b \in X \;\;\&\;\; a \in nodes \setminus X\}.$

The greatest fixpoint expression can be computed by the workset-based Program 2 justified by Theorem 2.

Program 2. Workset-based Fixpoint Iteration Program

```
$W \leftarrow nodes$;
while $\exists z \in ((W \cap roots) \bigcup \{b \mid b \in sucs(a) \;\&\; b \in W \;\&\; a \in nodes \setminus W\})$
    $W \leftarrow W - z$
return $W$
```

To improve the performance of this algorithm, we apply the Finite Differencing transformation [11] and incrementally maintain the invariant

$$WS = (W \cap roots) \bigcup \{b \mid b \in sucs(a) \;\&\; b \in W \;\&\; a \in nodes \setminus W\}.$$

The calculations to enforce the invariant (detailed in [13]) result in the code is shown in Program 3, where concurrent assignment is used to update both W and the workset WS.

Program 3. Optimized Fixpoint Iteration Algorithm

```
invariant $WS = (W \cap roots) \bigcup \{b \mid b \in sucs(a) \;\&\; b \in W \;\&\; a \in nodes \setminus W\}$
$W, WS$ := $nodes$, $roots$;
while $\exists z \in WS$ do
        $W, WS := W - z,\ WS \bigcup \{b \mid b \in sucs(z) \;\&\; b \in W\}) - z$
output $W$.
```

Program 3 represents the abstract structure of most marking algorithms. Our point is that its derivation, and further steps toward implementation, are carried out by generic, problem-independent transformations, supported by domain-specific simplifications, as above. Further progress toward a detailed implementation requires a variety of other transformations, including finite differencing, simplification, and datatype refinements. For example, the finite set W may be implemented by a characteristic function, which in turn is refined to a bit array, or concurrent data structures for local buffers or work-stealing queues.

3.2 Fixed Points in Dynamic Settings (Concurrent Collectors)

The classical fixed-point considerations work with a fixed monotone function f. In the garbage collection application this is justified as long as the graph, on which the collector works, remains fixed during the collector's activities. But as soon as the mutator is working in parallel with the collector, the graph keeps changing, while the collector is active. This can be modeled by considering a sequence of graphs G_0, G_1, G_2, ... and by making the function f dependent on these graphs: $f(G_0)(\ldots), f(G_1)(\ldots), f(G_2)(\ldots), \ldots$, where the function f is $f\colon \mathit{Graph} \to \mathit{Set}(\mathit{Node}) \to \mathit{Set}(\mathit{Node})$ and

$$f(G)(S) = S \bigcup \{b \mid a \in S \ \ \& \ \ b \in G.\mathit{sucs}(a)\}.$$

Intuitively, f extends a given set of nodes with the set of their successors in the graph. To ease readability we omit the explicit reference to the graphs and simply write $f_0, f_1, f_2, \ldots$. Using this notational liberty the specification of the underlying foundation is stated in Figure 10: the f_i are monotone ① and inflationary in r ②. Moreover the closure-forming operator $\widehat{f}$ is defined by ③.

SPEC Foundation	
EXTEND $\mathit{Cpo}(A)$	*// A is a cpo (alternatively: lattice)*
$f_0, f_1, f_2, \ldots\colon A \to A$	*// sequence of functions*
$\widehat{\_}\colon A \to A \to A \to A$	*//* $\widehat{f}$ *is reflexive-transitive closure of f*
$r\colon A$	*// "root"*
$x \leq y \Rightarrow f_i(x) \leq f_i(y)$	① *// all* f_i *are monotone*
$r \leq f_i(r)$	② *// all* f_i *are inflationary in r*
$\widehat{f}(x) =$ LEAST $s\colon\ x \leq s \wedge s = f(s)$	③ *// closure (computes least fixed point)*

Fig. 10. Initial Specification

Based on this foundation we can now formulate our goal. Recall the specification of the garbage collection task given by *Collector* in Figure 9 by the two inclusions $\mathit{live} \subseteq \mathit{trace}(G) \subset \mathit{nodes}$. This translates into our dynamic setting as $\mathit{live}_n \subseteq s \subset \mathit{nodes}$. We add as a working hypothesis that the set live_0 serves as an upper bound that we will need to guarantee in our dynamic algorithm: $\mathit{live}_n \subseteq s \subseteq \mathit{live}_0 \subset \mathit{nodes}$. The set live_0 is sometimes called the "snapshot-at-the-beginning" [1]. Since in our abstract setting live_n corresponds to the closure $\widehat{f_n}(r)$ and live_0 corresponds to the closure $\widehat{f_0}(r)$, we immediately obtain the abstract formulation ⑤ of our problem statement (Figure 11).

Axiom ④ is the abstract counterpart of the fundamental Proposition 1: the set of live nodes is monotonically decreasing over time, or, dually, garbage increases monotonically. For proof-technical reasons we have to conditionalize this property to any set x containing the roots r.)

Note that the existential formula ⑤ is trivially provable by setting $n = 0$ and $s = \widehat{f_0}(r)$. Actually the property ⑥ (see Lemma 5 below) shows that such an s exists for any n. However, our actual task will be to come up with a *constructive algorithm* that yields such an n and s.

For the specification *FixpointProblem* we can prove the property ⑥ (i.e. Lemma 5) that will be needed later on. This monotonic decreasing of the closure is in accordance with our intuitive perception of the Mutator's activities. The

SPEC Fixpoint-Problem		
EXTEND *Foundation*		
$r \le x \Rightarrow f_{i+1}(\widehat{f_i}(x)) \le \widehat{f_i}(x)$	$\textcircled{4}$	*// garbage can only grow*
THM $\exists\, n,\ s\colon\ \widehat{f_n}(r) \le s \le \widehat{f_0}(r)$	$\textcircled{5}$	*// $live_n \le s \le live_0$*
THM $r \le x \Rightarrow \widehat{f_0}(x) \ge \widehat{f_1}(x) \ge \widehat{f_2}(x) \ge \dots$	$\textcircled{6}$	*// Lemma 5*

Fig. 11. Fixpoint Specification

operation *delArc* may lead to fewer live nodes. And the operations *addArc* and *addNew* do not change the set of live nodes (since the freelist is part of the live nodes).

Lemma 5 (Antitonicity of Closure). *The closures are monotonically decreasing:*

$$\textit{For}\quad r \le x \quad \textit{we have}\quad \widehat{f_0}(x) \ge \widehat{f_1}(x) \ge \widehat{f_2}(x) \ge \dots \qquad \textcircled{6}$$

Proof: We use a more general formulation of this lemma: For monotone g and h we have the property

$$\forall x\colon\ g(\widehat{h}(x)) \le \widehat{h}(x) \Rightarrow \widehat{g}(x) \le \widehat{h}(x)$$

We show by induction that $\forall i\colon\ g^i(x) \le \widehat{h}(x)$. Initially we have $g^0(x) = x \le \widehat{h}(x)$ due to the general reflexivity property $\textcircled{3}$ of the closure. The induction step uses the induction hypothesis and then the premise: $g^{i+1}(x) = g(g^i(x)) \le g(\widehat{h}(x)) \le \widehat{h}(x)$. By instantiating f_{i+1} for g and f_i for h we immediately obtain $\widehat{f_{i+1}}(x) \le \widehat{f_i}(x)$ by using the axiom $\textcircled{4}$, when $r \le x$. *(End of proof)*

3.3 The *Microstep* Refinement

In order to get closer to constructive solutions we perform our first essential refinement. Generalizing the idea of Cai and Paige in Theorem 2, we add further properties to our specification, resulting in the new specification of Figure 12. Note that we now use some member s_n of the sequence $s_0, s_1, s_2, \dots$ as a witness for the existentially quantified s.

Proof of property $\textcircled{9}$: In a finite lattice the s_i cannot grow forever. Therefore there must be a fixpoint $s_n = f_n(s_n)$ due to axiom $\textcircled{8}$. Then the left half of the proof of $\textcircled{9}$ follows trivially from monotonicity:

$\forall i\colon\ r \le s_i$	*// axiom $\textcircled{7}$ and $\textcircled{8}$*
$\vdash\ \ r \le s_n = f_n(s_n)$	*// s_n is fixpoint*
$\vdash\ \ \widehat{f_n}(r) \le \widehat{f_n}(f_n(s_n)) = \widehat{f_n}(s_n) = s_n$	*// properties of $\widehat{f_n}$ Lemma 4*

The right half $s_n \le \widehat{f_0}(r)$ is a direct consequence of the following Lemma 6. *(End of proof)*

SPEC Micro-Step

EXTEND *FixpointProblem*
$s_0,\ s_1,\ s_2,\ \ldots : A$ *// sequence of approximations*
$s_0 = r$ $\textcircled{7}$ *// start with "root"*
$s_i < s_{i+1} \le f_i(s_i) \vee s_i = f_i(s_i)$ $\textcircled{8}$ *// computation step*
THM $\exists\, n\colon\ \widehat{f}_n(r) \le s_n \le \widehat{f}_0(r)$ $\textcircled{9}$ *// to be shown below*
THM $\widehat{f}_0(s_0) \ge \widehat{f}_1(s_1) \ge \ldots \ge \widehat{f}_n(s_n)$ $\textcircled{10}$ *// Lemma 6 below*

Fig. 12. The "micro-step approach"

Lemma 6 (Decreasing Closures). *As a variation of Lemma 5 we can show property* $\textcircled{10}$*: the closures are* ***de****creasing, even when applied to the* ***in****creasing* s_i*:*

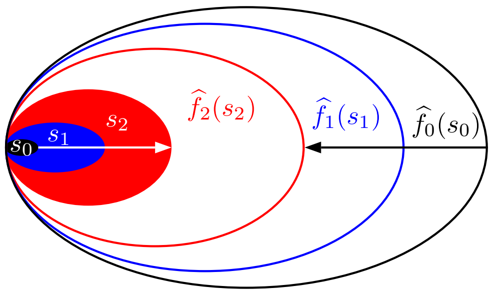
$$\forall i\colon\ \widehat{f}_{i+1}(s_{i+1}) \le \widehat{f}_i(s_i)$$

Proof: On the basis of Lemma 5 (property $\textcircled{6}$ in Figure 11) the proof follows directly from axiom $\textcircled{8}$ by monotonicity:

$s_{i+1} \le f_i(s_i)$ *// axiom* $\textcircled{8}$
$\vdash\ \widehat{f}_{i+1}(s_{i+1}) \le \widehat{f}_{i+1}(f_i(s_i)) \le \widehat{f}_i(f_i(s_i)) = \widehat{f}_i(s_i)$ *// monot. of* $\widehat{f}_{i+1}$*;* $\textcircled{6}$

Note that $\textcircled{6}$ is applicable here, since – due to $\textcircled{8}$ – $r \le f_i(s_i)$ holds. *(End of proof)*

Lemma 6 may be depicted as follows:



where the approximations $s_0,\ s_1,\ s_2,\ \ldots$ keep growing, while their closures $\widehat{f}_0(s_0),\ \widehat{f}_1(s_1),\ \widehat{f}_2(s_2),\ldots$ keep shrinking.

This essentially concludes the derivation that can reasonably be done on the abstract mathematical level of fixed points and lattices.

4 Garbage Collection in Dynamic Graphs

We now take specific properties of garbage collection into account, but still on the semi-abstract level of sets and graphs. First we note that our specification of garbage collection using sets and set inclusion is a trivial instance of the lattice-oriented specification in the previous section. Therefore all results carry over to the concrete problem. The morphism is essentially defined by the following map:

$$\Phi = \begin{bmatrix} A & \mapsto & Set(Node) \\ \leq & \mapsto & \subseteq \\ f_i(s) & \mapsto & f(G_i)(s) = s \cup G_i.sucs(s) = s \cup \bigcup_{a\in s} G_i.sucs(a) \\ r & \mapsto & G_0.roots \end{bmatrix}$$

- The basis now is a sequence of graphs $G_0,\ G_1,\ G_2,\ldots$ which are due to the activities of the Mutator.
- The function $f(G_i)(s) = s \cup \bigcup_{a\in s} G_i.sucs(a)$ adds to the set s all its direct successors. (We will retain the shorthand notation $f_i = f(G_i)$ in the following).

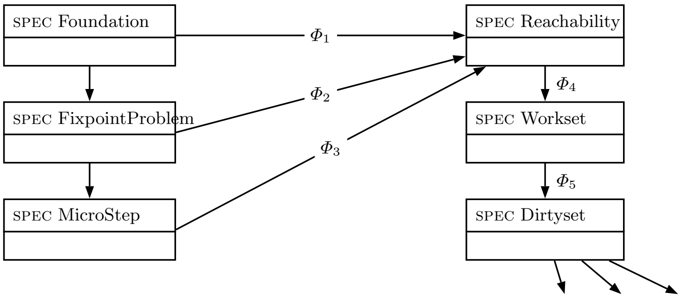


Fig. 13. Roadmap of refinements

Figure 13 illustrates the road map through our essential refinements. The left half shows the refinements that have been performed in the previous Section 3 on the abstract mathematical level of lattices and fixed points. The right half shows the refinements on the semi-abstract level of graphs and sets that will be presented in this section.

Lemma 7 (Morphism Abstract $\rightarrow$ Concrete). *Under the morphism Φ, all axioms of the abstract specifications Foundation, FixpointProblem and MicroStep hold for the more concrete specifications of graphs and sets (see Figure 13).*

Proof: We show the three morphism properties Φ_1, Φ_2, Φ_3 in turn.

Φ_1: The proof is trivial, since the monotonicity axiom ① is a direct consequence of the definition of $\Phi(f_i)$. Axiom ③ is just a definition.

Φ_2: To foster intuition, we first consider the special case $x = r$: the morphism translates:

$$\begin{array}{ll} ④ \overset{\Phi}{\mapsto}\ f_{i+1}\big(\widehat{f_i}(r)\big) \subseteq \widehat{f_i}(r) & \\ \quad\Leftrightarrow & // \ \widehat{f_i}(r) = live_i,\ def.\ of\ \Phi(f_i) \\ \quad\big(live_i \cup \bigcup_{a\in live_i} G_{i+1}.sucs(a)\big) \subseteq live_i & \\ \quad\Leftrightarrow & // \ (A_1 \cup \ldots \cup A_n) \subseteq B \Leftrightarrow \forall i : A_i \subseteq B \\ \quad\forall a \in live_i\colon G_{i+1}.sucs(a) \subseteq live_i & \end{array}$$

In order to prove this last property, i.e. $\forall a \in \mathit{live}_i\colon G_{i+1}.\mathit{sucs}(a) \subseteq \mathit{live}_i$, we must consider all nodes $a \in \mathit{live}_i$ and all (sequences of) actions that the Mutator can use to effect the transition $G_i \rightsquigarrow G_{i+1}$. We distinguish the two possibilities for $a \in \mathit{live}_i$:

(1) $a \in G_i.\mathit{freelist}$: Then there are two subcases (which base on the reasonable constraint that nodes in the freelist and newly created nodes do not have "wild" outgoing pointers):

$$\begin{array}{ll}(\mathit{1a}) & a \in G_{i+1}.\mathit{freelist}, \\ & \mathit{then}\ G_{i+1}.\mathit{sucs}(a) \subseteq G_{i+1}.\mathit{freelist} \subseteq G_i.\mathit{freelist} \subseteq \mathit{live}_i \\ (\mathit{1b}) & a \in G_{i+1}.\mathit{active}\ (\mathit{caused\ by\ addNew}), \\ & \mathit{then}\ \ G_{i+1}.\mathit{sucs}(a) = \emptyset\ ;\ \mathit{now}\ (2)\ \mathit{applies}\end{array}$$

(2) $a \in G_i.\mathit{active}$: Then there are three subcases for $b \in G_{i+1}.\mathit{sucs}(a)$:

$$\begin{array}{l}(\mathit{2a})\ (a \to b) \in G_i.\mathit{arcs}\ \vdash\ b \in G_i.\mathit{active} \subseteq \mathit{live}_i \\ (\mathit{2b})\ (a \to b)\ \mathit{created\ by\ addArc}(a,b)\ \vdash\ b \in G_i.\mathit{active} \subseteq \mathit{live}_i \\ (\mathit{2c})\ (a \to b)\ \mathit{created\ by\ addNew}(a)\ \vdash\ b \in G_i.\mathit{freelist} \subseteq \mathit{live}_i\end{array}$$

If we start this line of reasoning not from the roots r but from a superset $x \supseteq r$, then we need to consider supersets $\widehat{l_i} \supseteq \mathit{live}_i$ (where the hat shall indicate that these sets are closed under reachability) and prove $\forall a \in \widehat{l_i}\colon G_{i+1}.\mathit{sucs}(a) \subseteq \widehat{l_i}$. Evidently the reasoning in (1) and (2) applies here as well. But now there is a third case:

(3) $a \in G_i.\mathit{dead}$. In this case there is no operation of the Mutator that could change the successors of a (since all operations require $a \in \mathit{active}$). Hence $G_{i+1}.\mathit{sucs}(a) = G_i.\mathit{sucs}(a)$. Due to the closure property we have the implication $a \in \widehat{l_i} \Rightarrow G_i.\mathit{sucs}(a) \subseteq \widehat{l_i}$. The above equality then entails also $G_{i+1}.\mathit{sucs}(a) \subseteq \widehat{l_i}$.

Φ_3: The morphism Φ translates the axioms ⑦ and ⑧ into

$$s_i \subseteq s_i \cup \bigcup_{a \in s_i} G_i.\mathit{sucs}(a)$$

This is trivially fulfilled such that the constraint on the choice of s_{i+1} is well-defined. *(End of proof)*

When considering the last specification *Micro-Step* in Figure 12 then we have basically shown that any sequence $s_0, s_1, s_2, \ldots$ that fulfills the constraints ⑦ and ⑧ solves our task. But we have not yet given a *constructive* algorithm for building such a sequence. In the next refinement steps Φ_4 and Φ_5 we will proceed further towards such a constructive implementation (actually to a whole collection of implementation variants) by adding more and more constraints to our specification. Each of these refinements constitutes a design decision that narrows down the set of remaining implementations.

4.1 Worksets ("Wavefront")

As a first step towards more constructive descriptions we return to the standard idea of worksets (sometimes referred to as "wavefront"), which has already been illustrated Program 2, and in the examples in Section 2. This refinement is given in Figure 14.

SPEC Workset		
EXTEND *MicroStep*		
$b_0, b_1, b_2, \ldots : A$		*// completely treated ("black")*
$w_0, w_1, w_2, \ldots : A$		*// partially treated ("workset" or "gray")*
$s_i = (b_i \uplus w_i)$		*// partitioning into black and gray*
$\widehat{f}_i(s_i) = b_i \cup \widehat{f}_i(w_i)$	⑫	*// additional constraint*
THM $w_n = \emptyset \Rightarrow \widehat{f}_n(s_n) = b_n$	⑬	*// termination condition*

Fig. 14. The workset approach

The partitioning $s_i = (b_i \uplus w_i)$ arises naturally from the definition of the workset, as in Program 2. But the additional axiom ⑫ is a major constraint! It essentially states that the closure $\widehat{f}_i(s_i)$ of the current approximation s_i shall be primarily dependent on the closure of the workset w_i. This reduces the design space of the remaining implementations considerably – but from a practical viewpoint this is no problem, since we only exclude inefficient solutions. The theorem ⑬ stated in the specification provides a termination condition for the later implementations that is far more efficient than our original termination criterion $f_n(s_n) = s_n$.

An important observation: It is easily seen that the subtle error situation illustrated in Figure 6 in Section 2 violates the axiom ⑫. Therefore any further refinement of the specification *Workset* cannot exhibit this error. In other words: if we derive an implementation by refinement from the specification *Workset* in Figure 14, then we are certain that the bug cannot occur!

A major problem: Unfortunately, just introducing sufficient constraints for excluding error situations is not enough. Consider the situation of Figure 6 in Section 2. We have to ensure that the Mutator cannot perform the two operations $addArc(A, E)$ and $delArc(D, E)$ without somehow keeping the axiom ⑫ intact. This necessitates for the first time that the Mutator cooperates with the Collector, thus introducing constraints for the Mutator. Even though these constraints may be hidden in the component *Store*, they do have an implicit influence on the Mutator's working.

As has already been pointed out in Section 2, there are three principal possibilities to resolve this problem:

– One can stop the Mutator until the Collector has finished (Section 3.1).
– One can put A or E into the workset, when $addArc(A, E)$ is executed.
– One can put E into the workset, when $delArc(D, E)$ is executed.

Each of these solutions keeps the axiom ⑫ intact, but they have problems. Stopping the Mutator is unacceptable, since this destroys the very idea of having Mutator and Collector work concurrently. In both of the other cases the Mutator adds elements to the workset, while the Collector is taking them out of the workset. Naive implementations of this specification would not guarantee termination.

In the following we will present several refinements for solving this problem. These refinements are the high-level formal counterparts of solutions that can be found in the literature and in realistic production systems for the JVM and .Net.

4.2 Dirty Nodes

One can alleviate the stop times for the Mutator by splitting the workset into two sets, one being the original workset of the Collector, the other assembling the critical nodes from the Mutator. This is shown in Figure 15. The new axiom ⑭ is similar to ⑫ using the partitioning $w_i = (g_i \uplus d_i)$.

SPEC Dirtyset		
EXTEND *Workset*		
$g_0, g_1, g_2, \ldots : A$		*// treated by Collector ("gray")*
$d_0, d_1, d_2, \ldots : A$		*// introd. by Mutator ("dirty")*
$s_i = (b_i \uplus g_i \uplus d_i)$		*// partition black, gray and dirty*
$\widehat{f}_i(s_i) = b_i \cup \widehat{f}_i(g_i) \cup \widehat{f}_i(d_i)$	⑭	*// closure condition*
THM $g_n = \emptyset \Rightarrow \widehat{f}_n(s_n) = b_n \cup \widehat{f}_n(d_n)$	⑮	*// intermed. termination cond.*

Fig. 15. Introducing "dirty" nodes

This specification can be implemented by a Collector that successively treats the gray nodes in g_i until this set becomes empty (which can be guaranteed). But – by contrast to the earlier algorithms – this does not yet mean that all live nodes have been found. As the theorem ⑮ shows we still have to compute $\widehat{f}_i(d_i)$. But this additional calculation tends to be short in practice, and the Mutator can be stopped during its execution. Consequently, correctness has been retained and termination has been ensured.

The Mutator now adds "critical" nodes to the "dirty" set d_i. In order to keep the set d_i as small as possible one does not add all potentially critical nodes to it: as follows from axiom ⑭, black or gray nodes need not be put into d_i. And since d_i is a set, nodes need not be put into it repeatedly. Actually, when the Mutator executes $addArc(a, b)$ with $a \notin s_i$ ("a is still before the wavefront"), then axiom ⑭ would allows us the choice of putting a into d_i or not (similarly for b. Commonly, a is simply added to d_i.

4.3 Implementing the Step $s_i \mapsto s_{i+1}$

So far all our specifications only impose the constraint ⑧ (see *MicroStep* in Figure 12) on their implementations, that is:

$$s_i < s_{i+1} \leq f_i(s_i) \quad \vee \quad s_i = f_i(s_i)$$

The actual computation of the step $s_i \mapsto s_{i+1}$ has to be implemented by some function *step*. For this function we can have different degrees of granularity:

– In a *coarse-grained* implementation we pick some node x from the gray workset and add all its non-black successors to the workset. Then we color x black.
  This variant is simpler to implement and verify, but it entails a long atomic operation. The corresponding write barrier slows down the standard working of the Mutators.
– In a *fine-grained* implementation we treat the individual pointer fields within the current (gray) node x one-by-one. In our abstract setting this means that we work with the individual arcs.
  This makes the write barrier shorter and thus increases concurrency, but the implementation and its correctness proof become more intricate.

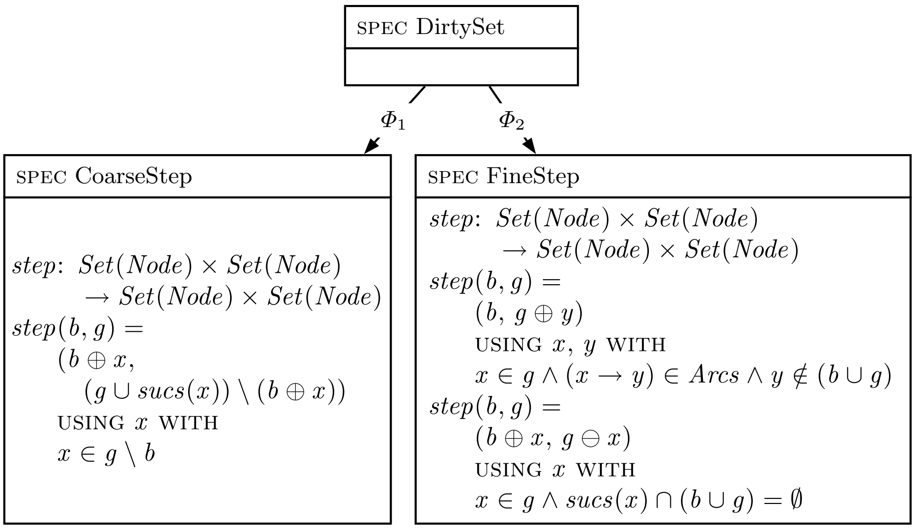
On our abstract level we treat this design choice by way of two different refinements. This is depicted in Figure 16 (where the shorthand notation ... USING x WITH $p(x)$ entails that the property only has to hold when such an x exists).

A note of caution. If we apply the morphism Φ introduced at the beginning of Section 4 directly, the strict inclusion $s_i < s_{i+1}$ of axiom ⑧ would not be provable. Therefore we must interpret

$$(b, g) < (b', g') \quad \overset{\Phi}{\mapsto} \quad b \subset b' \vee (b = b' \wedge g \subset g').$$

But there are still further implementation decisions to be made. Both *CoarseStep* and *FineStep* specify (at least partly) how the *step* operation deals with the selected gray node. But this still leaves one important design decision open: *How are the gray nodes selected?* In the literature we find several approaches to this task:

1. *Iterated scanning.* One may proceed as in the original paper by Dijkstra et al. [3] and repeatedly scan the heap, while applying *step* to all gray nodes



SPEC DirtySet

Φ_1 Φ_2

SPEC CoarseStep

$step$: $Set(Node) \times Set(Node) \rightarrow Set(Node) \times Set(Node)$
$step(b, g) =$
  $(b \oplus x,$
   $(g \cup sucs(x)) \setminus (b \oplus x))$
  USING x WITH
  $x \in g \setminus b$

SPEC FineStep

$step$: $Set(Node) \times Set(Node) \rightarrow Set(Node) \times Set(Node)$
$step(b, g) =$
  $(b,\ g \oplus y)$
  USING x, y WITH
  $x \in g \wedge (x \rightarrow y) \in Arcs \wedge y \notin (b \cup g)$
$step(b, g) =$
  $(b \oplus x,\ g \ominus x)$
  USING x WITH
  $x \in g \wedge sucs(x) \cap (b \cup g) = \emptyset$

Fig. 16. Step functions of different granularities

that are encountered. This has the advantage of not needing any additional space, but it may lead to many scans over the whole heap, in the worst case $\mathcal{O}(N^2)$ times, and is not considered practical.
2. Alternatively one performs the classical recursive graph traversal, which may equivalently be realized by an iteration with a workset managed as a stack. This allows all the well-known variations, ranging from a stack for depth-first traversal to a queue for breadth-first traversal. In any case the time cost is in the order $\mathcal{O}(|live|)$, since only the live nodes need to be scanned. However, there also is a worst-case need for $\mathcal{O}(|live|)$ space – and space is a scarce resource in the context of garbage collection.
3. One may compromise between the two extremes and approximate the workset by a data structure of bounded size (called a *cache* in [4,5]). When this cache overflows one has to sacrifice further scan rounds.
4. When there are multiple mutators, for efficiency it is necessary to have local worksets working concurrently.

These design choices are illustrated in Figure 17, but we refrain from coding all the technical details.

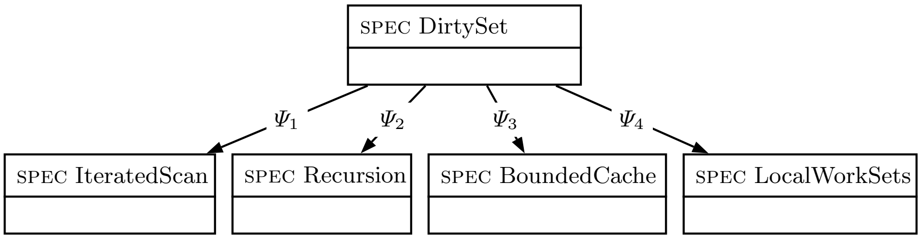


Fig. 17. Design choices for finding the gray nodes

It should be emphasized that the refinements Ψ_1, Ψ_2, Ψ_3, Ψ_4 of Figure 17 are independent of the refinements Φ_1, Φ_2 of Figure 16. This means that we can combine them in any way we like. The combination of some Φ_i with some Ψ_j is formally achieved by a pushout construction as already mentioned in Section 1. In a system like Specware [7] such pushouts are performed automatically.

Further refinements to handle generational garbage collectors, dirty cards, dirty pages and related techniques for scanning the dirty nodes are sketched in [13].

5 Conclusion

It is well known that realistic garbage collectors exhibit a huge amount of technical details that are ultimately responsible for the size and complexity of the verification efforts. The pertinent issues cover a wide range of questions such as:

– What are the exact read and write barriers?
– How do we treat the references in the global variables, the stacks and the registers?

– Where do we put the marker bits (in mark-and-sweep collectors) or the forward pointers (in copying collectors)?

Due to space limitations we have omitted discussion of these and other topics, which may however be found in the extended version of this paper [13], particularly the issue of computing the dynamically changing set of roots.

We have shown how the main design concepts in contemporary concurrent collectors can be derived from a common formal specification. The algorithmic basis of the concurrent collectors required the development of some novel generalizations of classical fixpoint iteration theory. We hope to find a wide variety of applications for the generalized theory, as there has been for the classical theory. This is of interest since the reuse of abstract design knowledge across application domains is a key factor in the economics of formal derivation technology. Alternative refinements from the basic algorithm lead to a family tree of concurrent collectors, with shared ancestors corresponding to shared design knowledge. While our presentation style has been pedagogical, the next step is to develop the derivation tree in a formal derivation system, such as Specware.

Acknowledgment. We are grateful to Erez Petrank and Chris Hawblitzel, with whom one of us (pp) enjoyed intensive discussions at Microsoft Research. Their profound knowledge on the challenges of practical real-world garbage collectors motivated us to push our original high-level and abstract treatment further towards concrete and detailed technical aspects – although we realize that we may still be on a very abstract level in the eyes of true practitioners. We would also like to thank Bernd Finkbeiner and the MPC reviewers for their helpful comments.

References

1. Azatchi, H., Levanoni, Y., Paz, H., Petrank, E.: An on-the-fly mark and sweep garabage collector based on sliding views. In: OOPSLA'03, Anaheim CA (2003)
2. Cai, J., Paige, R.: Program Derivation by Fixed Point Computation. Science of Computer Programming 11(3), 197–261 (1989)
3. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M.: On-the-fly garbage collection: An exercise in cooperation. Comm. ACM 21(11), 965–975 (1978)
4. Dolingez, D., Gonthier, G.: Portable, unobtrusive garbage collection for multiprocessor systems. In: POPL'94, Portland Oregon. ACM SIGPLAN Notices, pp. 70–83. ACM Press, New York (January 1994)
5. Dolingez, D., Leroy, X.: A concurrent generational garbage collector for a mulit-threaded implementation of ml. In: POPL'93. ACM SIGPLAN Notices, pp. 113–123. ACM Press, New York (1993)
6. Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. In: POPL'09, Savannah, Georgia, pp. 113–123 (October 2009)
7. Kestrel Institute, 3260 Hillview Ave., Palo Alto, CA 94304 USA. Specware System and documentation (2003), `http://www.specware.org/`
8. Kleene, S.: Introduction to Metamathematics. American Mathematical Society Press, Providence (1956)

9. MacCarthy, J.: Recursive functions of symbolic expressions and their computation by machine. Comm. ACM 3(4), 184–195 (1960)
10. McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. In: PLDI'07, San Diego (2007)
11. Paige, R., Koenig, S.: Finite differencing of computable expressions. ACM Transactions on Programming Languages 4(3), 402–454 (1982)
12. Pavlovic, D., Pepper, P.A., Smith, D.: Colimits for concurrent collectors. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 568–597. Springer, Heidelberg (2004)
13. Pavlovic, D., Pepper, P., Smith, D.: Formal derivation of concurrent garbage collectors. Technical Report TR-2010-1, Kestrel Institute (February 2010), `ftp://ftp.kestrel.edu/pub/papers/smith/PPS-2010.pdf`
14. Pepper, P., Hofstedt, P.: Funktionale Programmierung. Springer, Heidelberg (2006)
15. Russinoff, D.M.: A mechanically verified incremental garbage collectors. Formal Aspects of Computing 6, 359–390 (1994)
16. Smith, D.R.: KIDS – a semi-automatic program development system. IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16(9), 1024–1043 (1990)
17. Smith, D.R.: Toward a classification approach to design. In: Nivat, M., Wirsing, M. (eds.) AMAST 1996. LNCS, vol. 1101, pp. 62–84. Springer, Heidelberg (1996)
18. Smith, D.R.: Designware: Software development by refinement. In: Hoffman, M., Pavlovic, D., Rosolini, P. (eds.) Proceedings of the Eighth International Conference on Category Theory and Computer Science, pp. 355–370 (1999)
19. Smith, D.R., Parra, E.A., Westfold, S.J.: Synthesis of planning and scheduling software. In: Tate, A. (ed.) Advanced Planning Technology, pp. 226–234. AAAI Press, Menlo Park (1996)
20. Steele, G.L.: Multiprocessing compactifying garbage collection. Comm. ACM 18(9), 495–508 (1975)
21. Vechev, M.T., Yahav, E., Bacon, D.F.: Correctness-preserving derivation of concurrent garbage collection algorithms. In: PLDI 06, Ottawa, Canada. ACM Press, New York (2006)
22. Yuasa, T.: Real-time garbage collection on general-purpose machines. Journal of Systems and Software 11(3), 181–198 (1990)

Temporal Logic Verification of Lock-Freedom

Bogdan Tofan, Simon Bäumler, Gerhard Schellhorn, and Wolfgang Reif

Institute for Software and Systems Engineering
University of Augsburg
D-86135 Augsburg, Germany
{tofan,baeumler,schellhorn,reif}@informatik.uni-augsburg.de

Abstract. Lock-free implementations of data structures try to better utilize the capacity of modern multi-core computers, by increasing the potential to run in parallel. The resulting high degree of possible interference makes verification of these algorithms challenging. In this paper we describe a technique to verify lock-freedom, their main liveness property. The result complements our earlier work on proving linearizability, the standard safety property of lock-free algorithms. Our approach mechanizes both, the derivation of proof obligations as well as their verification for individual algorithms. It is based on an encoding of rely-guarantee reasoning using the temporal logic framework of the interactive theorem prover KIV. By means of a slightly improved version of Michael and Scott's lock-free queue algorithm we demonstrate how the most complex parts of the proofs can be reduced to relatively simple steps of symbolic execution.

Keywords: Verification, Temporal Logic, Compositional Reasoning, Rely-Guarantee, Lock-Freedom, Linearizability.

1 Introduction

The classic approach for protecting parts of a shared data structure from concurrent access is mutual exclusion locks. One severe disadvantage of this method is that the crash or suspension of a single process can cause a deadlock or delay of the entire system. Lock-free algorithms were developed to overcome this shortcoming. One of their main features is that the crash or delay of a single process has no negative effect on the progress of other processes. This is usually achieved by applying atomic synchronization primitives such as CAS (compare and swap) or LL/SC (load linked/store conditional) and an optimistic try and retry scheme:

1. The relevant part of the shared data structure to be modified is stored in a local variable (sometimes called "snapshot").
2. Modification of the shared data structure is prepared, e.g. local fields are assigned.
3. The shared data structure is updated in one step if no interference has occurred since taking the local snapshot.

If another process has changed the snapshot during phase 2 (which is kept as small as possible), the current process must retry until no interference hinders its update.

This basic idea is extended in lots of different ways, such as by introducing reciprocal helping schemes or executing additional algorithms between the fail and the retry of an update. These techniques have resulted in lock-free implementations of various data structures, amongst others stacks [1,2], queues [3], deques [4] and hash tables [5]. Some of the proposed algorithms had subtle errors which were found when trying to formally prove their correctness [6]. The complexity of these implementations justifies the effort of formal verification and various approaches have been proposed to prove correctness [7,8,9,10,11] and liveness [12,13,14].

The main correctness criterion for lock-free algorithms is *linearizability*. It requires each operation to appear to take effect instantaneously at some point (the linearization point) between invocation and response, behaving according to its sequential specification [15]. This property rules out certain interleavings but does not guarantee any kind of progress. *Lock-freedom* is a global liveness condition which requires that at all times in a concurrent execution, one of the running operations eventually completes [16]. Consequently, as soon as no further operations are invoked, all currently active operations eventually complete. However, if the system repeatedly invokes new operations, single processes might never complete, i.e. lock-freedom does not prevent single processes from starvation.

In this paper we present a verification approach based on rely-guarantee reasoning [17,18] and interval temporal logic [19,20] and demonstrate it using a practical lock-free queue algorithm published by Doherty et al. [7], based on the original implementation of Michael and Scott [3]. The approach allows to prove two decomposition theorems: a generic refinement theorem which can be instantiated to prove linearizability and a theorem for proving lock-freedom. Both theorems have been mechanically verified using the semi-automated prover KIV [21]. The theorem for proving linearizability has been described in [11], where the resulting proof obligations have been shown to be provable for a simple stack algorithm as well as for the dequeue operation of the queue. In this work we therefore focus on describing the lock-freedom theorem and its application to the enqueue algorithm.

The main contributions are:

- A fully mechanized approach for the intuitive specification and verification of lock-free algorithms. We provide an easy to read specification language and require no program counter values for reasoning.
- An expressive temporal logic framework which allows for a simple definition of the $\overset{+}{\rightarrow}$ operator from rely-guarantee reasoning, and to prove compositionality results for parallel programs as well as refinement (= trace inclusion) theorems.
- A decomposition theorem to prove lock-freedom which does not rely on the explicit construction of well-founded orders, but on intuitive arguments of program progress.

The paper is subdivided as follows: in Section 2 we describe the queue algorithm and argue informally about its liveness. Section 3 gives a short introduction to the temporal logic framework implemented in KIV. Section 4 describes the concurrent system model and rely-guarantee reasoning. Moreover, the decomposition theorem for proving lock-freedom is introduced. Section 5 shows its application to the queue. We conclude with a section about related work (Section 6) and a summary (Section 7).

2 Michael and Scott's Lock-Free Queue

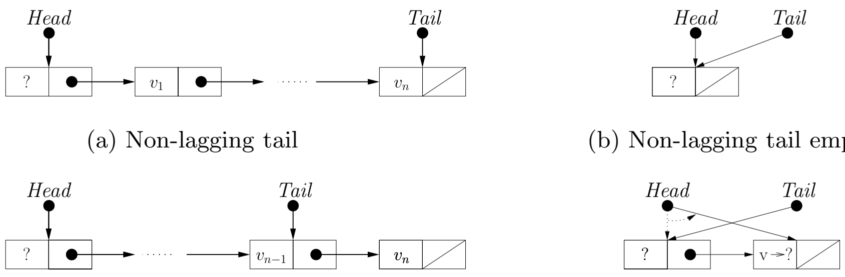
Lock-free algorithms typically use synchronization primitives such as CAS to atomically alter a shared data structure in the computer's memory. CAS can be formally specified in KIV as

```
CAS($Old, New; G, Succ$){
  **if*** $G = Old$ **then** {$G := New,\ Succ := true$} **else** {$Succ := false$}}
```

where value-parameters *Old* and *New* are read only whereas *G* and *Succ* denote reference-parameters that can be read and modified. CAS compares a global pointer *G* with the (snapshot) reference stored in pointer *Old*. If these memory locations are equal then *G* is updated to a new reference *New* and boolean flag *Succ* is set to true to indicate a successful CAS. Otherwise the flag is set to false indicating that no update has occurred. Since CAS executes atomically (a comma separates parallel assignments), evaluating the if-condition should not require an extra step (denoted as **if***). CAS does not guarantee that the value *A* of global pointer *G* has not been changed since it was read by a process. In the meantime, some other process might have changed *G* to *B* and then back to *A*. In a system that reuses freed references, these intermediate modifications can lead to subtle errors, since the content of a reallocated memory location might have been changed (ABA-problem). We assume (lock-free) garbage collection [22] and do not explicitly model memory reuse here. Any value assigned to *G* is going to be a newly allocated location, and this avoids an ABA-problem.

The queue is represented in memory as a singly linked list of nodes (pairs of values and references along with .val and .nxt selector functions), a global pointer *Head* which marks the front of the queue and a global pointer *Tail* indicating the end of the queue as shown in Figure 1 (a) and (b). At all times *Head* points to a dummy node (its value is irrelevant and denoted by a question mark). This avoids special cases for the empty queue in the implementation. There are two queue operations: the enqueue operation (*CEnq*) adds a node at the end of the queue; the dequeue operation (*CDeq*) removes the first node from the queue and returns its value. If the queue is empty, i.e. the dummy node's next reference is null, a special value *empty* is returned.

Attaching a new node at the end of the queue requires two global updates: the last node's next field must be set to the new node and the global tail pointer must be shifted. Since CAS allows only one atomic write access, it must be called twice. When a process encounters a lagging tail in-between these two



(a) Non-lagging tail (b) Non-lagging tail empty

(c) Lagging tail (d) Lagging tail empty

Fig. 1. Queue representation variants

CAS executions (see Figure 1 (c)), it helps by shifting the tail pointer before trying to add its new node in the next iteration.

```
E1  $CEnq(v; Hp, Tail, Newe, Tle, Nxte, SuccE)$ {
E2    choose $Ref$ with $Ref \neq null \wedge \neg\ Ref \in Hp$ in {
E3      $Hp := Hp \cup \{Ref\}, Newe := Ref, SuccE := false$;
E4      $Hp[Newe] := v \times null$;
E5      while $\neg\ SuccE$ do {
E6        $Tle := Tail$;
E7        $Nxte := Hp[Tle]$.nxt;
E8        if $Tle = Tail$ then {
E9          if $Nxte = null$ then {
E10           CAS($Nxte, Newe; Hp[Tle]$.nxt$, SuccE$)
E11         } else {
E12           CAS($Tle, Nxte; Tail$)}}}
E13     CAS($Tle, Newe; Tail$)}}
```

Fig. 2. Enqueue operation

Figure 2 shows the KIV specification of the enqueue operation (line numbers are given for explanatory purposes; they are not used in KIV). In lines E2 - E4 a new node is allocated (a fresh reference is chosen and added to the global application heap *Hp* in one atomic step) and initialized with input value v and a null next reference (a semicolon denotes sequential composition which may be interleaved). In E6 a local snapshot is taken. Its next reference is stored locally in the following line. The test in line E8 checks whether the global tail has not been changed since the snapshot was taken. If this test fails *CEnq* must retry its update due to interference. The next test in E9 discerns the role of the current loop execution: if *Nxte* is null, line E7 was executed when the global queue was in a non-lagging tail state and the current run might successfully attach a new node at the end of the queue in line E10 and subsequently exit the loop, given that no interference has occurred in the meantime. If the test in E9 is false, the loop will be reiterated and the current process can only try to help some other process by shifting the lagging tail pointer (line E12). The last instruction (line E13) tries to shift the tail pointer after attaching a new node to the queue. This "clean up" guarantees a non-lagging tail representation in quiescent states. *CEnq* uses a variant of CAS in which it is irrelevant to know whether it succeeds (lines E12 and E13). Since it is necessary to observe the values of local variables $\mathit{Newe}, \mathit{Tle}, \mathit{Nxte}, \mathit{SuccE}$ in assertions, they have been lifted to transient parameters (see Section 5).

The formalization of the dequeue operation is shown in Figure 3. A process executing *CDeq* takes a snapshot of the global head pointer in line D5 and then locally stores its next reference. If the snapshot has not become obsolete and the local next reference is null, dequeue returns *empty*. If the queue is not empty CAS is applied in line D12 to shift the global head pointer, making *Nxtd* the new dummy node. The remaining lines of code (D13-D16) then deal with a special configuration which emerges from shifting *Head* when the queue contains exactly one value v and the tail pointer is lagging (see Figure 1 (d)). Since the head pointer gets shifted ahead of the tail pointer, dequeue can help the process which has enqueued v (line D16).[1]

```
D1  $\mathit{CDeq}(; \mathit{Hp}, \mathit{Head}, \mathit{Tail}, \mathit{Hdd}, \mathit{Nxtd}, \mathit{SuccD}, O)$ {
D2     let $\mathit{Lo} = \mathit{empty},\ \mathit{Tld} = \mathit{null}$ in {
D3        $\mathit{SuccD} := \mathit{false}$;
D4        while $\neg\ \mathit{SuccD}$ do {
D5           $\mathit{Hdd} := \mathit{Head}$;
D6           $\mathit{Nxtd} := \mathit{Hp}[\mathit{Hdd}].\mathrm{nxt}$;
D7           if $\mathit{Hdd} = \mathit{Head}$ then {
D8              if $\mathit{Nxtd} = \mathit{null}$ then {
D9                 $\mathit{Lo} := \mathit{empty}; \mathit{SuccD} := \mathit{true}$
D10             } else {
D11                $\mathit{Lo} := \mathit{Hp}[\mathit{Nxtd}].\mathrm{val}$;
D12                CAS($\mathit{Hdd}, \mathit{Nxtd}; \mathit{Head}, \mathit{SuccD}$);
D13                if $\mathit{SuccD}$ then {
D14                   $\mathit{Tld} := \mathit{Tail}$;
D15                   if $\mathit{Tld} = \mathit{Hdd}$ then {
D16                      CAS($\mathit{Tld}, \mathit{Nxtd}; \mathit{Tail}$)}}}}}
D17       $O := \mathit{Lo}$}}
```

Fig. 3. Dequeue operation

The intuitive reason why the implementation is lock-free is that its loops are retried if some other process changes the queue in the critical time slot between taking a snapshot and trying to update the data structure. This change however implies that the interfering process eventually completes. In the formal proof we will reflect the simplicity of this intuitive argument by using an additional predicate U ("unchanged") which describes the absence of interference that can

[1] The original dequeue implementation of Michael and Scott reads the shared tail pointer whenever the loop-body is executed. The implementation given here reduces shared memory access if the loop has to be executed several times.

cause a process to retry its loop *infinitely* often. Proving lock-freedom then requires to show that each data structure operation eventually terminates when it either encounters no such interference or when it changes the shared state itself. For the dequeue operation this argument is simple. If it encounters no interference, then the CAS at D13 will be successful and dequeue terminates. A successful CAS at D13 is also the only place, where the process itself changes the shared data structure.

For the enqueue operation the argument is slightly more subtle. If the queue is not modified after taking the snapshot in E6, its loop might be executed *once* again before enqueue terminates, due to a lagging tail. But this does not hinder termination, so we do not count shifting a lagging tail as an interference (predicate U is true for that case). Finally, if enqueue changes the queue by adding a new node in E10 it terminates without further iterations.

As we will see, this intuitive reasoning is formally performed in KIV, by applying symbolic execution to step forward through each line of code of an operation.

3 Temporal Logic in KIV

This section briefly describes the temporal logic calculus integrated into the interactive theorem prover KIV. A more detailed description can be found in [23,24].

3.1 Interval Temporal Logic

The basis of interval temporal logic (ITL) [19,20] are algebras (to interpret the signature) and intervals, i.e. finite or infinite sequences of states (each mapping variable symbols to values in the algebra). Intervals typically evolve from program execution. In contrast to standard ITL, the formalism used here explicitly includes the behavior of the program's environment into each step: in an interval $I = [I(0), I'(0), I(1), I'(1), \ldots]$ the first program transition leads from the initial state $I(0)$ to the primed state $I'(0)$ whereas the next transition (from state $I'(0)$ to $I(1)$) is a transition of the program's environment. In this manner program and environment transitions alternate (similar to [25,26]).

Variables are partitioned into *static* variables v (written lower case), which never change their value ($I(0)(v) = I'(0)(v) = I(1)(v) = \ldots$) and *flexible* variables V (starting with an uppercase letter) which can have different values in different states of an interval. We write V, V', V'' to denote variable V in states $I(0)$, $I'(0)$ and $I(1)$ respectively. In the last state (characterized by the atomic formula **last**) of an interval, the value of a primed or double primed variable is equal to the value of the unprimed variable, i.e. after a program has terminated its variables do not change by convention.

The logic uses standard temporal operators ($\Box$, $\Diamond$, $\bullet$, **until**, **unless**, ...) as well as sequential programming constructs (:=, ; ,**if**, ...). We usually write α, β to indicate a program and φ, ψ to indicate a formula. $\alpha \parallel \beta$ is weak fair interleaving, **await** φ blocks execution until the test φ is satisfied (not used in the algorithms here). Programs and formulas can be mixed arbitrarily since they both evaluate to true or false over an algebra $\mathcal{A}$ and an interval I (hence system descriptions can be abstracted by temporal properties). A program evaluates to true ($\mathcal{A}, I \models \alpha$) if I is a possible run of the program. Note that a run of a program is always interleaved by arbitrary transitions of its environment. More details on the syntax and semantics of these operators can be found in [23].

3.2 Symbolic Execution and Induction

KIV is based on the sequent calculus. Sequents are assertions of the form $\Gamma \vdash \Delta$ where Γ and Δ are sets of formulas. A sequent states that the conjunction of all formulas in antecedent Γ implies the disjunction of all formulas in succedent Δ. Sequents are implicitly universally closed. A typical sequent (proof obligation) about interleaved programs has the form

$$\alpha, E, I \vdash \varphi$$

where an interleaved program α executes the system steps; the system's environment behavior is constrained by temporal formula E; I is a predicate logic formula that describes the current state and φ is the property which has to be shown. To verify that φ holds, symbolic execution is used. For example, a sequent of the form mentioned above might be

$$(M := M + 1;\ \beta),\ \Box\ M'' = M',\ M = 1\ \vdash\ \Box\ M > 0$$

The program executed is $M := M+1;\ \beta$ where β is an arbitrary program and the environment is assumed never to change counter M (formula $\Box\ M'' = M'$). The current state maps M to 1. The intuitive idea of a symbolic execution step is to execute the first program statement, i.e. to apply the changes on the current state and to discard the first statement. In the example above, a symbolic execution step leads to a trivial predicate logic goal for the initial state ($M = 1 \vdash M > 0$) and a sequent that describes the remaining interval from the second state on:

$$\beta,\ \Box\ M'' = M',\ M = 2\ \vdash\ \Box\ M > 0$$

M has value 2 in the new state which follows from the fact that after M has been set to two by the program transition, the environment leaves M unchanged. Otherwise M would have an arbitrary value in the new state. Symbolic execution concerns both programs and formulas and has two phases. In the first phase information about the first transition (both system and environment) is separated from information about the rest of the run. We get $M' = M + 1 \wedge M'' = M'$ from the assignment and the environment assumption for the first step and $\bullet\ \beta$ for the rest of the run. For $\Box\ M > 0$ we get $M > 0$ and $\bullet\ \Box\ M > 0$ using the unwinding rule $\Box\ \varphi \leftrightarrow \varphi \wedge \bullet\ \Box\ \varphi$. In the second phase of a symbolic execution step, the unprimed and primed variables M and M' are substituted with fresh static variables that describe the former state, whereas the double primed variable M'' is replaced with the unprimed variable M in the new state, and leading next operators ($\bullet$) are dropped.

In addition to symbolic execution, well-founded induction is used to deal with loops. For finite intervals it is possible to induce over the length of an interval. For infinite traces a well-founded order can often be derived from liveness properties $\Diamond\ \varphi$ by inducing over the number of steps N until φ holds:

$$\Diamond\ \varphi \ \leftrightarrow\ \exists\ N.\ (N = N'' + 1)\ \textbf{until}\ \varphi$$

The equivalence states that φ is eventually true, if and only if N can be decremented (note that $N = N'' + 1$ is equivalent to $N > 0 \wedge N'' = N - 1$) until φ becomes true. Proving a formula of the form $\Box\ \varphi$ on infinite traces is then simply done by rewriting $\Box\ \varphi$ to $\neg\ \Diamond\ \neg\ \varphi$ and a proof by contradiction. Similarly, an **unless** formula (as needed later in rely-guarantee proofs, cf. Section 5.1) can be reduced to the case of an eventually formula using the equivalence

$$\varphi\ \textbf{unless}\ \psi \leftrightarrow \forall\ B.\ (\Diamond\ B) \rightarrow (\varphi\ \textbf{unless}\ (\varphi \wedge B \vee \psi))$$

φ **unless** ψ is true if it is true on every prefix of the trace that is terminated by the first time when boolean variable B becomes true. This rewriting allows for extracting the liveness property $\Diamond\ B$ to prove that the initial unless formula holds, by applying well-founded induction over the number of steps until B is true. The (semantic) proofs of both equivalences above are straightforward.

4 Rely-Guarantee Reasoning and the Decomposition Theorem for Lock-Freedom

This section gives a short introduction to the concurrent system model in our approach and to the well-known decomposition technique of rely-guarantee reasoning. Furthermore, we describe a decomposition theorem for proving lock-freedom. Its formal proof is available online [27].

4.1 System Model and Rely-Guarantee Reasoning

A concurrent system is a program which spawns an arbitrary positive number of processes to execute in parallel

```
CSpawn($n$; $Act, In, CS, Out$) {
  if* $n = 0$ then
    CSeq($n$; $Act, In, CS, Out$)
  else
      CSeq($n$; $Act, In, CS, Out$)
    || CSpawn($n - 1$; $Act, In, CS, Out$)}
```

```
CSeq($m$; $Act, In, CS, Out$) {
  {   skip
    ∨ {$Act(m)$ := true;
       COP($m, In$; $CS, Out$);
       $Act(m)$ := false}
  }$^*$}
```

CSpawn consists of $n + 1$ processes that execute CSeq in parallel. Operation CSeq finitely or infinitely often (denoted by *) does some computations that have no direct influence on the underlying data structure (modeled as no operation *skip*) or it executes an arbitrary data structure operation COP (in the queue example, COP is simply the nondeterministic choice ($\vee$) between one of the two operations *CEnq* and *CDeq*).

Operation CSeq is called with a value parameter m of type *nat* which represents the identifier of the invoking process. Reference-parameter $Act : nat \to bool$ is a boolean function which is used to distinguish whether a process is currently active in the sense of currently executing COP (this activity flag is only relevant for proving lock-freedom). Function $In : nat \to input$ is used to pass an arbitrary input value $In(m)$ to COP. *In* is a reference parameter in CSeq whereas it is a value parameter in COP, i.e. whenever COP is invoked, its input value can differ from previous invocations due to changes on *In* by CSeq's environment (this ensures that different values can be enqueued). The remaining parameters include a generic state variable $CS : cstate$ for the (shared and local) state on which COP works and an output function $Out : nat \to output$ to return values.

Rely-guarantee reasoning is a widely used decomposition technique to prove properties of an overall concurrent system by looking at the system's components only [17,18]. To this end each process (component) m is extended with two predicates: a two-state rely predicate $R_m : cstate \times cstate$ describing the behavior of m's environment (including other processes within the system plus the environment of the entire system) and a binary guarantee predicate $G_m : cstate \times cstate$ which describes the impact of m on its environment (the first parameter of a guarantee/rely condition denotes the state before the system/environment step and the second argument denotes the next state). To ensure correctness each guarantee condition must preserve the rely conditions of all other processes

$$m \neq n \wedge G_m(CS_0, CS_1) \to R_n(CS_0, CS_1) \tag{1}$$

The intuitive idea of the rely-guarantee approach is to claim that every process m fulfills its guarantee G_m if every other process does not violate its rely condition R_m. To break circularity of this argument, a special implication operator $\overset{+}{\rightarrow}$ (as defined in [28]) is used which states that m fulfills its guarantee if its rely condition has not been violated in some preceding step ($R_m \overset{+}{\rightarrow} G_m$). The explicit separation between program and environment transitions in our logic enables us to specify guarantees as predicates $G_m(CS, CS')$ with unprimed and primed variables describing steps of process m. Rely conditions $R_m(CS', CS'')$ instead use primed and double primed variables to restrict steps of m's environment. The formal definition of $\overset{+}{\rightarrow}$ is then simply based on the temporal operator **unless**

$$R_m \overset{+}{\rightarrow} G_m :\equiv G_m(CS, CS')\,\textbf{unless}\,(G_m(CS, CS') \wedge \neg\, R_m(CS', CS''))$$

Since $\varphi\,\textbf{unless}\,\psi \leftrightarrow (\Box\ \varphi) \vee (\varphi\,\textbf{until}\,\psi)$, either the guarantee G_m always holds or it holds until a system step occurs in which the guarantee still holds, but where the subsequent environment transition violates m's rely condition.

In order to show that a process m which executes CSeq satisfies $R_m \overset{+}{\rightarrow} G_m$, two properties must be fulfilled. First, each guarantee must be reflexive (in case of skip or a step that sets the activity flag, the current state stays the same)

$$G_m(CS, CS) \tag{2}$$

Second, $R_m \overset{+}{\rightharpoonup} G_m$ must be preserved by the data structure operation

$$\mathrm{COP}(m, In; CS, Out), Inv(CS) \vdash R_m \overset{+}{\rightharpoonup} G_m \tag{3}$$

where predicate *Inv* : *cstate* introduces an invariant. Properties (2) and (3) also imply that every process m preserves its guarantee condition at all times, in an environment that always respects m's rely condition. To show that in this case m always preserves the invariant too, we stipulate stability of the invariant over rely steps:

$$Inv(CS') \wedge R_m(CS', CS'') \rightarrow Inv(CS'') \tag{4}$$

With (1) it follows that *Inv* is also stable over each local guarantee (note that (1) holds for arbitrary distinct natural numbers) and specifies indeed an invariant property

$$\mathrm{CS{\small EQ}}(m; \dots), \Box\, R_m(CS', CS''), Inv(CS) \vdash \Box\, (Inv(CS) \wedge Inv(CS'))$$

To lift this property (resp. (3)) to the level of an interleaved execution of the overall system CSpawn, it is necessary to be able to summarize several consecutive local rely steps in one rely step, i.e. we require R_m to be transitive

$$R_m(CS_0, CS_1) \wedge R_m(CS_1, CS_2) \rightarrow R_m(CS_0, CS_2) \tag{5}$$

Since the generic setting also takes into account the environment of the overall system, a global rely condition R : *cstate* $\times$ *cstate* is required too. It preserves each local rely condition

$$R(CS', CS'') \rightarrow R_m(CS', CS'') \tag{6}$$

Conditions (1) to (6) are the same as described in [11] for linearizability. The few extensions required to prove lock-freedom are introduced in the next section. As several of the following proof obligations will assume an invariant and a rely condition to always hold, we define the following abbreviation:

$$I(R) :\equiv Inv(CS) \wedge Inv(CS') \wedge R(CS', CS'')$$

4.2 Decomposition Theorem for Lock-Freedom

Lock-freedom is a global progress property of a concurrent system which states that at all times throughout an (infinite) execution of the system, eventually one process completes its currently running operation [16]. There are two further important liveness properties [29]: *wait-freedom* requires each invoked operation to eventually complete (thus it is stronger than lock-freedom); *obstruction-freedom* requires completion of every operation that eventually executes in isolation (hence it is a weaker property than lock-freedom). In contrast to lock-freedom, proofs of these properties require no decomposition technique, since they are already process-local. All three properties preclude the standstill (deadlock) of the system but in a lock-free implementation, repeated change of the data structure can force a single process to retry again and again.

In our formal setting (see Section 4.1) - apart from executing infinitely often COP - processes may also execute skip or terminate. Therefore an additional activity flag is required to detect termination of the data structure operation. A process m finishes its current execution of an operation when it resets its activity flag $Act(m)$. In a concurrent system which consists of n processes, global progress P is defined in terms of the activity flags as

$$\begin{aligned} & P(n, Act, Act') \\ \leftrightarrow\ & ((\exists\ m \leq n.\ Act(m)) \rightarrow \Diamond\ (\exists\ k \leq n.\ Act(k) \wedge \neg\ Act'(k))) \end{aligned}$$

That is, if there is at least one active process (m), one of them (k) will eventually reset its activity flag, i.e. complete its operation on the data structure.

To model the absence of interference that forces a process to reiterate, an additional predicate $U : cstate \times cstate$ ("unchanged") is added to the rely-guarantee theory. This predicate must be reflexive, because steps that leave the state unchanged do not interfere with other processes. It is also necessary (for the lifting) to be able to summarize several consecutive steps which satisfy U into one step by transitivity

$$\begin{aligned} & U(CS, CS) \\ & U(CS_0, CS_1) \wedge U(CS_1, CS_2) \rightarrow U(CS_0, CS_2) \end{aligned} \tag{7}$$

Furthermore, we exclude steps from the system's environment which unpredictably change the activity flags or the critical parts of the data structure by extending the global rely condition:

$$\begin{aligned} & R_{ext}(CS', Act', CS'', Act'') \\ \leftrightarrow\ & R(CS', CS'') \wedge Act'' = Act' \wedge U(CS', CS'') \end{aligned}$$

This extension is acceptable, since we assume that only processes within the overall interleaved system are allowed to manipulate these specific resources. Lock-freedom of CSpawn then follows from the following intuitive local proof obligation

$$\begin{aligned} & \mathrm{COP}(m, In; CS, Out), \Box\ I(R_m) \\ \vdash\ & \Box\ (\neg\ U(CS, CS') \vee (\Box\ U(CS', CS'')) \rightarrow \Diamond\ \mathbf{last}) \end{aligned} \tag{8}$$

At any time (leading $\Box$), a lock-free operation that updates the relevant part of the shared state itself in a step ($\neg\ U(CS, CS')$) or encounters no interference ($\Box\ U(CS', CS'')$), eventually terminates ($\Diamond$ **last**).

Properties (7) and (8) together with the rely-guarantee conditions of the previous subsection are sufficient to prove lock-freedom of the overall system, when initially the invariant holds and all activity flags are false.

Theorem 1 (Decomposition Theorem for Lock-Freedom)
If formulas (1) to (8) can be proved (for some Inv, U, R, R_m, G_m), then:

$$\mathrm{CSpawn}(n; \ldots), \Box\ R_{ext}, Inv(CS), \forall\ m \leq n.\ \neg\ Act(m) \vdash \Box\ P(n, Act, Act') \tag{9}$$

Given that the global environment satisfies R_{ext} at all times, the presence of an active operation will always lead to the completion of some (active) operation. Although there are no blocking steps in the queue example, the theorem holds for algorithms COP which include such steps too.

The theorem is proved in two stages. The first stage proves

$$\begin{array}{l} \text{CS}\textsc{eq}(m; \dots), \Box\, I(R_m), \neg\, Act(m) \\ \vdash \Box\, (\quad Act(m) \wedge (\neg\, U(CS, CS') \vee (\Box\, U(CS', CS''))) \\ \qquad \rightarrow \Diamond\, (Act(m) \wedge \neg\, Act'(m))) \end{array} \tag{10}$$

while the second proves the main theorem. Both proofs rely on the fact, that our logic allows to reduce a goal $\alpha \mid\mid \beta$ (resp. $\alpha; \beta$) to $\varphi \mid\mid \beta$ (resp. $\varphi; \beta$), when a lemma $\alpha \vdash \varphi$ is available (see [23] for more details). Note that for interleaving this fact crucially depends on our semantics with alternating system and environment steps. It does not hold in standard temporal logic.

A detailed description of the proofs is beyond the scope of this paper, we just give the main idea of the second proof. The proof of (9), which can be written in the form $\text{CS}\textsc{pawn}(n; \dots) \vdash \varphi(n)$, starts by induction over the number of processes. Lemma (10), which can be written as $\text{CS}\textsc{eq} \vdash \psi$, directly closes the base case. In the induction step, unfolding of $\text{CS}\textsc{pawn}(n + 1; \dots)$ gives an interleaving of CSeq and $\text{CS}\textsc{pawn}(n; \dots)$. The first formula in the interleaving can be replaced with ψ, while the second can be replaced with $\varphi(n)$ by the induction hypothesis. Therefore it remains to prove $\psi \mid\mid \varphi(n) \vdash \varphi(n + 1)$. The main part of the proof is now by induction over $\Box\, P(n+1, Act, Act')$ in $\varphi(n+1)$ and symbolic execution. The proof has a large number of cases, since a symbolic execution step of each of the two formulas ψ and $\varphi(n)$ can terminate (causing the other formula to remain), or do an unblocked or blocked step (the latter forcing a step of the other formula, or a blocked step if both block). Also in each symbolic execution step we have to prove the implication of the progress property for the current state. The proof is more complex than all the proofs of the case study. Since it has to be done once only, it moves much of the complexity of analyzing the lock-freedom property into the generic theory. The proof has been mechanized using KIV and is online [27].

5 Proving Lock-Freedom for the Queue

In this section we present the instantiation of the decomposition theorem for the queue. The presentation is in two parts: first we give the necessary rely and invariant conditions which are a subset of those used for proving linearizability in [11]; second we describe the instantiation of the unchanged predicate and outline the proof of termination for the enqueue operation. Full details are available online [27].

5.1 Rely-Guarantee Conditions and Invariant

The generic operation COP is instantiated with the nondeterministic choice between the two queue operations. The generic state variable *CS* becomes a tuple consisting of a shared state *Hp*, *Head*, *Tail* and local states $\mathit{Newef}(m)$, $\mathit{Tlef}(m)$, $\mathit{Nxtef}(m)$, $\mathit{Succef}(m)$, $\mathit{Hddf}(m)$, $\mathit{Nxtdf}(m)$, $\mathit{Succdf}(m)$ for every process m.

Since all processes execute the same set of operations, all processes will have the same rely condition R_m by symmetry. It claims that the environment step preserves the invariant *Inv* and that predicates $\mathit{Enqlocal}_m$ and $\mathit{Deqlocal}_m$ hold.

$$\begin{aligned} &R_m(CS', CS'') \\ \leftrightarrow\ &(\mathit{Inv}(CS') \rightarrow \mathit{Inv}(CS'')) \wedge \mathit{Enqlocal}_m(CS', CS'') \wedge \mathit{Deqlocal}_m(CS', CS'') \end{aligned}$$

The invariant ensures that there are no dangling pointers and that newly allocated nodes are disjoint from one another and from the queue. It also guarantees that the current state is a *valid* queue representation, i.e. it conforms to one of the variants shown in Figure 1.

Predicate $\mathit{Enqlocal}_m$ specifies that pointer variables $\mathit{Succef}(m)$, $\mathit{Tlef}(m)$ and $\mathit{Nxtef}(m)$ (which were lifted from originally local variables to global ones) are unchanged by other processes

$$\begin{aligned} &\mathit{Succef}''(m) = \mathit{Succef}'(m) \wedge \mathit{Tlef}''(m) = \mathit{Tlef}'(m) \\ \wedge\ &\mathit{Nxtef}''(m) = \mathit{Nxtef}'(m) \end{aligned} \tag{11}$$

The main interesting information necessary to prove lock-freedom is that whenever the snapshot's next pointer is not null, this reference remains untouched by m's environment:

$$\begin{aligned} &\mathit{Tlef}'(m) \neq \mathit{null} \wedge \mathit{Hp}'[\mathit{Tlef}'(m)].\mathrm{nxt} \neq \mathit{null} \\ \rightarrow\ &\mathit{Hp}''[\mathit{Tlef}''(m)].\mathrm{nxt} = \mathit{Hp}'[\mathit{Tlef}'(m)].\mathrm{nxt} \end{aligned} \tag{12}$$

This rely condition is interesting when a process shifts a lagging tail pointer for two reasons: first, to argue that this step maintains a valid queue representation and second, to ensure that it does not violate the unchanged predicate (cf. predicate Id_S in the next subsection). Proof obligation (3) from the rely-guarantee theory implies that this assumption is acceptable. Its proof rewrites the unless formula in the succedent as described in Section 3.2 to extract an inductive argument in case that a loop is reiterated. When symbolically executing the code of a queue operation of an arbitrary process m, it has to be shown that each step preserves m's guarantee condition G_m, given that m's local rely condition was true for the last environment transition. The local guarantee condition G_m (and the global rely condition R) is defined as weak as possible by constraint (1) (resp. (6)) of the rely-guarantee theory. Since proving (3) has also been necessary in our previous work to show that the queue algorithm is linearizable and since the required rely-guarantee conditions from the linearizability-proof were sufficient to prove lock-freedom too, the former proof of (3) has been reused.

Altogether the required rely conditions for lock-freedom of enqueue are a valid queue representation, (11), and (12). Similar assumptions as defined in $\mathit{Enqlocal}_m$ are defined in in $\mathit{Deqlocal}_m$ for the linearizability poof of the dequeuing process. However, for proving lock-freedom of dequeue, only the locality of $\mathit{Succdf}(m)$ and $\mathit{Hddf}(m)$ are required.

5.2 Unchanged Predicate

According to proof obligation (8) a suitable instantiation of predicate U must ensure termination of a process in an environment that respects U at all times and it must be preserved by each program transition, unless a transition eventually leads to completion (e.g. a successful CAS).

That is, when a process dequeues it is sufficient for its termination to assume that the global head pointer remains unchanged by the environment

$$Id_H :\equiv Head'' = Head'$$

When m enqueues, assuming that other processes n will not change the global tail pointer is not sufficient to ensure termination. Suppose a system execution in which m repeatedly shifts the lagging tail for every n which attaches a new node to the queue. In this situation, no other process ever changes the tail pointer, as this is done by m who never completes. Instead, U must ensure that m finally can attach its newly allocated node to the queue, i.e. no other process may add a new node. Two cases are discerned regarding the current representation. If the tail pointer does not lag (its next reference is null) neither the global tail pointer nor its next reference may be changed

$$Id_T :\equiv Tail'' = Tail' \wedge Hp''[Tail''].\mathrm{nxt} = Hp'[Tail'].\mathrm{nxt}$$

When the tail pointer is lagging, m assumes the following environment behavior: other processes leave the tail pointer and its next reference unchanged or they shift the tail to its direct successor node (which has a null next reference)

$$Id_S :\equiv Id_T \vee Tail'' = Hp'[Tail'].\mathrm{nxt} \wedge Hp''[Tail''].\mathrm{nxt} = null$$

Predicate U is the conjunction of these identities:

$$Id_H \wedge (Hp'[Tail'].\mathrm{nxt} = null \rightarrow Id_T) \wedge (Hp'[Tail'].\mathrm{nxt} \neq null \rightarrow Id_S)$$

It specifies that changes relevant for progress are enqueuing or removing an element, while moving a lagging tail does not guarantee progress and can only be done according to Figure 1.

5.3 Proof Outline

The unchanged predicate is reflexive and transitive. The temporal logic proof obligation (8) from Section 4.2 is divided into four subgoals by discerning which operation is currently executed (enqueue or dequeue) and splitting the disjunction in the succedent to distinguish whether a local transition of the current process changes the data structure or the environment satisfies the unchanged property at all times.[2] For enqueue we get two proof obligations

$$\begin{array}{l}\mathbf{E1}, \Box\ I(R_m) \vdash \Box\ (\neg\ U(CS, CS') \rightarrow \Diamond\ \mathbf{last}) \\ \mathbf{E1}, \Box\ I(R_m) \vdash \Box\ (\Box\ U(CS', CS'') \rightarrow \Diamond\ \mathbf{last})\end{array} \qquad (13)$$

[2] As the interleaving operator is not used in proof obligation (8), its proof is independent from the underlying scheduler. Scheduling issues are covered in the lifting proof of the decomposition theorem only.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{E8}\ldots,S_1\vdash\ldots}{\cdots\neq \mathit{null},\ \mathit{Tail}=\mathit{Tlef}(m)\vdash\ldots}\,(4) \qquad \cdots = \mathit{null}\vdash\ldots}{\mathbf{E7},\ldots,S_0\vdash\ldots}\,(3)}{\ldots \mathit{Hp}[\mathit{Tail}].\mathrm{nxt}\neq\mathit{null}\vdash\ldots}\,(2) \qquad \cdots=\mathit{null}\vdash\ldots}{\mathbf{E6},\ \mathit{VRU}\vdash\Diamond\,\mathbf{last}}\,(1)$$

$$\begin{aligned}
\mathit{VRU} &:\equiv \Box\ (\mathit{valid}(\mathit{Head},\mathit{Tail},\mathit{Hp}) \wedge (11) \wedge (12) \wedge U(CS', CS''))\\
S_0 &:\equiv \mathit{Tlef}(m) \neq \mathit{null}\\
S_1 &:\equiv \mathit{Tlef}(m) \neq \mathit{null} \wedge \mathit{Nxtef}(m) \neq \mathit{null} \wedge \mathit{Hp}[\mathit{Tlef}(m)].\mathrm{nxt} = \mathit{Nxtef}(m)
\end{aligned}$$

Fig. 4. Proof outline enqueue lock-free

where **Ek** denotes the remaining program starting from line Ek , e.g. and $\mathbf{E1} \equiv \mathit{CEnq}$ and $\mathbf{E12} \equiv \mathrm{CAS}(\mathit{Tle}, \mathit{Nxte}; \mathit{Tail}); \mathbf{while}\ \neg\ \mathit{SuccE}\ \mathbf{do}\ \ldots$ (these abbreviations are not used in KIV). The first is rather simple, since the only step with $\neg\ U(CS, CS')$ is a succeeding CAS at line E10 which sets the loop-flag to true, so the algorithm terminates after the final step E13.

The second proof is more challenging. It consists of an induction for the leading always operator and symbolically executing the enqueue operation until it either terminates or the induction hypothesis can be applied. During execution we get a side goal for every step: starting from the considered step, formula $\Box\ U(CS', CS'')$ must lead to termination. This can be proved by stepping to the start of the loop (instruction E5) and applying the following lemma

$$\mathbf{E5}, \Box\ I(R_m) \vdash \Box\ U(CS', CS'') \to \Diamond\ \mathbf{last}$$

which states that the additional environment assumption $\Box\ U(CS', CS'')$ is sufficient to guarantee termination of the loop of the enqueue operation.

Its proof needs no induction, but requires stepping through the loop once or twice, depending on whether the tail is lagging when the snapshot is taken; the basic idea is illustrated in Figure 4. In the conclusion of the proof tree, the first symbolic execution step to enter the while loop has already been executed. The remaining program is **E6** (instruction E6 takes the local snapshot $\mathit{Tlef}(m)$). In a valid state, the required rely conditions and the unchanged predicate are assumed to hold at all times (VRU); no further restrictions on the current state are necessary to prove termination of the loop. Proof step (1) is a case distinction on whether the current queue has a lagging tail pointer ($\mathit{Hp}[\mathit{Tail}].\mathrm{nxt} \neq \mathit{null}$). If the tail pointer is not lagging (second premise, right hand side) no further interference will hinder m to complete according to VRU, i.e. the proof consists of executing **E6** until completion. If the tail pointer is lagging behind (first premise, left hand side), proof step (2) symbolically executes the instruction at E6 (followed by an environment transition) which yields the new state S_0 and the remaining program is **E7**. Case distinction (3) tests whether the environment has helped m according to predicate U by shifting the lagging tail pointer (second premise). If this is true, the current proof obligation can be discarded by symbolic execution until the remaining program is again **E6** and using the second premise of proof step (1) as a lemma (during these symbolic execution steps - the test at E8 is false - the tail pointer and its next reference null remain unchanged). If however the tail is still lagging (first premise of proof step (3)) the snapshot is accurate, i.e. $Tail = Tlef(m)$, and the proof continues with symbolic execution of E7 (proof step (4)). In the new state S_1, the snapshot's next reference is $Nxtef(m)$ which is not null. We proceed analogously discerning whether the tail pointer is lagging and symbolic execution: at the latest when the CAS transition at E12 is (successfully) executed, a non-lagging tail representation is established and the second premise of step (1) can eventually be used again as a lemma to finish the proof.

Proving the analog properties to (13) for dequeue is straightforward. The locality assumptions (for the loop-flag and the snapshot) from the rely condition and knowing that the head pointer always remains unchanged according to U, imply termination. This is because after the snapshot is taken, the CAS at D12 will be successfully executed: it is the only dequeue step that does not satisfy the unchanged predicate, but it guarantees progress.

6 Related Work

The analysis of non-blocking algorithms is a current and highly active field of research. Several techniques have been proposed to prove correctness and liveness of these algorithms.

With respect to linearizability, Doherty et al. [7] were the first to publish a formal verification of the queue algorithm (including memory reuse and version numbers to avoid an ABA-problem) based on refinement of IO automata. In contrast to our approach, program counters and a global simulation relation are used to mechanize the proofs using PVS. Since single steps of a concrete algorithm are refined individually, an intermediate automaton and backward simulation had to be used to complete the formal proof for the dequeue operation, while our approach verifies trace inclusion directly avoiding backward simulation (see [11] for details).

Vafeiadis [30] also proves linearizability of the queue. His proof technique is closer to ours in also using rely-guarantee reasoning. A major difference is that his approach is based on adding abstract ghost code to the implementation, and not on refinement. To solve the problem of the dequeue operation, the use of a prophecy variable is suggested (which is basically equivalent to the use of backward simulation).

Many other groups have contributed to the verification of non-blocking algorithms. Groves et al. [8] for instance present the verification of linearizability of a more complex lock-free implementation based on trace reduction. Our approach is currently not able to formally handle these kind of (elimination) algorithms, where the linearization of an operation can be part of the execution of another process. Gao et al. [31] have described the verification of a lock-free hash table which took more than two man years of work.

A rather different approach is taken by Yahav et al. [32] using shape analysis [33]. The approach assumes that the abstract operations - although atomic - already work on the low level heap and that only their interleaving has to be shown correct. Therefore it compares the intermediate heaps that occur during interleaved execution of the algorithms to the structures at the beginning and the end and keeps track of the differences by a finite abstraction ("delta heap abstraction") to verify linearizability.

The third author has also contributed to Derrick et al. [34]. The approach given there is rather different: it is based on the Z specification language and requires program counters to encode steps of the algorithm as Z operations. Instead of rely-guarantee reasoning, Owicki-Gries [35] like proof obligations are generated. The approach is the only one we are aware of, that proves linearizability formally using the original definition of [15]. All other approaches (including ours in [11]) argue informally that linearizability holds.

Related to lock-freedom, we are aware only of two approaches: Colvin and Dongol [12,13] describe the verification of several lock-free implementations (including an array-based nonblocking queue [36]) by explicitly constructing a well-founded order on program counters and proving that each action either guarantees progress or reduces the value of the state according to the well-founded order. They identify progress actions, which correspond to those steps where our predicate U is false. Constructing a well-founded order is unnecessary in our approach, since it is implicit in stepping through the program.

A higher degree of automation is achieved by Gotsman et al. [14] based on rely-guarantee reasoning and techniques like shape analysis and separation logic [37]. Their approach can verify proof obligations that imply lock-freedom for several non-trivial algorithms automatically, using a combination of several tools. Derivation of these proof obligations however is done on paper. There are several differences in the proof obligations too: our approach does not use a reduction of CSpawn to a spawning procedure where the call to CSeq is replaced by COP (which needs some assumptions about symmetry to be correct). Our proof obligation ensures that the algorithm terminates after a step which falsifies U, while their proof obligation requires that no process can execute steps which change the data structure infinitely often. A close comparison for the queue example is hard, since the queue is only mentioned as one of the examples automatically provable.

Both related approaches assume potentially unfair scheduling, which is more adequate than our assumption of weak fairness. A closer analysis shows that we need fairness only to prove that a process is not suspended in favor of another process which executes skip steps only. Both related approaches consider processes which execute an infinite loop of calls to COP and no other instructions. If we replace the implementation of CSeq with such a loop, the fair interleaving operator can be replaced with an unfair one. We prefer the more general formalization of CSeq, since it is realistic that a process executes other statements or terminates rather than just calling COP repeatedly. Nevertheless we have mechanized a version for this loop with unfair interleaving too. For simplicity, the current proof is limited to algorithms without blocking steps.

The proof proceeds much like the original one, since the symbolic execution rules for non-fair interleaving are the same as for fair interleaving. The main difference is that without weak fairness, it can no longer be guaranteed that the first of two interleaved processes will do a step eventually. Instead, an additional case split is necessary which gives the same goal as for weak fairness, plus an extra goal for the case where the first process is never scheduled again, so only the second remains. This proof is available online [27] too.

7 Summary

We have described a decomposition theorem that reduces the proof of the global property lock-freedom to process-local proof obligations and we have shown how this theorem can be applied to prove lock-freedom of a non-trivial lock-free queue implementation. All specifications and proofs are fully mechanized in the interactive theorem prover KIV and the main proofs of lock-freedom in the queue case study are highly automated. The theory shares rely-guarantee conditions with those necessary to prove linearizability. We believe that our technique closely follows the intuitive arguments necessary to prove lock-freedom.

In future work we will consider the ABA-problem in an additional refinement step (similar to [8]), by extending the current implementation with reference-recycling and version numbers. Moreover, we will try to improve our method by better exploiting the symmetry of typical lock-free implementations in the rely-guarantee theory and by including a formal definition of linearizability within the reduction approach (similar to [34]).

Acknowledgements

We want to thank the reviewers for their constructive remarks, in particular on the weak fairness assumption, which have helped to improve this paper.

References

1. Treiber, R.K.: System programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center (1986)
2. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: SPAA '04: ACM symposium on Parallelism in algorithms and architectures, pp. 206–215. ACM Press, New York (2004)
3. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. 15th ACM Symp. on Principles of Distributed Computing, pp. 267–275 (1996)
4. Michael, M.M.: Cas-based lock-free algorithm for shared deques. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 651–660. Springer, Heidelberg (2003)
5. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA 2002, pp. 73–82. ACM, New York (2002)

6. Doherty, S., Detlefs, D.L., Groves, L., Flood, C.H., Luchangco, V., Martin, P.A., Moir, M., Shavit, N., Steele Jr., G.L.: Dcas is not a silver bullet for nonblocking algorithm design. In: SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, pp. 216–224. ACM, New York (2004)
7. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)
8. Groves, L., Colvin, R.: Trace-based derivation of a scalable lock-free stack algorithm. Formal Aspects of Computing (FAC) 21(1-2), 187–223 (2009)
9. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 129–136. ACM, New York (2006)
10. Gao, H., Hesselink, W.H.: A formal reduction for lock-free parallel algorithms. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 305–309. Springer, Heidelberg (2004)
11. Bäumler, S., Schellhorn, G., Tofan, B., Reif, W.: Proving linearizability with temporal logic. In: Formal Aspects of Computing (FAC), (2009), `http://www.springerlink.com/content/7507m59834066h04/`
12. Colvin, R., Dongol, B.: Verifying lock-freedom using well-founded orders. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 124–138. Springer, Heidelberg (2007)
13. Colvin, R., Dongol, B.: A general technique for proving lock-freedom. Sci. Comput. Program. 74(3), 143–165 (2009)
14. Gotsman, A., Cook, B., Parkinson, M., Vafeiadis, V.: Proving that nonblocking algorithms don't block. In: Principles of Programming Languages, pp. 16–28. ACM, New York (2009)
15. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
16. Massalin, H., Pu, C.: A lock-free multiprocessor os kernel. SIGOPS Oper. Syst. Rev. 26(2), 108 (1992)
17. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP'83, pp. 321–332. North-Holland, Amsterdam (1983)
18. Misra, J.: A reduction theorem for concurrent object-oriented programs. In: McIver, A., Morgan, C. (eds.) Programming methodology, pp. 69–92. Springer, New York (2003)
19. Moszkowski, B.: Executing Temporal Logic Programs. Cambridge University Press, Cambridge (1986)
20. Cau, A., Moszkowski, B., Zedan, H.: ITL – Interval Temporal Logic. Software Technology Research Laboratory, SERCentre, De Montfort University, The Gateway, Leicester LE1 9BH, UK (2002), `http://www.cms.dmu.ac.uk/~cau/itlhomepage`
21. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction—A Basis for Applications. Systems and Implementation Techniques, vol. II, pp. 13–39. Kluwer Academic Publishers, Dordrecht (1998)
22. Gao, H., Groote, J.F., Hesselink, W.H.: Lock-free parallel and concurrent garbage collection by mark&sweep. Sci. Comput. Program. 64(3), 341–374 (2007)

23. Bäumler, S., Balser, M., Nafz, F., Reif, W., Schellhorn, G.: Interactive verification of concurrent systems using symbolic execution. AI Communications 23(2-3), 285–307 (2010)
24. Balser, M.: Verifying Concurrent System with Symbolic Execution. Shaker Verlag, Germany (2006)
25. Collette, P., Knapp, E.: Logical foundations for compositional verification and development of concurrent programs in unity. In: Alagar, V.S., Nivat, M. (eds.) AMAST 1995. LNCS, vol. 936, pp. 353–367. Springer, Heidelberg (1995)
26. Roever, W.P.D., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press, Cambridge (2001)
27. Online Presentation of the KIV-specifications and the Verification of the Queue (and Stack), `http://www.informatik.uni-augsburg.de/swt/projects/lock-free.html`
28. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems (1995)
29. Dongol, B.: Formalising progress properties of non-blocking programs. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 284–303. Springer, Heidelberg (2006)
30. Vafeiadis, V.: Modular fine-grained concurrency verification. PhD thesis, University of Cambridge (2007)
31. Gao, H., Groote, J.F., Hesselink, W.H.: Lock-free dynamic hash tables with open addressing. Distrib. Comput. 18(1), 21–42 (2005)
32. Amit, D., Rinetzky, N., Reps, T.W., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007)
33. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
34. Derrick, J., Schellhorn, G., Wehrheim, H.: Proving linearizability via non-atomic refinement. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 195–214. Springer, Heidelberg (2007)
35. Owicki, S.S., Gries, D.: An Axiomatic Proof Technique for Parallel Programs I. Acta Inf. 6, 319–340 (1976)
36. Colvin, R., Groves, L.: Formal verification of an array-based nonblocking queue. In: ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, Washington, DC, USA, pp. 507–516. IEEE Computer Society Press, Los Alamitos (2005)
37. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, p. 1. Springer, Heidelberg (2001)

Gradual Refinement
Blending Pattern Matching with Data Abstraction

Meng Wang[1], Jeremy Gibbons[1], Kazutaka Matsuda[2], and Zhenjiang Hu[3]

[1] Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{menw,jg}@comlab.ox.ac.uk
[2] Graduate School of Information Sciences, Tohoku University
Aramaki aza Aoba 6-3-09, Aoba-ku, Sendai-city, Miyagi-pref. 980-8579, Japan
kztk@kb.ecei.tohoku.ac.jp
[3] GRACE Center, National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
hu@nii.ac.jp

Abstract. Pattern matching is advantageous for understanding and reasoning about function definitions, but it tends to tightly couple the interface and implementation of a datatype. Significant effort has been invested in tackling this loss of modularity; however, decoupling patterns from concrete representations while maintaining soundness of reasoning has been a challenge. Inspired by the development of invertible programming, we propose an approach to abstract datatypes based on a right-invertible language RINV—every function has a right (or pre-) inverse. We show how this new design is able to permit a smooth incremental transition from programs with algebraic datatypes and pattern matching, to ones with proper encapsulation (implemented as abstract datatypes), while maintaining simple and sound reasoning.

1 Introduction

1.1 Program Development

Suppose that you are developing a program involving some data structure. You don't yet know which operations you will need on the data structure, or what efficiency constraints you will impose on those operations. Instead, you want to prototype the program, and conduct some initial experiments on the prototype; on the basis of the results from those experiments, you will decide whether a naive representation of the data structure suffices, or whether you need to choose a more sophisticated implementation. In the latter case, you do not want to have to conduct major surgery on your prototype in order to refactor it to use a different representation.

The traditional solution to this problem is to use data abstraction: identify (or evolve) an interface for the abstract datatype, program to that interface, and allow the implementation to vary without perturbing the program. However, that requires you to prepare in advance for the possible change of representation:

it doesn't provide a smooth revision path if you didn't have the foresight to introduce the interface in the first place, but used a bare algebraic datatype as the representation.

Moreover, choosing a naive representation in terms of an algebraic datatype has considerable attractions. Programs that manipulate the data can be defined using *pattern matching* over the constructors of the datatype, rather than having to use 'observer' operations on a data abstraction. This leads to a concise and elegant programming style, which being based on equations is especially convenient for reasoning about program behaviour [42].

1.2 Pattern Matching

As a simple example, consider encoding binary numbers as lists of bits, most significant first:

```
data Bin = Zero | One
type Num = [Bin]
```

Functions are typically defined by pattern matching. Consider normalizing binary numbers by eliding leading zeroes.

```
normal :: Num → Num
normal []           = []            -- clause (1)
normal (One : num)  = One : num     -- clause (2)
normal (Zero : num) = normal num    -- clause (3)
```

The definition forms a collection of equations, which give a straightforward explanation of the operational behaviour of the function:

```
  normal [Zero, One, Zero]
≡   { clause (3) }
  normal [One, Zero]
≡   { clause (2) }
  [One, Zero]
```

They are also convenient for calculation; for example, here is one case of an inductive proof that *normal* is idempotent:

```
  normal (normal (Zero : num))
≡   { clause (3) }
  normal (normal num)
≡   { inductive hypothesis }
  normal num
≡   { clause (3) }
  normal (Zero : num)
```

An equivalent definition without using pattern matching is harder to read:

$$\begin{aligned}&\mathit{normal} :: \mathit{Num} \to \mathit{Num}\\&\mathit{normal}\ \mathit{num} = \textbf{if}\ \mathit{null}\ \mathit{num} \vee \mathit{one}\ (\mathit{head}\ \mathit{num})\ \textbf{then}\ \mathit{num}\\&\qquad\qquad\qquad\ \textbf{else}\ \mathit{normal}\ (\mathit{tail}\ \mathit{num})\end{aligned}$$

It is also much less convenient for calculating with.

Pattern matching has accordingly been supported as a standard feature in most modern functional languages, since its introduction in Hope [7]. More recently, it has started gaining recognition from the object-oriented community [9, 28, 24] too. Unfortunately, the appeal of pattern matching wanes when we need to change the implementation of a data structure: function definitions are tightly coupled to a particular representation, and a change of representation has a far-reaching effect. As a result, it has been observed that the wide spread of pattern matching "leads to a discontinuity in programming: programmers initially use pattern matching heavily, and are then forced to abandon the technique in order to regain abstraction over representations" [39].

1.3 Our Contribution

In this work, then, we strive to address the tension between the convenience of pattern matching and the flexibility of data abstraction by proposing a mechanism to allow programs written with pattern matching to be refactored smoothly and incrementally into ones with abstract datatypes (ADTs) [23], without losing the benefits of simple equational reasoning. In particular, we:

– propose the use of definitions with pattern matching as constructive specifications of ADTs;
– devise an equational reasoning framework for both the primitives of and the user-defined operations on ADTs;
– identify necessary and sufficient conditions for correctness of such equational reasoning;
– design a right-invertible language RINV that guarantees these conditions by construction.

For the sake of demonstration, we explain our proposal using Haskell; but any language providing algebraic datatypes would work just as well.

The rest of the paper is structured as follows. Section 2 gives a brief introduction to ADT specification methods. Section 3 presents our proposed design for pattern matching with ADTs, and Section 4 provides a formal definition of the right-invertible language RINV on which our design is based. We then evaluate the performance and explore alternative points in the design space of our system (Section 5), before discussing related work (Section 6) and concluding (Section 7).

2 ADTs and Their Specification

By definition, an ADT is characterized not by its representation or implementation, but by its interface: a fixed set of primitive operations, together with a specification of their semantics. Different styles of specification possess different strengths and weaknesses, which makes them more or less suitable as refactoring targets from programs defined with algebraic datatypes and pattern matching. In this section, we briefly discuss two popular ways of defining the semantics, namely axiomatic and constructive.

An *axiomatic specification* is implicit: the behaviour of the operations is defined by relating them to each other by a collection of equations. For example, consider operations suitable for queue structures:

```
adt Queue a
  emptyQ  :: Queue a
  enQ     :: a → Queue a → Queue a
  deQ     :: Queue a → Queue a
  first   :: Queue a → a
  isEmpty :: Queue a → Bool
```

The following axioms are sufficient to specify the semantics:

```
deQ (enQ a emptyQ) ≡ emptyQ
deQ (enQ a q)      ≡ enQ a (deQ q) ⇐ isEmpty q ≡ False
first (enQ a emptyQ) ≡ a
first (enQ a q)    ≡ first q       ⇐ isEmpty q ≡ False
isEmpty emptyQ     ≡ True
isEmpty (enQ a q)  ≡ False
```

(All the free variables above are assumed to be universally quantified over well-defined terms, and equality takes precedence over implication.) If the ADT is implemented in a 'faithful' manner [40], the above specification is all that is known to users, and any properties of programs using the datatype should be derived only from these axioms. The axiomatic approach avoids suggesting any particular representation, and so provides a high degree of abstraction. On the other hand, axiomatic specifications are not easy to construct.

As an alternative, a *constructive specification* explicitly defines the semantics of operations by expressing them in terms of an underlying model. For example, the queue ADT can be related to the familiar list model:

```
emptyQ  = emptylist
isEmpty = isNull
deQ     = tail
enQ a q = append q (wrap a)
first   = head
```

(where *wrap* turns an element into a singleton list). The list model can be seen as another ADT that is sufficiently powerful to simulate the queue ADT. Apparently, the constructive approach makes the specifications easier to write and to understand. The underlying model can be further instantiated into a representation as an algebraic datatype (also known as a *typical object* in the literature [22]):

$$\textbf{data}\ \mathit{Queue}\ a = \mathit{None} \mid \mathit{More}\ a\ (\mathit{Queue}\ a)$$

which results in the following specifications:

$$\begin{aligned}
&\mathit{emptyQ} &&= \mathit{None}\\
&\mathit{first}\ (\mathit{More}\ a\ q) &&= a\\
&\mathit{isEmpty}\ \mathit{None} &&= \mathit{True}\\
&\mathit{isEmpty}\ x &&= \mathit{False}\\
&\mathit{enQ}\ a\ \mathit{None} &&= \mathit{More}\ a\ \mathit{None}\\
&\mathit{enQ}\ a\ (\mathit{More}\ x\ q) &&= \mathit{More}\ x\ (\mathit{enQ}\ a\ q)\\
&\mathit{deQ}\ (\mathit{More}\ a\ q) &&= q
\end{aligned}$$

It it worth emphasising that the datatype acts only as a model of the ADT: it may suggest but it does not imply a particular implementation. We also note that this constructive approach does not cover all ADTs: for example, unordered sets cannot be fully modelled by an algebraic datatype.

Whether the behaviour of an ADT is specified axiomatically or constructively, the specification can be used in reasoning about programs that make use of the ADT. In particular, one can use the model underlying a constructive specification to infer properties of an implementation; for example, we can easily recover the axiom $\mathit{deQ}\ (\mathit{enQ}\ a\ q) \equiv \mathit{enQ}\ a\ (\mathit{deQ}\ q) \Leftarrow \mathit{isEmpty}\ q \equiv \mathit{False}$ with the following derivation.

$$\begin{aligned}
&\quad \mathit{deQ}\ (\mathit{enQ}\ a\ (\mathit{More}\ b\ q))\\
\equiv&\quad \{\, \mathit{enQ}\,\}\\
&\quad \mathit{deQ}\ (\mathit{More}\ b\ (\mathit{enQ}\ a\ q))\\
\equiv&\quad \{\, \mathit{deQ}\,\}\\
&\quad \mathit{enQ}\ a\ q\\
\equiv&\quad \{\, \mathit{deQ}\,\}\\
&\quad \mathit{enQ}\ a\ (\mathit{deQ}\ (\mathit{More}\ b\ q))
\end{aligned}$$

3 Reasoning with Constructive Specifications

Our purpose is to allow incremental refactoring of a program, replacing an algebraic datatype used with pattern matching by a more sophisticated ADT implementation.

As described in Section 2, a typical approach to specifying an ADT is to use an algebraic datatype as its model and the definitions by pattern matching for constructive specifications of the operations. In moving from an initial program explicitly depending on an algebraic datatype to a refactored one using an ADT, one will generally have to reimplement some of the existing functions as primitive operations of the new ADT, and rewrite the remaining functions in terms of these primitives.

There are two problems with this process. Firstly, it is a ‘big bang’ refactoring: all uses of the original algebraic datatype have to be changed at once, even though some of the old definitions may not gain from the refactoring. Secondly, it loses the benefits of pattern matching for the functions that have to be redefined in terms of the ADT primitives: it is no longer so convenient for reasoning.

In this section, we propose a framework free from the above pitfalls: refactoring can be done selectively; and at any point in the process, executability and reasoning are fully supported. We look into the details of the design by means of examples.

3.1 A First Example: FIFO Queue

The queue ADT we have seen is defined via the following specification.

$$\begin{array}{l}
\mathbf{adt}\ \mathit{Queue}\ a = \mathit{None} \mid \mathit{More}\ a\ (\mathit{Queue}\ a) \\
\quad \begin{array}{ll}
\mathit{emptyQ} & = \mathit{None} \\
\mathit{first}\ (\mathit{More}\ a\ q) & = a \\
\mathit{isEmpty}\ \mathit{None} & = \mathit{True} \\
\mathit{isEmpty}\ x & = \mathit{False} \\
\mathit{enQ}\ a\ \mathit{None} & = \mathit{More}\ a\ \mathit{None} \\
\mathit{enQ}\ a\ (\mathit{More}\ x\ q) & = \mathit{More}\ x\ (\mathit{enQ}\ a\ q) \\
\mathit{deQ}\ (\mathit{More}\ a\ q) & = q
\end{array}
\end{array}$$

This looks similar to an algebraic datatype declaration, but the right-hand side of the definition introduces a model, instead of an implementation, of the ADT. The primitive operations of the ADT are specified in term of this model; despite having a previous life as an executable function, each specification now serves only to express semantics, and is to be replaced by a corresponding ADT implementation at run-time.

As an example, the *enQ* declaration should now be interpreted as a specification:

$$\begin{array}{lll}
\mathit{enQ}\ a\ q & \equiv \mathit{More}\ a\ \mathit{None} & \Leftarrow q \equiv \mathit{None} \\
\mathit{first}\ (\mathit{enQ}\ a\ q) & \equiv \mathit{first}\ q \wedge \mathit{deQ}\ (\mathit{enQ}\ a\ q) \equiv \mathit{enQ}\ a\ (\mathit{deQ}\ q) & \Leftarrow q \not\equiv \mathit{None}
\end{array}$$

rather than as a concrete implementation.

Now let's consider a possible concrete representation of queue structures:

$$\mathbf{type}\ \mathit{Queue}\ a = ([a],[a])$$

The second list of the pair, representing the latter part of a queue, is reversed, so that enqueuing simply prefixes an element onto it. The primitive operations can be implemented as follows:

$$\begin{array}{ll}
\mathit{emptyQ\_} & = ([\,],[\,]) \\
\mathit{first\_}\ ([\,], bq) & = \mathit{last}\ bq \\
\mathit{first\_}\ ((a : fq), bq) & = a \\
\mathit{isEmpty\_}\ ([\,],[\,]) & = \mathit{True} \\
\mathit{isEmpty\_}\ q & = \mathit{False} \\
\mathit{enQ\_}\ a\ (fq, bq) & = (fq, a : bq) \\
\mathit{deQ\_}\ ([\,], bq)) & = \mathit{deQ\_}\ (\mathit{reverse}\ bq, [\,]) \\
\mathit{deQ\_}\ (a : fq, bq) & = (fq, bq)
\end{array}$$

(We use the naming convention of adding an underscore "_" to an operation's name to distinguish its implementation from its specification.) The implementations are what really execute when ADTs are used. Since at the moment, we only reimplemented the primitive operations, queues can still be constructed using the model. During execution, any value constructed in the model is firstly converted to a value in the implementation before being passed to the implemented operations; and the resulting output is converted back to the model. For example, given the program $enQ\ 1\ None$, what really executes is $(\mathsf{to} \circ enQ\ 1 \circ \mathsf{from})\ None$, where the two functions to and from convert the abstract representation of a queue into the model and back again. For the case of queue ADT, the to function can be defined as follows:

$$\begin{aligned}
&\mathsf{to}\ ([\,],[\,]) &&= None\\
&\mathsf{to}\ ([\,],q) &&= \mathsf{to}\ (reverse\ q,[\,])\\
&\mathsf{to}\ (x:xs,q) &&= More\ x\ (\mathsf{to}\ (xs,q))
\end{aligned}$$

The from function should be the right inverse of to—that is, $\mathsf{to} \circ \mathsf{from} \equiv id$.

The operations will not always have types as simple as $Queue\ a \to Queue\ a$, like $enQ\ 1$ does. Suppose we have polymorphic model datatype $M\ \ a$ and abstract implementation $N\ \ a$, and polymorphic conversion functions $\mathsf{to} :: N\ \ a \to M\ \ a$ and $\mathsf{from} :: M\ \ a \to N\ \ a$. In the general case, a model function will take not just a single value in the model (of type $M\ \ a$), but some combination of model values and other arguments. We capture this in terms of an operation $\mathcal{F}$ on polymorphic datatypes M. Similarly, the operation will return a different combination $\mathcal{G}$ of model values and other results.

Technically, if polymorphic datatypes are represented as functors, then $\mathcal{F},\mathcal{G}$ are functors on the functor category (what Martin *et al.* [26] call 'higher-order functors', or 'hofunctors' for short), so that $\mathcal{F}\ \ M$ and $\mathcal{G}\ \ M$ are themselves functors. Then the model function f will have type $\mathcal{F}\ \ M\ \ a \to \mathcal{G}\ \ M\ \ a$, and the corresponding implementation function $f\_ :: \mathcal{F}\ \ N\ \ a \to \mathcal{G}\ \ N\ \ a$ should satisfy the correctness condition $\mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \equiv f$, as shown in the following commuting diagram.

$$\begin{tikzcd}
\mathcal{F}\ N\ a \arrow[d, "f\_"'] & \mathcal{F}\ M\ a \arrow[l, "\mathcal{F}\ \mathsf{from}"'] \arrow[d, "f"] \\
\mathcal{G}\ N\ a \arrow[r, "\mathcal{G}\ \mathsf{to}"] & \mathcal{G}\ M\ a
\end{tikzcd}$$

Given the right-inverse property, we can simplify the proof obligation

$$\mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \equiv f$$

to the *promotion* condition [3]

$$\mathcal{G}\ \mathsf{to} \circ f\_ \equiv f \circ \mathcal{F}\ \mathsf{to}$$

which does not involve the function from, as the following lemma demonstrates.

Lemma 1 (Round trip). *Given* $\mathcal{G}\ \mathsf{to} \circ f_{\_} \equiv f \circ \mathcal{F}\ \mathsf{to}$*, we have* $f \equiv \mathcal{G}\ \mathsf{to} \circ f_{\_} \circ \mathcal{F}\ \mathsf{from}$*.*

Proof

$$\begin{array}{ll} & \mathcal{G}\ \mathsf{to} \circ f_{\_} \circ \mathcal{F}\ \mathsf{from} \\ \equiv & \{\ \mathcal{G}\ \mathsf{to} \circ f_{\_} \equiv f \circ \mathcal{F}\ \mathsf{to}\ \} \\ & f \circ \mathcal{F}\ \mathsf{to} \circ \mathcal{F}\ \mathsf{from} \\ \equiv & \{\ \mathcal{F} \text{ is a hofunctor}\ \} \\ & f \circ \mathcal{F}\ (\mathsf{to} \circ \mathsf{from}) \\ \equiv & \{\ \mathsf{to} \circ \mathsf{from} \equiv id\ \} \\ & f \circ \mathcal{F}\ id \\ \equiv & \{\ \mathcal{F} \text{ is a hofunctor}\ \} \\ & f \end{array}$$

□

For example, for the operation $\mathit{first} :: \mathit{Queue}\ a \to a$ of the queue ADT, $\mathcal{F}$ is the identity hofunctor, matching the source type $\mathit{Queue}\ a$, and $\mathcal{G}$ is the constantly-identity hofunctor ($\mathcal{G}\ F = \mathit{Id}$), matching the target type a. The operation must satisfy the following promotion condition:

$$\mathit{first}_{\_} \equiv \mathit{first} \circ \mathsf{to}$$

The promotion equations for the rest of the operations are listed below.

$$\begin{array}{ll} \mathsf{to} \circ \mathit{deQ}_{\_} & \equiv \mathit{deQ} \circ \mathsf{to} \\ \mathsf{to} \circ \mathit{enQ}_{\_}\ a & \equiv \mathit{enQ}\ a \circ \mathsf{to} \\ \mathit{isEmpty}_{\_} & \equiv \mathit{isEmpty} \circ \mathsf{to} \\ \mathsf{to}\ \mathit{emptyQ}_{\_} & \equiv \mathit{emptyQ} \end{array}$$

The proofs of such equations follow by standard equational reasoning.

The astute reader may have noticed that we have avoided explicitly defining the from function. This is because validating user-defined to and from with traditional methods requires additional machinery not available within most mainstream languages, such as Haskell or ML. Instead, we explore a correctness-by-construction technique: in the next section, we will present a combinator-based language RINV implemented as a library in Haskell, in which every definable function gets a right inverse for free. That is to say, the ADT implementer writes only to, in RINV, and the corresponding from function is automatically generated. For the sake of completeness, in the case of the queue ADT presented above, a possible definition in RINV reads:

$$\mathsf{to} = \mathit{fold}\ (\mathit{none} \triangledown \mathit{more}) \circ \mathit{app} \circ (\mathit{id} \times \mathit{reverse})$$

However, the details of the language are completely orthogonal to the discussion in this section, and can be safely ignored for the time being.

In summary, other than the conventional expectation that implementations of ADTs are certified against the specifications, the only additional requirement on the implementer is the definition of the to function. Once this is done, an

ADT is sealed off; users of the ADT only interact with the model. They can expect to reason about their programs faithfully using properties of the model; for example, exactly the same reasoning on the model as shown at the end of Section 2 allows us to conclude that

$$deQ\ (enQ\ a\ q) \equiv enQ\ a\ (deQ\ q) \Leftarrow isEmpty\ q \equiv False$$

As mentioned before, a programmer using the ADT now has the choice of keeping the original definitions using pattern matching against the model, or of refactoring them into the traditional style using only the primitive operations of the ADT, or of having a mixture of the two. For example, it is very convenient to define a map function in terms of the list-like model:

```
mapQ1 :: (a → b) → Queue a → Queue b
mapQ1 f None        = None
mapQ1 f (More a q) = More (f a) (mapQ1 f q)
```

or a prioritisation function, which is essentially a stable sort based on element weight:

```
prioritise :: Ord a ⇒ Queue a → Queue a
prioritise None          = None
prioritise (More x xs) = insert x (prioritise xs)
   where insert y None          = More y None
         insert y (More x xs) = if y ⩽ x then More y (More x xs)
                                  else       More x (insert y xs)
```

A definition using only the primitive operations is likely to be more clumsy:

```
mapQ2 :: (a → b) → Queue a → Queue b
mapQ2 f q = mapQ2acc f q emptyQ
   where mapQ2acc f q accq =
     if isEmpty q then accq
     else mapQ2acc f (deQ q) (enQ (f (first q)) accq)
```

Nevertheless, since the non-primitive functions and the primitive operation specifications are based on the same model, we can prove the equivalence of the two versions through equational reasoning.

Definitions with pattern matching are almost always more elegant [42]. However, from time to time, we may want to use the primitive operations for the sake of efficiency, or for reuse of legacy libraries. For example, consider a circular queue that is read for a certain amount of time, say repeatedly playing a piece of music. Using the primitive *enQ* operation allows us to take advantage of its constant time performance.

```
play1 :: Time → Queue (IO ()) → IO ()
play1 0 q         = first q
play1 (n + 1) q = do hd
```

$$\begin{array}{l} \qquad\qquad\qquad\quad \mathit{play1}\ n\ (\mathit{enQ}\ \mathit{hd}\ \mathit{tl}) \\ \quad \textbf{where}\ \mathit{hd} = \mathit{first}\ q \\ \qquad\qquad \mathit{tl} = \mathit{deQ}\ q \end{array}$$

The two styles of programming can be mixed:

$$\begin{array}{lll} \mathit{play2}\ 0 & (\mathit{More}\ a\ q) = a & \\ \mathit{play2}\ (n+1) & (\mathit{More}\ a\ q) = \textbf{do}\ a & \\ & \qquad\qquad \mathit{play2}\ n\ (\mathit{enQ}\ a\ q) & \end{array}$$

Equational reasoning interacts with both styles in the obvious way.

3.2 Translation into Haskell

The semantics of non-primitive functions on ADTs can be elaborated by a mechanical translation into ordinary Haskell, following a rather straightforward scheme: each use of a primitive function is replaced with its implementations, precomposed with to and postcomposed with from (subject to the appropriate hofunctors).

First of all, **adt** declarations are translated into **data** declarations.

$$\textbf{data}\ \mathit{Queue}'\ a = \mathit{None} \mid \mathit{More}\ a\ (\mathit{Queue}'\ a)$$

Functions that are written using pattern matching against the model now work with the new datatype. The primitive operations that are defined on the actual implementation require their inputs to be converted from the model before consumption, and the outputs converted back to the model. Effectively, all the translated functions and constructors have the model datatypes as source and target types; the implementations remain only as intermediate structures. As an example, *play2* is translated into the following.

$$\begin{array}{lll} \mathit{play2}'\ 0 & (\mathit{More}\ a\ q) = a & \\ \mathit{play2}'\ (n+1) & (\mathit{More}\ a\ q) = \textbf{do}\ a & \\ & \qquad\qquad \mathit{play2}'\ n\ ((\mathsf{to} \circ \mathit{enQ\_}\ a \circ \mathsf{from})\ q) & \end{array}$$

Given the round trip law (Lemma 1), it is easy to conclude that $\mathit{play2}'$ is equivalent to *play2*, in the sense that exactly the same output is produced for each input.

Theorem 2. *The translation into Haskell is semantics-preserving.*

Proof. Follows directly from Lemma 1. □

3.3 Optimization

Up to now, we have achieved sound equational reasoning for ADTs with little additional burden for the ADT implementer. As a result, program construction can benefit from pattern matching and straightforward proofs of correctness.

The run-time performance of non-primitive functions making use only of pattern matching can be understood by considering the models as datatypes; however, when primitive operations are called, additional conversion overhead will occur. This performance loss is to be expected for definitions such as *play2*, where an obvious switch from pattern matching to primitive operations is inevitable. However, it may be surprising that *play1*, which only involves primitive operations, is not faster. The translated code is the following.

```
play1' 0 q       = (to ∘ first_ ∘ from) q
play1' (n + 1) q = do hd
                      play1' n ((to ∘ enQ_ hd ∘ from) tl)
  where hd = (to ∘ first_ ∘ from) q
        tl = (to ∘ deQ_ ∘ from) q
```

There are conversions everywhere in the program. It will be disastrous if all of them have to be executed. Since there is no pattern matching involved, we can try to remove the conversions through fusion. Indeed, the correctness of such fusion follows from the promotion condition. Let's take an expression fragment from the above definition for demonstration. Consider

$$(\mathsf{to} \circ \mathit{enQ\_}\ \mathit{hd} \circ \mathsf{from})\ ((\mathsf{to} \circ \mathit{deQ\_} \circ \mathsf{from})\ q)$$

Our target is to fuse the intermediate conversions to produce

$$(\mathsf{to} \circ \mathit{enQ\_}\ \mathit{hd} \circ \mathit{deQ\_} \circ \mathsf{from})\ q$$

This would clearly follow $\mathsf{from} \circ \mathsf{to} \equiv \mathit{id}$, but this is not a property that we guarantee—for good reason, since requiring it in addition to the existing right inverse property $\mathsf{to} \circ \mathsf{from} \equiv \mathit{id}$ demands isomorphic implementations and models, which is too restrictive to be practically useful. Instead, using the promotion condition, we can prove a weaker property that is sufficient for fusion.

Theorem 3 (Fusion Soundness). *Given operation specifications* $f :: \mathcal{F}\ M\ a \to \mathcal{G}\ M\ a$ *and* $g :: \mathcal{G}\ M\ a \to \mathcal{H}\ M\ a$*, and their implementations* $f\_ :: \mathcal{F}\ N\ a \to \mathcal{G}\ N\ a$ *and* $g\_ :: \mathcal{G}\ N\ a \to \mathcal{H}\ N\ a$*, we have* $\mathcal{H}\ \mathsf{to} \circ g\_ \circ \mathcal{G}\ \mathsf{from} \circ \mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \equiv \mathcal{H}\ \mathsf{to} \circ g\_ \circ f\_ \circ \mathcal{F}\ \mathsf{from}$*.*

Proof

$$\begin{array}{ll}
 & \mathcal{H}\ \mathsf{to} \circ g\_ \circ \mathcal{G}\ \mathsf{from} \circ \mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \\
\equiv & \{\text{promotion: } \mathcal{H}\ \mathsf{to} \circ g\_ \equiv g \circ \mathcal{G}\ \mathsf{to}\} \\
 & g \circ \mathcal{G}\ \mathsf{to} \circ \mathcal{G}\ \mathsf{from} \circ \mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \\
\equiv & \{\mathcal{G} \text{ is a hofunctor; } \mathsf{to} \circ \mathsf{from} \equiv \mathit{id}\} \\
 & g \circ \mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \\
\equiv & \{\text{promotion: } \mathcal{H}\ \mathsf{to} \circ g\_ \equiv g \circ \mathcal{G}\ \mathsf{to}\} \\
 & \mathcal{H}\ \mathsf{to} \circ g\_ \circ f\_ \circ \mathcal{F}\ \mathsf{from}
\end{array}$$

□

Basically, this theorem states that although the input to $g\_$ may differ from the output of $f\_$, due to the $\mathsf{from} \circ \mathsf{to}$ conversions, nevertheless the post-conversion of $g\_$'s output brings possibly different results into the same value in the model.

It is now clear that when pattern matching is not used, *strength reduction* [29] is able to lift the conversion out of the recursion, so that it is done only once. The translation of *play1* can be optimized into the following, which is free from any overhead.

$$play1'\ n = \mathsf{to} \circ play1''\ n \circ \mathsf{from}$$

```
play1'' 0 q       = first q
play1'' (n + 1) q = do hd
                       play1'' n (enQ hd tl)
   where hd = first q
         tl = deQ q
```

3.4 More Examples

Join Lists. As an alternative to the biased linear list structure, the *join* representation of lists has been proposed for program elegance [27,4], efficiency [37], and more recently, parallelism [38]. It can be defined as:

```
data List a = Empty | Unit a | Join (List a) (List a)
```

As a simple example, a constant-time append function (in constrast to the linear-time left-biased-list counterpart) can be defined with this representation.

```
append l1 l2 = Join l1 l2
```

At the same time, we don't want to give up on the familiar notion of *Nil* and *Cons*. Instead, they can serve as a model of the join representation.

```
adt List a = Nil | Cons a (List a)
   append Nil ys         = ys
   append (Cons x xs) ys = Cons x (append xs ys)
```

We can now inherit the rich body of function definitions on lists. For example,

```
head (Cons x xs) = x
```

Binary Numbers. In the introduction, we showed a representation of binary numbers as lists of digits with the most significant bit first (MSB). This representation is intuitive, and offers good support for most operations; however, for incrementing a number, having the least significant bit (LSB) first is better.

```
type Num = [Bin]

incr' :: Num → Num
incr' []           = [One]
incr' (Zero : num) = One : num
incr' (One : num)  = Zero : (incr' num)
```

Effectively, in order to use the above definition with any other operations, we only need to reverse the MSB representation, and a type synonym for *Num* can be used as documentation of this intention. However, the synonyms are of no help to the compiler in guaranteeing correct usage, because they are simply two different names for the same type. At the same time, defining the two representations as completely different types is very cumbersome. With our proposal, we can hide one representation in an ADT, which effectively eliminates any possibility of misuse.

$$\begin{array}{l}\textbf{adt}\ \mathit{Num} = [\mathit{Bin}] \\ \quad \mathit{incr} = \mathit{reverse} \circ \mathit{incr}' \circ \mathit{reverse}\end{array}$$

In contrast to other examples, there is no new pattern interface other than the list constructors. However, a programmer using the ADT now only deals with a single representation of binary numbers.

4 The Right-Invertible Language 'RINV'

Our design of ADTs discussed previously relies on the existence of a right inverse, from, of the user-defined conversion function to. This can be guaranteed by writing to in a right-invertible language that automatically generates a right inverse for each function constructed. In this section, we introduce such a language.

The language RINV is defined as a combinator library in Haskell, the syntax of which is as below. (Non-terminals are indicated in small capitals.)

Language	RINV	$::=$ CSTR $\mid$ PRIM $\mid$ COMB
Constructors	CSTR	$::=$ $\mathit{nil} \mid \mathit{cons} \mid \mathit{snoc} \mid \mathit{wrap} \mid \ldots$
Primitives	PRIM	$::=$ $\mathit{app} \mid \mathit{id} \mid \mathit{assocr} \mid \mathit{assocl} \mid \mathit{swap} \mid \ldots$
Combinators	COMB	$::=$ RINV $\circ$ RINV $\mid$ *fold* RINV $\mid$ RINV ∇ RINV $\mid$ RINV $\times$ RINV

The language is similar in flavour to the *pointfree* style of programming [5], but with the additional feature that a right inverse is automatically generated for each function that is constructed. As a result, a definition $f :: s \leftrightarrows t$ in RINV actually represents a pair of functions (hence the notation $\leftrightarrows$): the forward function $[\![f]\!] :: s \to t$, and its right inverse $[\![f]\!]^{\circ} :: t \to s$, which together satisfy $[\![f]\!] \circ [\![f]\!]^{\circ} \equiv \mathit{id}$. For convenience when clear from context, we don't distinguish between f and its forward function $[\![f]\!]$.

The generated right inverses are intended to be total, so the forward functions have to be surjective; this property holds of the primitive functions (except for individual constructors of a multi-variant algebraic datatype) and is preserved by the combinators.

There is an extensible set of primitive functions defining the basic non-terminal building blocks of the language. Any surjective function could be made a primitive in RINV. All primitive functions are uncurried; this fits better with the invertible framework, where a clear distinction between input and output is required. For the sake of demonstration, we present a small but representative collection of primitive functions above: *swap*, *assocl*, and *assocr* rearrange the components of an input pair; *id* is the identity operation; *app* is the uncurried append function on lists. As we will show, with just these few we can define many interesting functions.

The set of constructor functions is also extensible, via new datatypes. We use lowercase names for the uncurried versions of constructors. In addition to the left-biased list constructor *cons* that comes with the usual datatype declaration, we also include its right-biased counterpart *snoc*, which adds an element at the end; it can be defined in Haskell as

$$\mathit{snoc} = \lambda(x, \mathit{xs}) \rightarrow \mathit{xs} \mathbin{+\!\!+} [x]$$

Another additional constructor for lists is *wrap*, which creates a singleton list.

$$\mathit{wrap}\ x = [x]$$

This ability to admit functions that do not directly arise from a datatype declaration as constructors is crucial for the expressiveness of RINV, which otherwise would be rigidly surjective. Although this might seem ad hoc, it is by no means arbitrary. One should only use functions that truly model a different representation of the datatype. For example, *snoc* and *nil* form the familar backward representation of lists, while *wrap*, *nil* and the primitive function *app* correspond to the join list representation found in Section 3.4.

Since constructor functions are exceptions to the surjectivity rule, lone constructors must be combined with other functions by the 'junc' combinator $\triangledown$, which dispatches to one of two functions according to the result of matching on a sum. When one of the operands of $\triangledown$ is surjective, or the two operands cover both constructors of a two-variant datatype, the result is surjective. For example, $\mathit{nil} \ \triangledown \ \mathit{cons}$ and $\mathit{nil} \ \triangledown \ \mathit{id}$ are both surjective, but $\mathit{cons} \ \triangledown \ \mathit{snoc}$ is not. Since $\triangledown$ can be nested, this result extends to datatypes with more than two constructors. Constructor functions can be composed with other functions as well, using the standard function composition combinator $\circ$, but only to the left: once a non-surjective function appears in a chain of compositions other than in the leftmost position, it is difficult to analyse the exact range of the composition, and the check for surjectivity ceases to be syntactic.

Other than the two already mentioned combinators, $\times$ is the cartesian product of two functions, and *fold* f is the unique homomorphism from the (implicit) initial algebra of a datatype to algebra f. We do not explicitly mention the datatype itself, as it is understood from context. Fold is the only combinator in RINV that is recursive. In combination with *swap*, *assocl* and *assocr*, $\times$ is able to define all functions that rearrange the components of a pair, while $\triangledown$ is useful in constructing the algebra for a *fold*. We don't include $\vartriangle$, the dual of $\triangledown$, in RINV, because of surjectivity, as will be explained shortly.

With the language RINV, we can state the following property.

Theorem 4 (Right invertibility). *Given a function f in* RINV*, for finitely defined input x, $([\![f]\!] \circ [\![f]\!]^{\circ})\ x \equiv x$.*

The correctness of this theorem should become evident by the end of this section, as we discuss in detail the various constructs of RINV and their properties. (Throughout this paper, unless otherwise mentioned, we always assume finitely defined values).

4.1 The Primitive Functions

The function *id* is the identity; functions *assocr*, *assocl* and *swap* manipulate pairs.

$$\begin{aligned}
&\mathit{assocr} :: ((a,b),c) \leftrightarrows (a,(b,c))\\
&\mathit{assocl} :: (a,(b,c)) \leftrightarrows ((a,b),c)\\
&\mathit{swap} :: (a,b) \leftrightarrows (b,a)
\end{aligned}$$

Together with the combinators $\times$ and $\circ$, these are sufficient to define many interesting functions on pairs. For example,

$$\begin{aligned}
&\mathit{subr} :: (b,(a,c)) \leftrightarrows (a,(b,c))\\
&\mathit{subr} = \mathit{assocr} \circ (\mathit{swap} \times \mathit{id}) \circ \mathit{assocl}\\
&\mathit{trans} :: ((a,b_1),(b_2,c)) \leftrightarrows ((a,b_2),(b_1,c))\\
&\mathit{trans} = \mathit{assocl} \circ (\mathit{id} \times \mathit{subr}) \circ \mathit{assocr}
\end{aligned}$$

Function *app* is the uncurried append function, which is not injective. The admission of non-injective functions is one of the most important distinctions between RINV and other invertible languages [31], allowing us to break away from the isomorphism restriction. There are many possible right inverses for *app*, of which we pick one:

$$[\![\mathit{app}]\!]^{\circ} = \lambda xs \rightarrow \mathit{splitAt}\ ((\mathit{length}\ xs + 1)\ \text{`}\mathit{div}\text{`}\ 2)\ xs$$

4.2 The Constructors

The semantics of the constructor functions are simple: they follow directly from the corresponding constructors introduced by datatype declarations, except for being uncurried. For example,

$$\begin{aligned}
&[\![\mathit{nil}]\!] = \lambda() \rightarrow [\,]\\
&[\![\mathit{cons}]\!] = \lambda(x, xs) \rightarrow x : xs
\end{aligned}$$

Constructors *snoc* and *wrap* are not primitive on left-biased lists, but can be encoded:

$$\begin{aligned}
&[\![\mathit{snoc}]\!] = \lambda(xs, x) \rightarrow xs +\!\!\!+ [x]\\
&[\![\mathit{wrap}]\!] = \lambda x \rightarrow [x]
\end{aligned}$$

Inverses of the primitive constructor functions are obtained simply by swapping the right- and left-hand sides of the definitions. For example, we have

$$\begin{aligned}
&[\![\mathit{nil}]\!]^{\circ} = \lambda[\,] \rightarrow ()\\
&[\![\mathit{cons}]\!]^{\circ} = \lambda(x : xs) \rightarrow (x, xs)
\end{aligned}$$

They are effectively partial 'guard' functions, succeeding when the input value matches the pattern. The right inverses of *snoc* and *wrap* are

$$\begin{aligned}
[\![snoc]\!]^\circ\,[x] &= ([\,], x)\\
[\![snoc]\!]^\circ\,(x : xs) &= \mathbf{let}\ (ys, y) = [\![snoc]\!]^\circ\, xs\ \mathbf{in}\ (x : ys, y)\\
[\![wrap]\!]^\circ\,[x] &= x
\end{aligned}$$

The inverses of constructor functions are generally not case-exhaustive. For example, $[\![cons]\!]^\circ$ only accepts non-empty lists, while $[\![nil]\!]^\circ$ only accepts the empty list. As a result, in contrast to primitive functions, constructor functions cannot be composed arbitrarily, as we will see shortly.

4.3 The Combinators

The combinators in RINV are familiar to functional programmers.

Composition, Sum and Product. Combinator $\circ$ sequentially composes two functions:

$$\begin{aligned}
[\![f \circ g]\!] &= [\![f]\!] \circ [\![g]\!]\\
[\![f \circ g]\!]^\circ &= [\![g]\!]^\circ \circ [\![f]\!]^\circ
\end{aligned}$$

Its inverse is the reverse composition of the inverses of the two arguments.

Combinators $\times$ and $\triangledown$ compose functions in parallel. The former applies a pair of functions component-wise to its input:

$$\begin{aligned}
&(\times) :: (a \leftrightarrows b) \to (c \leftrightarrows d) \to ((a, c) \leftrightarrows (b, d))\\
&[\![f \times g]\!] = \lambda(w, x) \to ([\![f]\!]\ w, [\![g]\!]\ x)\\
&[\![f \times g]\!]^\circ = \lambda(y, z) \to ([\![f]\!]^\circ\ y, [\![g]\!]^\circ\ z)
\end{aligned}$$

It is well known that $\times$ can be defined in term of a more primitive combinator $\triangle$, which executes both of its input functions on a single datum:

$$\begin{aligned}
&(\triangle) :: (a \leftrightarrows b) \to (a \leftrightarrows c) \to (a \leftrightarrows (b, c))\\
&[\![f \triangle g]\!] = \lambda x \to ([\![f]\!]\ x, [\![g]\!]\ x)
\end{aligned}$$

However, in the backward direction, $[\![f]\!]^\circ\ x$ and $[\![g]\!]^\circ\ y$ would have to converge, which is difficult to enforce statically. Indeed, functions constructed with $\triangle$ are generally not surjective, and so do not have total right inverses; for this reason, we exclude $\triangle$ from RINV.

The combinator $\triangledown$ consumes an element of a sum type.

$$\begin{aligned}
&\mathbf{data}\ Sum\ a\ b = Inl\ a \mid Inr\ b\\
&(\triangledown) :: (a \leftrightarrows c) \to (b \leftrightarrows c) \to (Sum\ a\ b \leftrightarrows c)\\
&[\![f \triangledown g]\!] = \lambda x \to \mathbf{case}\ x\ \mathbf{of}\ \{Inl\ a \to [\![f]\!]\ a\ ; Inr\ b \to [\![g]\!]\ b\}
\end{aligned}$$

In the backward direction, if both f and g are surjective, it doesn't matter which branch is chosen. However, the use of constructor functions deserves some attention, since they are not surjective in isolation. As a result, in the event that $[\![f]\!]^\circ$ fails on certain inputs, $[\![g]\!]^\circ$ should be applied. To model this failure handling, we lift functions in RINV into the *Maybe* monad (allowing an extra possibility for the return value), and handle a failure in the first function by invoking the second.

$$[\![f \triangledown g]\!]^\circ = \lambda x \rightarrow ([\![f]\!]^\circ\, x) \text{ `}\mathit{mplus}\text{` } ([\![g]\!]^\circ\, x)$$

This shallow backtracking is sufficient because the guards of conditionals are only pattern matching outcomes, which are completely decided at each level. For brevity, we still use the non-monadic types for $f \triangledown g$, with the understanding that all functions in RINV are lifted to the *Maybe* monad in the implementation.

In general, it is not an easy task to check (joint) surjectivity of functions. However, in RINV, this test is made relatively straightforward, since the only possible cause for $f \triangledown g$ not to be jointly surjective is that both f and g use constructor functions; in this case, it is clear that we need the complete set of constructors to satisfy the condition of joint surjectivity. We demonstrate this check with examples towards the end of this section.

The more intricate part is to analyse the surjectivity of the composition (and hence the totality of its inverse). It is clear that if one of the functions in a chain of compositions is not surjective, the composed function may also be non-surjective. However, there is no easy way of determining the range of such a composition if the non-surjective function is not the leftmost one in the chain, which makes it unsuitable for constructing jointly surjective functions through the $\triangledown$ combinator as discussed above. Therefore, in RINV, we disallow compositions involving constructor functions on the right of a composition.

Fold. With the ground prepared, we are now ready to discuss recursive combinators. We define

$$\begin{aligned}[\![\mathit{fold}\ f]\!] &= \mathit{fold}_X\ [\![f]\!]\\ [\![\mathit{fold}\ f]\!]^\circ &= \mathit{unfold}_X\ [\![f]\!]^\circ\end{aligned}$$

The forward semantics of *fold f* is defined in terms of the standard fold_X for a datatype X, and the backward semantics is defined by a corresponding unfold_X. In what follows, we call the f in *fold f* the 'body' of the fold. Note that *unfold* is not in RINV, but is used to define right inverses. In this paper, we overload *fold* and *unfold* when the datatype is understood. Intuitively, *fold* disassembles a structure and replaces the constructors with applications of the body, effectively collapsing the structure. Function *unfold*, on the other hand, takes a seed, splitting it with the body into building blocks of a structure and new seeds, which are themselves recursively unfolded. In short, *fold* collapses a structure, whereas *unfold* grows one.

When an algebraic datatype X is given, Haskell definitions of fold_X and unfold_X can be generated. For example, consider the datatype of lists:

$$\begin{aligned}&\mathit{fold}_L :: (\mathit{Sum}\ ()\ (a, b) \rightarrow b) \rightarrow (\mathit{List}\ a \rightarrow b)\\ &\mathit{fold}_L\ f = \lambda \mathit{xs} \rightarrow \textbf{case}\ \mathit{xs}\ \textbf{of}\end{aligned}$$

$$
\begin{array}{ll}
\qquad [\,] & \rightarrow f\ (\mathit{Inl}\ ()) \\
\qquad (x : xs) & \rightarrow f\ (\mathit{Inr}\ (x, (\mathit{fold}_L\ f\ xs)))
\end{array}
$$

$$
\begin{array}{l}
\mathit{unfold}_L :: (b \rightarrow \mathit{Sum}\ ()\ (a, b)) \rightarrow (b \rightarrow \mathit{List}\ a) \\
\mathit{unfold}_L\ f = \lambda b \rightarrow \textbf{case}\ f\ b\ \textbf{of}\ \mathit{Inl}\ () \quad \rightarrow [\,] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Inr}\ (a, b) \rightarrow a : (\mathit{unfold}_L\ f\ b)
\end{array}
$$

Another example is leaf-labelled binary trees. Note that the constructor *Fork* is uncurried to fit better into the RINV framework.

$$
\begin{array}{l}
\textbf{data}\ \mathit{LTree}\ a = \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{LTree}\ a, \mathit{LTree}\ a) \\[4pt]
\mathit{fold}_T :: (\mathit{Sum}\ a\ (b, b) \rightarrow b) \rightarrow \mathit{LTree}\ a \rightarrow b \\
\mathit{fold}_T\ f = \lambda t \rightarrow \textbf{case}\ t\ \textbf{of} \\
\quad \mathit{Leaf}\ a \qquad\quad \rightarrow f\ (\mathit{Inl}\ a) \\
\quad \mathit{Fork}\ (t_1, t_2) \rightarrow f\ (\mathit{Inr}\ (\mathit{fold}_T\ f\ t_1, \mathit{fold}_T\ f\ t_2)) \\[4pt]
\mathit{unfold}_T :: (a \rightarrow \mathit{Sum}\ a\ (b, b)) \rightarrow b \rightarrow \mathit{LTree}\ a \\
\mathit{unfold}_T\ f = \lambda b \rightarrow \textbf{case}\ f\ b\ \textbf{of} \\
\qquad\qquad\qquad \mathit{Inl}\ a \rightarrow \mathit{Leaf}\ a \\
\qquad\qquad\qquad \mathit{Inr}\ (b_1, b_2) \rightarrow \mathit{Fork}\ (\mathit{unfold}_T\ f\ b_1, \mathit{unfold}_T\ f\ b_2)
\end{array}
$$

We use *unfold* to construct the right inverse of *fold*. From [12], we have the following lemma.

Lemma 5. $\mathit{fold}\ [\![f]\!] \circ \mathit{unfold}\ [\![f]\!]^{\circ} \sqsubseteq \mathit{id}$.

Since both *fold* and *unfold* are case-exhaustive when their bodies are case-exhaustive, the only reason for not having an equality in the lemma above is that *unfold* is potentially non-terminating: when a body does not split a seed into 'smaller' seeds, unfolding a seed creates an infinite structure. It is well known that a function constructed by *unfold* terminates if the seed transformation is *well-founded* (that is, there should be no infinite descending chain of seeds). Static termination checkers exist in the literature [20, 36] and are orthogonal to the discussion here.

4.4 Programming in RINV

With the knowledge of RINV, we are now ready to look into the kinds of function we can define with it.

To start with, let's look first at a very useful derived combinator *map* that can be defined in term of *fold*. For example, *map* on lists, map_L, is defined as follows.

$$
\begin{array}{l}
\mathit{map}_L :: (a \leftrightarrows b) \rightarrow (\mathit{List}\ a \leftrightarrows \mathit{List}\ b) \\
\mathit{map}_L\ f = \mathit{fold} \circ (\mathit{nil} \triangledown (\mathit{cons} \circ (f \times \mathit{id})))
\end{array}
$$

Function $\mathit{map}_L\ f$ applies argument f uniformly to all the elements of a list, without modifying the list structure. Since *nil* and *cons* form a complete set of constructors for lists, we know they are jointly surjective.

Similarly, *map* on leaf-labeled trees, map_T, is defined as follows.

$$\begin{array}{l} map_T :: (a \leftrightarrows b) \rightarrow (Tree\ a \leftrightarrows Tree\ b) \\ map_T\ f = fold_T \circ ((leaf \circ f) \triangledown fork) \end{array}$$

The function *reverse* on lists can be defined as a fold:

$$\begin{array}{ll} reverse & = fold\ (nil \triangledown snoc) \\ [\![reverse]\!]^\circ & = unfold\ [\![nil \triangledown snoc]\!]^\circ \end{array}$$

In the forward direction, a list is taken apart and the first element is appended to the rear of the output list by *snoc*. This process terminates on reaching an empty list, when an empty list is returned as the result. Function $[\![snoc]\!]^\circ$ extracts the last element in a list and adds it to the front of the result list by *unfold*, which terminates when $[\![nil]\!]^\circ$ can be successfully applied (i.e when the input is the empty list). Since *nil* and *snoc* form a complete set of constructors for lists, they are jointly surjective.

Function *reverse* is also used to construct the *apprev* function that reverses a list and appends it.

$$\begin{array}{l} apprev :: ([a],[a]) \rightarrow [a] \\ apprev = app \circ (id \times reverse) \end{array}$$

Function *apprev* reverses the second list before concatenating the two. For example, we have:

$$apprev\ ([1,2],[3,4,5,6,7]) = [1,2,7,6,5,4,3]$$

The companion $apprev^\circ$ function is

$$\begin{array}{l} apprev^\circ :: [a] \rightarrow ([a],[a]) \\ apprev^\circ = [\![app \circ (id \times reverse)]\!]^\circ \end{array}$$

In the backward direction, a list is split into two, and functions $[\![id]\!]^\circ$ and $[\![reverse]\!]^\circ$ are applied to the two parts. For example, we have

$$\begin{array}{rl} apprev\ (apprev^\circ\ ([1,2,7,6,5,4,3])) & \equiv apprev\ ([1,2,7,6],[3,4,5]) \\ & \equiv [1,2,7,6,5,4,3] \end{array}$$

On the other hand,

$$\begin{array}{rl} apprev^\circ\ (apprev\ ([1,2],[3,4,5,6,7])) & \equiv apprev^\circ\ ([1,2,7,6,5,4,3]) \\ & \equiv ([1,2,7,6],[3,4,5]) \end{array}$$

It is clear from above that $apprev^\circ$ is not a left inverse of *apprev*, and it is not intended to be a term in the language RINV.

Our last example is the traversal of node-labelled binary trees.

$$\textbf{data}\ BinTree\ a = BLeaf \mid BNode\ a\ (BinTree\ a, BinTree\ a)$$

The fold/unfold functions for binary trees are as follows.

$$\begin{array}{l} \mathit{fold}_B :: (\mathit{Sum}\ ()\ (a,(b,b)) \to b) \to (\mathit{BinTree}\ b \to b) \\ \mathit{fold}_B\ f = \lambda x \to \textbf{case}\ x\ \textbf{of} \\ \qquad\qquad \begin{array}{ll} \mathit{BLeaf} & \to f\ (\mathit{Inl}\ ()) \\ \mathit{BNode}\ a\ (l,r) & \to f\ (\mathit{Inr}\ (a,(\mathit{fold}_B\ f\ l,\mathit{fold}_B\ f\ r))) \end{array} \end{array}$$

$$\begin{array}{l} \mathit{unfold}_B :: (b \to \mathit{Sum}\ ()\ (a,(b,b))) \to (b \to \mathit{BinTree}\ b) \\ \mathit{unfold}_B\ f = \\ \quad \lambda x \to \textbf{case}\ f\ x\ \textbf{of} \\ \qquad \begin{array}{ll} \mathit{Inl}\ () & \to \mathit{BLeaf} \\ \mathit{Inr}\ (a,(l,r)) & \to \mathit{BNode}\ a\ (\mathit{unfold}_B\ f\ l,\mathit{unfold}_B\ f\ r) \end{array} \end{array}$$

Using the fold_B combinator, pre- and post-order traversal of a binary tree can be defined as follows.

$$\begin{array}{ll} \mathit{preOrd} & = \mathit{fold}_B\ (\mathit{nil} \triangledown (\mathit{cons} \circ (\mathit{id} \times \mathit{app}))) \\ \mathit{postOrd} & = \mathit{fold}_B\ (\mathit{nil} \triangledown (\mathit{snoc} \circ \mathit{swap} \circ (\mathit{id} \times \mathit{app}))) \end{array}$$

In the forward direction, fold_B adds the node value at one end of the concatenation of the two subtrees' traversals. In the backward direction, a node value is extracted from the input list, and the rest of the list is divided and grown into individual trees.

As a final remark to readers familiar with pointfree programming in Haskell, the primitive function *app* can be defined as a fold:

$$\mathit{app} = \mathit{uncurry}\ (\mathit{flip}\ (\mathit{foldr}\ (:)))$$

which effectively partially applies *foldr* and awaits an input as the base case. This idiom of taking an extra argument to form the base case is difficult to realize when the fold body is constructed independently, as it is in RINV. For some cases, it even threatens the existence of right inverses as unfolds. For example, the following function in Haskell

$$\begin{array}{l} f :: a \to [\mathit{LTree}\ a] \to \mathit{LTree}\ a \\ f\ a = \mathit{foldr}\ \mathit{Fork}\ (\mathit{Leaf}\ a) \end{array}$$

does not have a right inverse as an unfold. In RINV, we rule out definitions of this kind, and treat *app* as a primitive.

5 Discussion

5.1 The Dual Story

In this paper, we have picked the `to` function to be provided by ADT implementers; the design of RINV and the subsequent discussion of ADTs is based on this decision. However, this choice is by no means absolute. One can well imagine ADT implementers coming up with `from` functions first, and a left-invertible language generating the corresponding to functions; this would give the same invertibility property $\mathsf{to} \circ \mathsf{from} \equiv id$. But the implementer is now expected to prove a different promotion condition, $f\_ \circ \mathcal{F}\ \mathsf{from} \equiv \mathcal{G}\ \mathsf{from} \circ f$, adapted to involve only from. Nevertheless, the crucial round-trip law and fusion law that form the foundation of the translation and optimization are still derivable; for round-trip, we have

$$\begin{array}{ll}
 & \mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \\
\equiv & \{ f\_ \circ \mathcal{F}\ \mathsf{from} \equiv \mathcal{G}\ \mathsf{from} \circ f \} \\
 & \mathcal{G}\ \mathsf{to} \circ \mathcal{G}\ \mathsf{from} \circ f \\
\equiv & \{ \mathcal{G} \text{ is a hofunctor; } \mathsf{to} \circ \mathsf{from} \equiv id \} \\
 & f
\end{array}$$

and for fusion:

$$\begin{array}{ll}
 & \mathcal{H}\ \mathsf{to} \circ g\_ \circ \mathcal{G}\ \mathsf{from} \circ \mathcal{G}\ \mathsf{to} \circ f\_ \circ \mathcal{F}\ \mathsf{from} \\
\equiv & \{ f\_ \circ \mathcal{F}\ \mathsf{from} \equiv \mathcal{G}\ \mathsf{from} \circ f \} \\
 & \mathcal{H}\ \mathsf{to} \circ g\_ \circ \mathcal{G}\ \mathsf{from} \circ \mathcal{G}\ \mathsf{to} \circ \mathcal{G}\ \mathsf{from} \circ f \\
\equiv & \{ \mathcal{G} \text{ is a hofunctor; } \mathsf{to} \circ \mathsf{from} \equiv id \} \\
 & \mathcal{H}\ \mathsf{to} \circ g\_ \circ \mathcal{G}\ \mathsf{from} \circ f \\
\equiv & \{ f\_ \circ \mathcal{F}\ \mathsf{from} \equiv \mathcal{G}\ \mathsf{from} \circ f \} \\
 & \mathcal{H}\ \mathsf{to} \circ g\_ \circ f\_ \circ \mathcal{F}\ \mathsf{from}
\end{array}$$

5.2 Reasoning about Efficiency

A controversy in any design that embeds implicit computations into pattern matching is the datatype-like notation. We think this feature is positive, since it preserves the elegant syntax of pattern matching and offers backward compatibility, an important property for incremental refactoring. On the other hand, there is the concern that this similar look and feel may cause programmers to overlook the possibility of non-constant run-time cost of pattern matching on models. This is certainly a valid concern. As we have seen in Section 3.3, such conversions only occur when primitive operations and pattern matching interact. If this occurs in a recursion, run-time complexity could be affected. However, it is clear that this inefficiency can be eliminated by not mixing primitive operations and pattern matching in recursions.

5.3 Nested and Overlapping Patterns

Two well-regarded features of pattern matching are the scalability with respect to nesting and the sharing between overlapping patterns. For example, consider a function that sums elements of a list pair-wise:

$$\begin{array}{ll}
pairSum\ Nil & = Nil \\
pairSum\ (Cons\ x\ Nil) & = Cons\ x\ Nil \\
pairSum\ (Cons\ x\ (Cons\ y\ ys)) & = Cons\ (x + y)\ (pairSum\ ys)
\end{array}$$

Nested patterns allow simultaneous matching and variable binding to patterns below top level (such as y above), in contrast to the sequential checking of expressions as guards. There is often a degree of sharing between patterns; for example, input to *pairSum* that, when evaluated, fails to match the first pattern does not need to be evaluated again for subsequent clauses. This is even more important for pattern matching on models where non-constant computation (i.e., the to function) may be needed. Our proposal supports both features nicely: nested patterns are written exactly the same way as with datatypes; and execution of to functions is done prior to pattern matching and is shared among all the patterns.

5.4 RINV Expressiveness

In our system, the set of definable models is determined by the existence of to functions in RINV that map to them. RINV is designed to be extensible: new primitives (and even new combinators) can be added to the language if needed. The real limitation of RINV we face here is that all functions must be surjective, in order to ensure existence of the right inverses: valid model values are bounded by the actual range of the user-defined to function; invertibility is not guaranteed for model values outside this range.

Totality of from is certainly desirable if it is used for model conversion, since failures will not be observable through reasoning. In the current proposal, the to functions in RINV are always surjective, which rules out some useful programs. An example already mentioned is the combinator $\triangle$ which executes both of its input functions, and is defined as

$$\begin{aligned}&(\triangle) :: (a \to b) \to (a \to c) \to a \to (b, c)\\ &(f \triangle g) = \lambda x \to (f\ x, g\ x)\end{aligned}$$

Since $f \triangle g$ is generally not surjective, it doesn't have a right inverse, despite the fact that we can easily guard against inconsistent input in the reverse direction as follows.

$$\begin{aligned}&[\![f \triangle g]\!]^{\circ} = \lambda(a, b) \to \textbf{if}\ x == y\ \textbf{then}\ x\ \textbf{else}\ \mathit{error}\ \texttt{"violation"}\\ &\quad \textbf{where}\ x = [\![f]\!]^{\circ}\ a\ ; y = [\![g]\!]^{\circ}\ b\end{aligned}$$

Definitions like the one above are known as *weak right inverses* [30].

Another useful function is *unzip*, which can be defined as a fold.

$$\mathit{unzip} = \mathit{fold}_L\ ((\mathit{nil} \triangle \mathit{nil}) \triangledown ((\mathit{cons} \times \mathit{cons}) \circ \mathit{trans}))$$

This definition will be rejected in RINV, since $\mathit{cons} \times \mathit{cons}$ and $\mathit{nil} \triangle \mathit{nil}$ are not jointly surjective. Indeed, *unzip* only produces pairs of lists of equal length. This is also the very reason that we exclude *unfold* as a combinator in RINV, as it in general only constructs structures of a particular shape, as determined by the splitting strategy of its body.

If a model value outside the range is constructed, the integrity of model level equational reasoning may be corrupted. On the other hand, it is valid to argue that the same invariant assumed for the original datatype prior to the refactoring applies to the model too. For example, consider a program that requires balanced binary trees. A to function that only produces balanced binary trees is safe if the invariant is correctly preserved in the original program. It remains an open question whether we should allow programmers to take some reasonable responsibilities, or should insist on enforcing control through the language.

6 Related Work

Efforts to combine data abstraction and pattern matching started two decades ago with Wadler's *views* proposal [43]; and it is still a hot research topic [8, 34, 10, 35, 11, 41, 19, 21, 39, 33].

Wadler's views provide different ways of viewing data than their actual implementations. With a pair of conversion functions, data can be converted *to* and *from* a view. Consider the forward and backward representations of lists:

```
data List a = Nil | Cons a (List a)
view List a = Lin | Snoc (List a) a
  to Nil                      = Lin
  to (Cons x Nil)             = Snoc Nil x
  to (Cons x (Snoc xs y))     = Snoc (Cons x xs) y
  from Lin                    = Nil
  from (Snoc Nil x)           = Cons x Nil
  from (Snoc (Cons x xs) y)   = Cons x (Snoc xs y)
```

The **view** clause introduces two new constructors, namely *Lin* and *Snoc*, which may appear in both terms and patterns. The first argument to the view construction *Snoc* refers to the datatype *List a*, so a snoclist actually has a conslist as its child. The to and from clauses are similar to function definitions. The to clause converts a conslist value to a snoclist value, and is used when *Lin* or *Snoc* appear as the outermost constructor in a pattern on the left-hand side of an equation. Conversely, the from clause converts a snoclist into a conslist, when *Lin* or *Snoc* appear in an expression. Note that we are already making use of views in the definition above; for example, *Snoc* appears on the left-hand side of the third to clause, matching against which will trigger a recursive invocation of to.

Functions can now pattern match on and construct values in either the datatype or one of its views.

```
last (Snoc xs x) = x

rotLeft (Cons x xs)  = Snoc xs x
rotRight (Snoc xs x) = Cons x xs

rev Nil          = Lin
rev (Cons x xs)  = Snoc (rev xs) x
```

Upon invocation, an argument is converted into the view by the to function; after completion of the computation, the result is converted back to the underlying datatype representation.

Just as with our proposal, this semantics can be elaborated by a straightforward translation into ordinary Haskell. First of all, view declarations are translated into data declarations.

$$\mathbf{data}\ \mathit{Snoc}\ a = \mathit{Lin} \mid \mathit{Snoc}\ (\mathit{List}\ a)\ a$$

Note that the child of *Snoc* refers to the underlying datatype: view data is typically hybrid (in contrast to our approach). Now the only task is to insert the conversion functions at appropriate places in the program.

```
last xs = case to xs of Snoc xs x → x

rotLeft xs  = case xs of Cons x xs → from (Snoc xs x)
rotRight xs = case to xs of Snoc xs x → Cons x xs

rev xs = case xs of
  Nil          → from Lin
  (Cons x xs)  → from (Snoc (rev xs) x)
```

In contrast to our approach, Wadler exposes both a datatype and its views to programmers. To support reasoning across the different representations, the conversion clauses are used as axioms.

For example, we can evaluate an expression:

$$\begin{array}{ll}
 & \mathit{last}\ (\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ \mathit{Nil})) \\
\equiv & \{\ \mathit{Cons}\ x\ \mathit{Nil} = \mathit{Snoc}\ \mathit{Nil}\ x\ \} \\
 & \mathit{last}\ (\mathit{Cons}\ 1\ (\mathit{Snoc}\ \mathit{Nil}\ 2)) \\
\equiv & \{\ \mathit{Cons}\ x\ (\mathit{Snoc}\ xs\ y) = \mathit{Snoc}\ (\mathit{Cons}\ x\ xs)\ y\ \} \\
 & \mathit{last}\ (\mathit{Snoc}\ (\mathit{Cons}\ 1\ \mathit{Nil})\ 2) \\
\equiv & \{\ \mathit{last}\ \} \\
 & 2
\end{array}$$

or calculate with functions:

$$\begin{array}{ll}
 & \mathit{rotRight}\ (\mathit{rotLeft}\ (\mathit{Cons}\ x\ xs)) \\
\equiv & \{\ \mathit{rotLeft}\ \} \\
 & \mathit{rotRight}\ (\mathit{Snoc}\ xs\ x) \\
\equiv & \{\ \mathit{rotRight}\ \} \\
 & \mathit{Cons}\ x\ xs
\end{array}$$

However, this style of reasoning is limited in expressiveness. For example, there is no way to calculate with $\mathit{rotLeft} \circ \mathit{rotRight}$, because in Wadler's setting, inputs are always constructed in the underlying datatype: in $(\mathit{rotLeft} \circ \mathit{rotRight})\ (\mathit{Cons}\ x\ xs)$, there is no way of converting $\mathit{Cons}\ x\ xs$ to a *Snoc* view, which would allow the calculation of *rotRight* to proceed. (The claim in the original paper [43] that $\mathit{rotLeft} \circ \mathit{rotRight} \equiv \mathit{id}$ is provable is incorrect; what is actually provided is a proof that $\mathit{rotRight} \circ \mathit{rotLeft} \equiv \mathit{id}$.)

A perhaps more noticeable weakness of views is the use of user-defined conversions as axioms. It is expected for a view type to be isomorphic to a subset of its underlying datatype, and for the pair of conversions between the values of the two types to be each other's full inverses. This is certainly restrictive; and Wadler didn't suggest any way to enforce such an invertibility condition. As pointed out by Wadler himself [43], and followed up by several others [8,34], this assumption is risky, and may lead to nasty surprises that threaten soundness of reasoning.

Inspired by Wadler's proposal, our work ties up the loose ends of views by hiding the underlying datatype as an ADT, and using only the view (our model) for pattern matching. The implementations of primitive operations of the ADT can be proven correct through comparison against the constructive specification, at no additional cost to ADT implementers. The language RINV for defining conversions guarantees right invertibility, a weaker condition that lifts the isomorphism restriction on abstract representations and models. In contrast to views, our system does not cater for multiple views of the same ADT, because given no explicit axioms connecting them, it is difficult to reason across views.

'Safe' variants of views have been proposed before [8,34]. To circumvent the problem of equational reasoning, one typically restricts the use of view constructors to patterns, and does not allow them to appear on the right-hand side of a definition. As a result, expressions like *Snoc Lin* 1 become syntactically invalid. Instead, values are only constructed by 'smart constructors', as in *snoc lin* 1. In this setting, equational reasoning has to be conducted on the source level with explict applications of `to`. A major motivation for such a design is to admit views and sources with conversion functions that do not satisfy the invertibility property. In another words, let *Constr* and *constr* be a constructor and its corresponding smart constructor; in general, we have $Constr\ x \not\equiv constr\ x$. This appears to hinder program comprehension, since the very purpose of the convention that the name of a smart constructor differs only by case from its 'dumb' analogue is to suggest the equivalence of the two.

More recently, language designers have started looking into more expressive pattern mechanisms. *Active patterns* [10,35] and many of their variants [11,41, 19,21,39,35] go a step further, by embedding computational content into pattern constructions. All the above proposals either explicitly recognise the benefit of using constructors in expressions, or use examples that involve construction of view values on the right-hand sides of function definitions. Nevertheless, none of them are able to support pattern constructors in expressions, due to the inability to reason safely. Knowing that there is an absence of good solutions for supporting constructors in expressions, some work focusses mainly on examples that are primarily data consumers, an escape that is expected to be limited and short-lived. Another common pitfall of active patterns is the difficulty in supporting nested and overlapping patterns, as discussed in Section 5.3, because each active pattern is computed and matched independently.

In particular, equational reasoning with ADTs is one of the central themes in two notable proposals [8, 35]. These proposals demonstrate the possibility of reasoning about programs containing safe views or active patterns, through the axiomatic specifications of ADTs. In particular, it is observed in [8] that an algebraic datatype called the 'associated free type' (model in our case) may serve as the interface of an ADT. Through the view mechanism, an associated free type can differ in structure from the abstract representation of the ADT. In contrast to ours, their proposal is not able to separate functions over an ADT into primitive and non-primitive, an essential feature for incremental refactoring; nor to recognize the value of right-invertibility of `to` as the key to sound reasoning with models.

The language RINV owes its origins to the rich literature on *invertible* programming [32, 2], a programming paradigm where programs can be executed both forwards and backwards. Mu *et al.* concentrate their effort on designing a language that provides only injective functions. The resulting language *Inv* is a combinator library that syntactically rules out any non-injective functions. The most novel operator of *Inv* is *dup f*, which duplicates the input and applies f to one copy. In the backward direction, the two copies of the duplicated input are checked for consistency before being restored. It is shown that *Inv* is practically useful for maintaining consistency of structured data related by some transformations [31, 17]. Invertible arrows [2] extend the arrow framework [18] (a generalization of monads) with a combinator that encodes pairs of functions being each other's inverses. In an aside in [2], it is recognized that when full invertibility is not achievable (due to the non-isomorphic nature of the two sides), biased (either left- or right-) invertibility is nevertheless approximation.

Right inverses have been studied as a component of the much more elaborate bidirectional programming framework of *lenses* [13,6,15,14] targeting view-updates of XML databases; in this context, right inverses are known as 'create' functions. Based on record types, the combinators of lenses have little similarity to those of RINV. A distinctive feature of the lenses framework is the use of *semantic* types [16] to give precise bounds to the ranges of forward functions (thus the domains of backward functions). As a result, surjectivity now concerns the relationships of domains/ranges of lenses connected by a combinator, instead of being a property between a function and its target datatype.

7 Conclusion

Algebraic datatypes and pattern matching offer great promise to programmers seeking simple and elegant programming, but the promise turns sour when modular changes are demanded. Our work tackles this long-standing problem by proposing a framework for refactoring programs written with pattern matching into ones with ADTs: programmers are able to selectively reimplement original function definitions into primitive operations of the ADT, and either rewrite the rest in terms of the primitive ones, or simply leave them unchanged. This migration is completely incremental: executability and proofs through equational reasoning are preserved at all times during the process.

At the heart of our proposal is a novel design of ADTs enriched with algebraic models for backwards-compatible pattern matching. The model has the same interface as the datatype that is being replaced; and the original definitions of the selected primitive operations are turned into constructive specifications, through which equational reasoning connects the primitive operations and the rest of the functions. The soundness of such reasoning is established by the right-inverse property of the conversion pairs that bridge the model and the abstract representation.

We have implemented the language RINV as a combinator library in Haskell, and a naïve translation of ADTs into Haskell immediately follows. However, it remains to investigate how the fusion theory developed in the paper can be applied in practice. There is literature on fusing embedding-projection pairs (conceptually similar to our to and from functions) [1,25]. However the treatment of recursive functions using a fixpoint combinator in [1] is not applicable directly, because the 'deep' embedding in our approach (not producing hybrid data) does not allow conversions to be isolated into a non-recursive part. Instead, we plan to employ an analysis of mixed uses of patterns and primitive operations; and use strength reduction to remove the conversions, as outlined in Section 3.3.

At this stage, our focus is on supporting refactoring of programs written with datatypes and pattern matching, which automatically excludes some ADTs, such as unordered sets, that cannot be fully modelled by algebraic datatypes. We leave it as future work to investigate the applicability of our proposal in a more general setting.

Acknowledgements

We are grateful to Ralf Hinze for his valuable comments on an early draft of the paper; the binary number example is due to him. This work is supported by the EPSRC grant *Generic and Indexed Programming* (EP/E02128X), and was partly conducted during the first author's internship at National Institute of Informatics, Japan. Part of the work of the third author was done while the author was at University of Tokyo as JSPS Research Fellow supported by Grant-in-Aid for JSPS Fellows $20 \cdot 9584$.

References

1. Alimarine, A., Smetsers, S.: Optimizing generic functions. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 16–31. Springer, Heidelberg (2004)
2. Alimarine, A., Smetsers, S., van Weelden, A., van Eekelen, M., Plasmeijer, R.: There and back again: Arrows for invertible programming. In: Haskell Workshop, pp. 86–97. ACM, New York (2005)
3. Bird, R.S.: The promotion and accumulation strategies in transformational programming. ACM Transactions on Programming Languages and Systems 6(4), 487–504 (1984)
4. Bird, R.S.: An introduction to the theory of lists. In: Broy, M. (ed.) Logic of Programming and Calculi of Discrete Design. NATO ASI Series F, vol. 36, pp. 3–42. Springer, Heidelberg (1987); Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University

5. Bird, R.S.: A calculus of functions for program derivation. In: Research Topics in Functional Programming, pp. 287–307. Addison-Wesley, Reading (1990)
6. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: Resourceful lenses for string data. In: Principles of Programming Languages, January 2008. ACM, New York (2008)
7. Burstall, R., MacQueen, D., Sannella, D.: Hope: An experimental applicative language. In: Lisp and Functional Programming, pp. 136–143. ACM, New York (1980)
8. Burton, F.W., Cameron, R.D.: Pattern matching with abstract data types. Journal of Functional Programming 3(2), 171–190 (1993)
9. 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)
10. Erwig, M.: Active patterns. In: Kluge, W.E. (ed.) IFL 1996. LNCS, vol. 1268, pp. 21–40. Springer, Heidelberg (1997)
11. Erwig, M., Peyton Jones, S.: Pattern guards and transformational patterns. In: Haskell Workshop. ACM, New York (2000)
12. Fokkinga, M., Meijer, E.: Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, Netherlands (January 1991)
13. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. ACM Transactions on Programming Languages and Systems 29(3) (May 2007); Preliminary version in POPL '05 (2005)
14. Foster, J.N., Pierce, B.C., Zdancewic, S.: Updatable security views. In: CSF '09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium, Washington, DC, USA, pp. 60–74. IEEE Computer Society Press, Los Alamitos (2009)
15. Foster, J.N., Pilkiewicz, A., Pierce, B.C.: Quotient lenses. In: International Conference on Functional Programming, pp. 383–396. ACM, New York (2008)
16. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. Journal of the ACM 55(4) (2008)
17. Hu, Z., Mu, S.-C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. In: Partial Evaluation and Program Manipulation, pp. 178–189. ACM, New York (2004)
18. Hughes, J.: Generalising monads to arrows. Science of Computer Programming 37(1-3), 67–111 (2000)
19. Jay, C.B.: The pattern calculus. ACM Transactions on Programming Languages and Systems 26(6) (2004)
20. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Principles of Programming Languages, pp. 81–92. ACM, New York (2001)
21. Licata, D., Peyton Jones, S.: View patterns: lightweight views for Haskell (2007), http://hackage.haskell.org/trac/ghc/wiki/ViewPatterns
22. Liskov, B., Guttag, J.: Program Development in Java: Abstraction, Specification, and Object-Oriented Design. Addison-Wesley, Boston (2000)
23. Liskov, B., Zilles, S.: Programming with abstract data types. In: ACM Symposium on Very High Level Languages (1974)
24. Liu, J., Myers, A.C.: JMatch: Iterable abstract pattern matching for Java. In: Dahl, V., Wadler, P. (eds.) PADL 2003. LNCS, vol. 2562, pp. 110–127. Springer, Heidelberg (2002)

25. Magalhães, J.P., Holdermans, S., Jeuring, J., Löh, A.: Optimizing generics is easy! In: PEPM '10: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation, pp. 33–42. ACM, New York (2010)
26. Martin, C., Gibbons, J., Bayley, I.: Disciplined, efficient, generalised folds for nested datatypes. Formal Aspects of Computing 16(1), 19–35 (2004)
27. Meertens, L.G.L.T.: Algorithmics: Towards programming as a mathematical activity. In: CWI Symposium on Mathematics and Computer Science. CWI-Monographs, vol. 1, pp. 289–344. North-Holland, Amsterdam (1986)
28. Moreau, P.-E., Ringeissen, C., Vittek, M.: A pattern matching compiler for multiple target languages. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 61–76. Springer, Heidelberg (2003)
29. Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Communications of the ACM 22(2), 96–103 (1979)
30. Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: Automatic inversion generates divide-and-conquer parallel programs. In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 146–155. ACM, New York (2007)
31. Mu, S.-C., Hu, Z., Takeichi, M.: An algebraic approach to bi-directional updating. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 2–18. Springer, Heidelberg (2004)
32. Mu, S.-C., Hu, Z., Takeichi, M.: An injective language for reversible computation. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 289–313. Springer, Heidelberg (2004)
33. Nogueira, P., Moreno-Navarro, J.J.: Bialgebra views: A way for polytypic programming to cohabit with data abstraction. In: Workshop on Generic Programming, pp. 61–73. ACM, New York (2008)
34. Okasaki, C.: Views for Standard ML. In: ACM Workshop on ML (1998)
35. Palao Gostanza, 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, New York (1996)
36. Sereni, D.: Termination analysis and call graph construction for higher-order functional programs. In: Ramsey, N. (ed.) International Conference on Functional Programming, pp. 71–84. ACM Press, New York (2007)
37. Sleep, M.R., Holmström, S.: A short note concerning lazy reduction rules for append. Software: Practice and Experience 12(11) (1982)
38. Steele Jr, G.L.: Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. In: International Conference on Functional Programming, pp. 1–2. ACM, New York (2009)
39. Syme, D., Neverov, G., Margetson, J.: Extensible pattern matching via a lightweight language extension. In: International Conference on Functional Programming, pp. 29–40. ACM, New York (2007)
40. Thompson, S.: Lawful functions and program verification in Miranda. Science of Computer Programming 13(2-3), 181–218 (1990)
41. Tullsen, M.: First class patterns. In: Pontelli, E., Santos Costa, V. (eds.) PADL 2000. LNCS, vol. 1753, p. 1. Springer, Heidelberg (2000)
42. Wadler, P.: A critique of Abelson and Sussman: Why calculating is better than scheming. ACM SIGPLAN Notices 22(3), 83–94 (1987)
43. Wadler, P.: Views: A way for pattern matching to cohabit with data abstraction. In: Principles of Programming Languages, pp. 307–313. ACM, New York (1987)

Author Index