EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computing Science

Relational programming, program inversion

and the derivation of

parsing algorithms

by
Ed Knapen

Supervisor: Prof.dr. Roland Backhouse

Revised edition

November 23, 1993

Abstract

The spec calculus is a programming algebra that supports and even encourages the derivation of *relational* programs in a clear and *calculational* style.
After a brief introduction of the spec calculus two particular uses are investigated. First, the notion of program inversion is introduced. This notion from imperative programming allows one to construct from a given program R that satisfies $\{P\}\ R\ \{Q\}$ a program R^{-1} that satisfies $\{Q\}\ R^{-1}\ \{P\}$. It is shown that, as may be expected, a relational framework allows easy definition of and calculation with inverses of arbitrary programs. Several examples are discussed and used to introduce different methods to calculate implementable inverses.
The second topic is the derivation of precedence parsing algorithms. A simple algorithm for the calculation of the inorder traversal of a tree is transformed in two steps into an algorithm for precedence parsing. The first transformation is the use of program inversion to obtain an algorithm that constructs a tree from its inorder traversal. The second step is a restriction of the non-determinism of the algorithm. It is shown how the resulting algorithm corresponds to precedence parsing algorithms.

Contents

Acknowledgements **iii**

Preface **v**

1 Introduction **1**

2 The spec calculus **5**
- 2.1 Introduction 5
- 2.2 The lattice structure 5
- 2.3 Composition 6
- 2.4 Reverse 6
- 2.5 Monotypes 7
- 2.6 Domains 7
- 2.7 Imps an co-imps 7
- 2.8 Relators 8
- 2.9 Polynomial relators 8
  - 2.9.1 Monotypes 9
  - 2.9.2 Disjoint sum 9
  - 2.9.3 Cartesian product 11
- 2.10 Sections and points 13
- 2.11 Complemented domains and conditionals 13
  - 2.11.1 Domain complement 13
  - 2.11.2 Conditionals 13
- 2.12 Least fixed points 14
  - 2.12.1 Natural numbers 14
  - 2.12.2 Snoclists 15
- 2.13 Summary 16

3 Morphisms **17**
- 3.1 Introduction 17
- 3.2 The morphism theorem 17
- 3.3 The proof 19
- 3.4 Catamorphisms and anamorphisms 22
- 3.5 Paramorphisms 24
- 3.6 Modularisation 26
- 3.7 Tupling 28
- 3.8 Relationships between morphisms 30

3.9 Reductivity proofs 31
3.10 Summary 32

4 Program inversion **33**
4.1 Introduction 33
4.2 Inverting programs 33
4.3 Number representations 36
4.4 A calculated witness 40
4.5 Summary 45

5 Trees, traversals and precedence parsing **47**
5.1 Introduction 47
5.2 Labeled binary trees 47
5.3 First attempt 48
5.4 Second attempt 49
5.5 About halfway there 53
5.6 Removing the domain restrictions 55
5.7 Heaps 57
5.8 Precedence parsing 59
5.9 Summary 60

6 Conclusions and suggestions for further research **61**

Bibliography **63**

A Trees, traversals and precedence parsing **65**

Acknowledgements

Think where man's glory
most begins and ends,
And say my glory was
I had such friends.
William Butler Yeats

First of all I would like to thank Roland Backhouse for his continuous support and guidance. I am also greatly indebted to Netty van Gasteren who gave numerous improvements of the presentation of the work in this report.
Chritiene Aarts, Eerke Boiten, Henk Doornbos, Rik van Geldrop, Bart Gijsen, Paul Hoogendijk, Ed Voermans, Harold Weffers, Jaap van der Woude and all the other members of the 'Friday morning club' are gratefully acknowledged for many comments and suggestions.
Richard Verhoeven is acknowledged for all his help with the Mathpad editing system.
Last, but not least I would like to thank my parents and family, Judith Matthijsse, Richard Hensgens, Natasja Paulssen, Michel Reniers, Dany Benoit, Meivan Cheng, Maurice Satijn and Corrijan Geijtenbeek for their much needed and appreciated support.

Preface

I learn by going where I have to go
Roethke

On a cold February morning, we began our research now presented in this report. The research, under supervision of prof.dr. R.C. Backhouse at the Department of Mathematics and Computing Science of the Eindhoven University of Technology, extended over a period of eight months. During this time we investigated several uses of the calculus of relational programming that is developed by R.C. Backhouse and his colleagues. Our first goal was the study of the derivation of parsing algorithms, especially precedence parsing algorithms, in a relational framework like the spec calculus.

Later, during the bright days of spring, our attention shifted and for a while our research concentrated on the introduction of the intriguing concept of program inversion into the spec calculus.

After a little break in the midst of summer, new publications focussed our attention again on our original goals. The work of R. Leermakers, L. Augusteijn and F.E.J. Kruseman Aretz on functional parsing algorithms (cf. [Leerm0], [Leerm1], [Leerm2]) and the derivation by R. Bird and O. de Moor of Earley's algorithm in a relational setting (cf. [Bird]), were new inspirations for our own work.

Alas! Much too soon the temperature has dropped again and on this cold Wednesday afternoon in October, the last lines are being written to turn the work of eight months into a presentable form.

E.G. Knapen
Eindhoven, October 20, 1993

Chapter 1

Introduction

The spec calculus

Since the introduction of programmable machines people have been searching for suitable methods to develop programs to specify the behaviour of such machines. At first, such programs described the desired behaviour of a machine in the same way as recipes describe the actions of a cook preparing a dish. To support this programming style, called *imperative* programming, a great number of imperative programming languages were introduced, like *Pascal*, *Modula-2*, and *C*.

However, imperative programming is no longer the only available programming style. In fact, program derivation has become a major topic of research itself. One of the most important results of this research is the notion that programming is a mathematical activity that requires application of methods of mathematical understanding, calculation and proof. This notion has contributed to the introduction of several new programming styles, like *functional* programming.

But developments did not stop with the introduction of functional programming. Due to the unsatisfactory treatment of nondeterminism in a purely functional calculus attention moved to the development of a calculus for *relational* programming. By 1989 the spirit of relational calculi for software (and also hardware) development was in the air, and investigated by, amongst many others, R.C. Backhouse and his colleagues. A result of their research was the development of a truly polymorphic, relational programming calculus: the *spec calculus*.

A second goal of the research that led to the construction of the spec calculus, was the wish to develop a calculus that supported and even encouraged clear and economical calculations. Both imperative and functional programming make a ubiquitous use of bound variables. As a consequence formal manipulations often become unnecessary long and unwieldy.
In the realms of category theory and the Bird-Meertens formalism the minimisation of bound variables has long been advocated, but everyday programming practice is still far behind.

Relational programming

An important part of the introduction of a new programming calculus, like the spec calculus, is showing that it is powerful enough to solve the problems it has been designed for. From a theoretical point of view this has already been done: in [Backh] it is shown how the Bird-Meertens formalism can be rendered within the spec calculus. This formalism has already

been applied to a number of problems.
From a practical point of view this is an unsatisfying course, because it gives no clues of how the capacities of the new system can be used. In fact we restrict ourselves to the use of the constructs of a different formalism. If we want to assess the practicality of the spec calculus then we have to solve problems with it. Such work has been done by, for example, F. Rietman. In [Rietman] he explored the applicability of the system by solving a standard problem in computing science: segment sums.
This report is a continuation of the exploration of the applicability of the spec calculus. The fields from which we take our examples are program inversion and parsing algorithms.

Program inversion

Program inversion can be used as a technique for the derivation of programs. However, a relational framework is needed to define and calculate with inverses of arbitrary programs because such inverses are often non-deterministic. We show that in the spec calculus program inversion can be applied as a useful programming technique and give guidelines to ease such applications.

Parsing algorithms

We also show some first results of the application of the spec calculus in the area of the derivation of parsing algorithms and especially precedence parsing algorithms. In fact, many concepts in language theory are defined in terms of relations, like derivability, production rules, precedence relations, etc.
The choice for the topic parsing algorithms was made in a rather arbitrary way and the study of program inversion was more or less a consequence of this choice. Recent publications, for example the work of R. Leermakers, L. Augusteijn and F.E.J. Kruseman Aretz on functional parsing algorithms (cf. [Leerm0], [Leerm1], [Leerm2]) and the derivation by R. Bird and O. de Moor of Earley's algorithm in a relational setting (cf. [Bird]), and our own results, however, have shown that it was the right choice.

Overview

This report is more or less self contained. This means that a reader with some knowledge of imperative programming or (even better) of functional programming, combined with some familiarity with parsing theory, should be able to follow the main line of this report.
However, it is impossible to present in a report of this size all the theory behind the spec calculus. This is not because this theory is difficult or elaborate. The point is that the theory is built up from scratch in order to make it as calculational as possible. Therefore readers are advised to consult [Backh] in which they can find a complete introduction to the spec calculus.
In chapter 2 we present a short introduction to the spec calculus. Proofs of the properties and rules stated here can be found in [Backh].
In chapter 3 we introduce the notion of morphisms. Techniques from functional programming are used to derive relational programs, thereby showing their applicability.
In chapter 4 we introduce the concept of program inversion. We look for a definition of the

inverse of an arbitrary program and show how such inverses can be used as a programming technique.
In chapter 5 we use the technique of program inversion to invert a program that constructs the inorder traversal of a tree. By suitable adaptations we convert the resulting program into an algorithm for precedence parsing.
In chapter 6 we give some conclusions and identify problems that need further research.

Chapter 2

The spec calculus

2.1 Introduction

Programming is a game of *specifications* and *implementations*. The game starts with a specification: a formal description of the desired behaviour of a program. The goal is the derivation of an implementation: an executable program. Both specifications and implementations should therefore be expressions that are allowed in our programming calculus. One way to unify these different classes of expressions is to define them both as binary relations on input and output values. We use relations, rather than functions because we want to be able to work with non-deterministic programs.

Our programming calculus should therefore be a framework that allows the manipulation of relations. Furthermore, it should permit and even encourage clear and economical calculation. Unnecessary dummies, subscripts and arguments, for example, that obscure the interpretation of an expression should be avoided. For this reason we adopt an elegant algebraic setting as the basis of our calculus.

Preferably such an algebraic setting is built up in several layers and each layer is a well-known unity like a lattice or a monoid. The axioms used to define these unities should be phrased in a way suitable for calculation.

Fortunately, such a calculus does not have to be built from scratch. Tarski (cf. [Tarski]), Hoare and He (cf. [Hoare]) and many others have explored the relational calculus as a framework for programming. Based on this work the *spec calculus* has been constructed by Backhouse et al. (cf. [Backh]): a framework for the derivation of relational programs.

In this chapter we summarise the axioms and rules, together with their interpretations, of the spec calculus. A complete introduction, with proofs of the theorems stated in this chapter, can be found in [Backh].

2.2 The lattice structure

A binary relation on a set $\mathcal{A}$ is a subset of the cartesian product $\mathcal{A}\times\mathcal{A}$. A relation is thus an element of the powerset $\mathcal{P}(\mathcal{A}\times\mathcal{A})$. We call these relations *specs* (from specification) to avoid confusion between the objects we are going to discuss and the words we use in these discussions. The first axiom is that the specs form a complete lattice. The top of this lattice is denoted by $\top\!\!\top$, the bottom is denoted by $\perp\!\!\perp$. Their interpretations are respectively the total relation $\mathcal{A}\times\mathcal{A}$ and the empty relation. We write $\sqsubseteq$ for the order of the lattice and $\sqcup$ ('cup') and $\sqcap$ ('cap') for the join and the meet, respectively. For the cup we have

$$(2.1) \qquad R\sqcup S \ \sqsubseteq \ T \quad \equiv \quad R \sqsubseteq T \ \wedge \ S \sqsubseteq T \qquad .$$

The interpretation of the cup is the union of specs: $x(R\sqcup S)y \ \equiv \ x\ R\ y \ \vee \ x\ S\ y$.
For the cap we have

$$(2.2) \qquad T \ \sqsubseteq \ R\sqcap S \quad \equiv \quad T \sqsubseteq R \ \wedge \ T \sqsubseteq S \qquad .$$

Its interpretation is the intersection of specs: $x(R\sqcap S)y \ \equiv \ x\ R\ y \ \wedge \ x\ S\ y$.
The interpretation of the order is: $R \sqsubseteq S \ \equiv \ \forall(x,y :\ x\ R\ y :\ x\ S\ y)$.

2.3 Composition

Relations can be composed just like functions; the interpretation of such a composition is:

$$x\ (R \circ S)\ y \ \equiv \ \exists(z :\ x\ R\ z :\ z\ S\ y) \quad .$$

This composition is associative and has as unit the identity relation, so we have as an axiom:

$$(2.3) \qquad (\circ, I) \qquad \text{is a monoid.}$$

The interpretation of I is the identity relation: $x\ I\ y \ \equiv \ x = y$. The sections $(R\ \circ)$ and $(\circ\ R)$ distribute over arbitrary cups. As a consequence $\circ$ is monotonic in both arguments with respect to $\sqsubseteq$. From now on we just state that an operator is monotonic if the operator is monotonic with respect to $\sqsubseteq$.

2.4 Reverse

Unlike functions, relations can always be reversed. The interpretation of the reverse $R^{\cup}$ of a relation R is

$$x\ R^{\cup}\ y \ \equiv \ y\ R\ x \qquad .$$

The stirr frying pan symbol '$^{\cup}$' that we use as a unary postfix operator is pronounced accordingly as 'wok'.
We have as an axiom that the wok operator is its own Galois adjoint (an introduction into the theory of Galois adjoints can be found in [Backh]):

$$(2.4) \qquad R^{\cup} \sqsupseteq S \ \equiv \ R \sqsupseteq S^{\cup} \qquad .$$

A consequence of this axiom is that reverse is its own inverse: $R^{\cup\cup} = R$. Furthermore it follows that reverse distributes over arbitrary cups and caps. So we also have $\top\!\!\top^{\cup} = \top\!\!\top$, $\bot\!\!\bot^{\cup} = \bot\!\!\bot$ and reverse is monotonic.
Reverse and composition are related by the axiom:

$$(2.5) \qquad (R \circ S)^{\cup} \ = \ S^{\cup} \circ R^{\cup} \qquad .$$

A consequence of (2.5) is $I^{\cup} = I$; it is not very difficult to prove this from (2.4) and (2.5).

2.5 Monotypes

A relation R that satisfies $R \sqsubseteq I$ is called a *monotype*. We denote monotypes by uppercase identifiers A, B, C. Because these relations are subsets of I they satisfy $x\ A\ y \;\Rightarrow\; x = y$. So monotypes are partial identity relations and can be identified with subsets of the universe $\mathcal{A}$. Therefore we sometimes write $x \in A$ instead of $x\ A\ x$.
Monotypes have several very simple and convenient properties:

(2.6) $\qquad A = I \sqcap A = A^{\cup} = A \circ A \qquad ,$

(2.7) $\qquad A \circ B = B \circ A = A \sqcap B \qquad .$

2.6 Domains

We need to refer to the 'domain' and 'co-domain' of a spec. Instead of these two terms we use the terms *right-domain* and *left-domain*. For all specs R the right-domain $R{>}$ is defined by the Galois connection:

(2.8) $\qquad R{>} \sqsubseteq A \quad \equiv \quad R \ \sqsubseteq\ \top\!\!\top \circ A \qquad ;$

dually, the left-domain $R{<}$ is defined by:

(2.9) $\qquad R{<} \sqsubseteq A \quad \equiv \quad R \ \sqsubseteq\ A \circ \top\!\!\top \qquad .$

That these are good definitions follows from the fact that $(\top\!\!\top\ \circ)$ and $(\circ\ \top\!\!\top)$ distribute over arbitrary caps of monotypes. An immediate consequence of these definitions is that the domain operators distribute over arbitrary cups and therefore are monotonic. Some other properties are: $R{<} \circ R \ = \ R \ = \ R \circ R{>}$, $R^{\cup}{<} \ = \ R{>}$ and the dual $R^{\cup}{>} \ = \ R{<}$. There are still more properties that prove to be very useful. For the left-domain operator we have:

(2.10) $\qquad (R \circ S){<} \ = \ (R \circ S{<}){<} \qquad ,$

(2.11) $\qquad (R \circ S){<} \ \sqsubseteq\ R{<} \qquad .$

For convenience we also list the dual properties of the right-domain operator:

(2.12) $\qquad (R \circ S){>} \ = \ (R{>} \circ S){>} \qquad ,$

(2.13) $\qquad (R \circ S){>} \ \sqsubseteq\ S{>} \qquad .$

The interpretation of the left-domain of a spec R is $x \in R{<} \quad \equiv \quad \exists(y ::\ x\ R\ y)$. For the right-domain we have the dual $x \in R{>} \quad \equiv \quad \exists(y ::\ y\ R\ x)$.

2.7 Imps an co-imps

In this section we define two special classes of specs: *imps* and *co-imps*. The interpretation of an imp is a function: a zero- or single valued spec.

Definition 2.14 $\quad imp.R \quad \equiv \quad R \circ R^{\cup} \ \sqsubseteq\ I \ .$

The dual of this definition is:

Definition 2.15 $\quad co\text{-}imp.R \quad \equiv \quad R^{\cup} \circ R \ \sqsubseteq\ I \ .$

The interpretation of a co-imp is an *injective* spec; a spec R is injective if and only if for all x, y and z we have $y = z \quad \Leftarrow \quad x\ R\ y \ \wedge\ x\ R\ z$.
A spec is said to be a *bijection* if and only if it is both an imp and a co-imp.

2.8 Relators

A very important rôle in the definition of datatypes in the spec calculus is played by the *relators*.

Definition 2.16 A *relator* F is a function from specs to specs such that:

- $F.I \;\sqsubseteq\; I$,
- $F.R \;\sqsubseteq\; F.S \;\Leftarrow\; R \;\sqsubseteq\; S$,
- $F.(R \circ S) \;=\; F.R \circ F.S$,
- $F.(R^{\cup}) = (F.R)^{\cup}$.

This defines an *endorelator*: a unary relator from a given spec algebra $\mathcal{A}$ to itself. We can however extend the definition to functions that map an m-ary vector of specs into an n-ary vector of specs, for some natural numbers m and n. Since binary operators ($m = 2 \;\wedge\; n = 1$) will be used very often, we give their definition here. Let $\otimes$ be (the infix notation of) a binary relator. Then its defining properties are:

- $I{\otimes}I \;\sqsubseteq\; I$,
- $R{\otimes}U \;\sqsubseteq\; S{\otimes}V \;\Leftarrow\; R \sqsubseteq S \;\wedge\; U \sqsubseteq V$,
- $(R \circ S) \otimes (U \circ V) \;=\; R{\otimes}U \;\circ\; S{\otimes}V$,
- $R^{\cup} \otimes S^{\cup} \;=\; (R{\otimes}S)^{\cup}$.

A theorem relating the domain operators and relators is:

Theorem 2.17 If F is a relator then

$$F.(R^{>}) = (F.R)^{>}$$

and dually

$$F.(R^{<}) = (F.R)^{<} \quad .$$

□

2.9 Polynomial relators

An important class of relators are the *polynomial relators*. These are constructed using just three components: monotypes, a *disjoint sum* operator and a *cartesian product* operator. We present the axiomatisation of these components in this section.

2.9.1 Monotypes

We have already introduced the monotypes, but we now posit the existence of a special monotype, denoted by $\mathbb{1}$, pronounced as 'unit', and satisfying:

(2.18) $\quad \perp\!\!\perp \neq \mathbb{1}$

and

(2.19) $\quad I \;\sqsupseteq\; \mathbb{1} \circ \top\!\!\top \circ \mathbb{1} \qquad .$

The interpretation of $\mathbb{1}$ is a set the contains exactly one element; we denote this element by $\mathbb{1}$ too.

2.9.2 Disjoint sum

To define the disjoint sum operator we begin by postulating the existence of two specs: $\hookrightarrow$ (pronounced 'inject left') and $\hookleftarrow$ (pronounced 'inject right'). As an axiom we have that the injections are both imps:

(2.20) $\quad (\hookrightarrow \circ \hookrightarrow\cup) \sqcup (\hookleftarrow \circ \hookleftarrow\cup) \;\sqsubseteq\; I \qquad .$

Further, we define a binary operator $\triangledown$ (pronounced 'junc') on specs.

Definition 2.21 $\quad P \triangledown Q = (P \circ \hookrightarrow\cup) \;\sqcup\; (Q \circ \hookleftarrow\cup) \;\; .$

Finally, the disjoint sum operator $+$ (pronounced 'plus') is defined as follows:

Definition 2.22 $\quad P + Q = (\hookrightarrow \circ P) \;\triangledown\; (\hookleftarrow \circ Q) \;\; .$

For the interpretation of these constructs we assume that the universe is a term algebra formed by closing some base set under the unary operators $\hookrightarrow$ and $\hookleftarrow$. These are mappings from the term x to $\hookrightarrow.x$ and $\hookleftarrow.x$, respectively. For the $\triangledown$ we have that $x(P \triangledown Q)y$ if and only if

$$\exists(z :: \; y = \hookrightarrow.z \;\wedge\; x \; P \; z) \;\vee\; \exists(z :: \; y = \hookleftarrow.z \;\wedge\; x \; Q \; z) \qquad .$$

For the $+$ we have $x \; (P + Q) \; y$ if and only if

$$\begin{array}{ll} & \exists(u, v :: \; x = \hookrightarrow.u \;\wedge\; y = \hookrightarrow.v \;\wedge\; u \; P \; v) \\ \vee & \exists(u, v :: \; x = \hookleftarrow.u \;\wedge\; y = \hookleftarrow.v \;\wedge\; u \; Q \; v) \;\; . \end{array}$$

The following properties of $\triangledown$ and $+$ are often used:

Theorem 2.23 ($\triangledown +$ fusion)

$$P \triangledown Q \;\circ\; R + S \;=\; (P \circ R) \;\triangledown\; (Q \circ S) \;\; .$$

□

Theorem 2.24 (++ fusion)

$$P + Q \;\circ\; R + S \;=\; (P \circ R) \;+\; (Q \circ S) \;\; .$$

□

Corollary 2.25 $+$ is a relator.

□

Theorem 2.26 (Spec-$\triangledown$ fusion)

$$P \circ (Q \triangledown R) = (P \circ Q) \ \triangledown \ (P \circ R) \quad .$$

□

Theorem 2.27 (Computation rules for $\triangledown$)

$$P \triangledown Q \ \circ \ \hookrightarrow \ = \ P \quad ,$$

$$P \triangledown Q \ \circ \ \hookleftarrow \ = \ Q \quad .$$

□

Theorem 2.28 (Computation rules for $+$)

$$P + Q \ \circ \ \hookrightarrow \ = \ \hookrightarrow \circ P \quad ,$$

$$P + Q \ \circ \ \hookleftarrow \ = \ \hookleftarrow \circ Q \quad .$$

□

Theorem 2.29

$$(P+Q)> \ = \ P> \ + \ Q> \quad ,$$

$$(P+Q)< \ = \ P< \ + \ Q< \quad .$$

□

Theorem 2.30

$$(P \triangledown Q)> \ = \ P> \ + \ Q> \quad ,$$

$$(P \triangledown Q)< \ = \ P< \sqcup Q< \quad .$$

□

Theorem 2.31 (Cancellation properties) For all $\trianglelefteq \ \in \{\sqsubseteq, =, \sqsupseteq\}$:

$$P \triangledown Q \ \trianglelefteq \ R \triangledown S \ = \ P \trianglelefteq R \ \wedge \ Q \trianglelefteq S \quad ,$$

$$P + Q \ \trianglelefteq \ R + S \ = \ P \trianglelefteq R \ \wedge \ Q \trianglelefteq S \quad .$$

□

2.9.3 Cartesian product

Again we start by postulating the existence of two specs: $\ll$ (pronounced 'project left') and $\gg$ (pronounced 'project right'). The corresponding axiom is the dual of (2.20):

(2.32) $\quad (\ll \cup \circ \ll) \sqcap (\gg \cup \circ \gg) \sqsubseteq I \quad .$

A binary operator $\vartriangle$ (pronounced 'split') on specs is defined by:

Definition 2.33 $\quad P \vartriangle Q = (\ll \cup \circ P) \ \sqcap \ (\gg \cup \circ Q) \ .$

And finally, the cartesian product operator $\times$ (pronounced 'times') is defined by:

Definition 2.34 $\quad P \times Q = (P \circ \ll) \ \vartriangle \ (Q \circ \gg) \ .$

For the interpretation of the cartesian product we assume that the universe is also closed under the binary operator mapping the pair of terms x, y to the term (x,y). The interpretation of $\ll$ and $\gg$ is that they project a pair onto its left and right component, respectively.

For the $\vartriangle$ we have that $(x,y) \ (P \vartriangle Q) \ z$ if and only if

$$x \ P \ z \ \wedge \ y \ Q \ z \quad .$$

For the $\times$ we have $(u,v) \ (P \times Q) \ (x,y)$ if and only if

$$u \ P \ x \ \wedge \ v \ Q \ y \ .$$

Most of the properties of the $\triangledown$ and the $+$ can be dualised: for $\vartriangle$ and $\times$ we have the following useful properties:

Theorem 2.35 ($\times \vartriangle$ fusion)

$$P \times Q \ \circ \ R \vartriangle S \ = \ (P \circ R) \ \vartriangle \ (Q \circ S) \ .$$

□

Theorem 2.36 ($\times\times$ fusion)

$$P \times Q \ \circ \ R \times S \ = \ (P \circ R) \times (Q \circ S) \ .$$

□

Corollary 2.37 $\quad \times$ is a relator.

□

A complicating factor in our theory is the lack of a dual of theorem (2.26). Instead we have the following two theorems for $\vartriangle$.

Theorem 2.38 ($\vartriangle$-spec fusion)

$$(R \vartriangle S) \circ T \ \sqsubseteq \ (R \circ T) \ \vartriangle \ (S \circ T) \ .$$

□

Theorem 2.39 ($\vartriangle$-imp fusion) $\quad$ For all imps f we have

$$(R \vartriangle S) \circ f = (R \circ f) \;\vartriangle\; (S \circ f) \;\; .$$

□

Theorem 2.40 (Computation rules for $\vartriangle$)

$$\ll \;\circ\; P \vartriangle Q \;=\; P \circ Q{>} \;\; ,$$

$$\gg \;\circ\; P \vartriangle Q \;=\; Q \circ P{>} \;\; .$$

□

Theorem 2.41 (Computation rules for $\times$)

$$\ll \;\circ\; (P{\times}Q) = P \circ \ll \;\circ (\mathcal{I}{\times}Q{>}) \;\; ,$$

$$\gg \;\circ\; (P{\times}Q) = Q \circ \gg \;\circ (P{>}{\times}\mathcal{I}) \;\; .$$

□

Theorem 2.42

$$(P{\times}Q){>} \;=\; P{>} \times Q{>} \;\; ,$$

$$(P{\times}Q){<} \;=\; P{<} \times Q{<} \;\; .$$

□

Theorem 2.43 (Domain trading)

$$P \vartriangle Q = (P \circ Q{>}) \;\vartriangle\; (Q \circ P{>}) \;\; .$$

□

Corollary 2.44

$$A \sqsupseteq Q{>} \;\wedge\; B \sqsupseteq P{>} \quad\Rightarrow\quad P \vartriangle Q = (P \circ A) \;\vartriangle\; (Q \circ B) \;\; .$$

□

2.10 Sections and points

As an interface between relations and the elements of the universe of our interpretation, we introduce *points*. If p is a value of our universe then the relation $p^\bullet$ exists and has the following interpretation:

$$x\ p^\bullet\ y \quad \equiv \quad x = p \ .$$

We can use points to define a *section* of an arbitrary relation R with $R{>} \ \sqsubseteq \ I{\times}I$. We can define two sections: $(R\ p)$ and $(p\ R)$ for every value p of our universe. They have the following definitions:

Definition 2.45 $\quad (R\ p) \quad = \quad R \quad \circ \quad I \vartriangle p^\bullet \ .$

Definition 2.46 $\quad (p\ R) \quad = \quad R \quad \circ \quad p^\bullet \vartriangle I \ .$

The interpretation of a section $(R\ p)$ is:

$$x\ (R\ p)\ y \ \equiv \ x\ R\ (y,p) \ ,$$

for $(p\ R)$ we find:

$$x\ (p\ R)\ y \ \equiv \ x\ R\ (p,y) \ .$$

2.11 Complemented domains and conditionals

Our calculus would not be complete without any notion of a conditional (for example an $\underline{\text{if}} - \underline{\text{fi}}$ statement). To define conditionals, however, we first need to introduce the *complemented domains*.

2.11.1 Domain complement

The *complemented right-domain* of a spec R, denoted by $R{>}\!\bullet$ is defined by:

(2.47) $\quad R{>}\!\bullet \ \sqsupseteq \ A \quad \equiv \quad \perp\!\!\perp \ \sqsupseteq \ R \circ A \quad .$

One of the most important properties that we use is:

Lemma 2.48 $\quad R{>} \sqcup R{>}\!\bullet \ = \ I \quad$ and $\quad R{>} \sqcap R{>}\!\bullet \ = \ \perp\!\!\perp \ .$

The notion of the *complemented left-domain* can be defined by dualising the properties above. The interpretation of the complemented right-domain of a spec R is
$x \in R{>}\!\bullet \quad \equiv \quad \neg\exists(y :: \ y\ R\ x)$. Hence $x \in R{>}\!\bullet \quad \equiv \quad x \ \notin \ R{>}$.

2.11.2 Conditionals

Definition 2.49 (Conditional) For all specs P we define the binary operator $\lhd P \rhd$ by:

$$R \lhd P \rhd S = (R \circ P{>}) \sqcup (S \circ P{>}\!\bullet) \ .$$

□

The conditional $R \lhd P \rhd S$ can be viewed as a spec which applies R to the elements of $P{>}$ and applies S to the other ones.

2.12 Least fixed points

An important property of the lattice structure introduced in section 2.2 is the *Knaster-Tarski theorem.* Very briefly stated, it says that every monotonic function f has a least fixed point, i.e. the equation

$$X \;::\; X = f.X$$

has a smallest solution, which we denote by μf, that is characterised by the following two properties:

$$\mu f \;=\; f.\mu f$$

and, for all X,

$$\mu f \sqsubseteq X \quad \Leftarrow \quad f.X \sqsubseteq X \quad .$$

A theorem that relates the least fixed points of two monotonic functions is the *μ-fusion rule.*

Theorem 2.50 (μ-fusion) For all endofunctions f that have an adjoint

$$f.\mu g \;=\; \mu h \quad \Leftarrow \quad f \bullet g = h \bullet f \;\; .$$

□

We can combine the notions of least fixed points and relators to define monotypes. More specifically: the least fixed point of an arbitrary relator is a monotype.

Theorem 2.51 $\mu F \sqsubseteq I$ for all relators F.

Proof

$$
\begin{array}{ll}
 & \mu F \sqsubseteq I \\
\Leftarrow & \qquad \{\text{def. } \mu F\} \\
 & F.I \;\sqsubseteq\; I \\
\equiv & \qquad \{F \text{ is a relator }\} \\
 & \mathit{true} \quad .
\end{array}
$$

□

2.12.1 Natural numbers

As an example we look at the set containing all natural numbers, $\mathbb{N}$. To define this set we introduce the relator $\mathcal{N}$.

Definition 2.52 ($\mathcal{N}$) $\mathcal{N} = (\mathbb{1}+)$.

This means that $\mathcal{N}.X = \mathbb{1} + X$ for all specs X. The set containing all natural numbers is then represented in our calculus by $\mu\mathcal{N}$. The relator $\mathcal{N}$ is an instance of a special class of relators: it is a sum relator. As a consequence we can express the *constructors* of $\mu\mathcal{N}$:

$$
\begin{array}{ll}
 & \mu\mathcal{N} \\
= & \qquad \{\mu\mathcal{N} \text{ is a fixed point of } \mathcal{N}\} \\
 & \mathcal{N}.\mu\mathcal{N} \\
= & \qquad \{\text{def. } \mathcal{N}\} \\
 & 1\!\!1 \ + \ \mu\mathcal{N} \\
= & \qquad \{\text{def. } +\} \\
 & (\hookrightarrow \circ 1\!\!1) \ \triangledown \ (\hookleftarrow \circ \mu\mathcal{N}) \quad .
\end{array}
$$

We give the constructors special names, reflecting their interpretations.

Definition 2.53 $\quad zero = \hookrightarrow \circ 1\!\!1 \quad .$

Definition 2.54 $\quad succ \ = \ \hookleftarrow \circ \mu\mathcal{N} \quad .$

For our examples we introduce two relations that map a pair of natural numbers onto a natural number: *plus* and *times*. Their respective interpretations are the well-known addition and multiplication:

$$x \ plus \ (y,z) \quad \equiv \quad x = y + z$$

and

$$x \ times \ (y,z) \quad \equiv \quad x = y{*}z \quad .$$

Furthermore we define the sections: $(a*)$ and $(/a)$ for all natural numbers a. The interpretation of $(a*)$ is

$$x \ (a*) \ y \quad \equiv \quad x = a{*}y \quad ;$$

the interpretation of $(/a)$ is

$$x \ (/a) \ y \quad \equiv \quad x \ = \ y \ / \ a \quad ,$$

where / denotes the division operator on natural numbers (i.e. the result is a natural number).

2.12.2 Snoclists

Using the same method we can define the *snoclists*. We first introduce the relator $\mathcal{S}$.

Definition 2.55 ($\mathcal{S}$) $\quad \mathcal{S}.X \ = \ 1\!\!1 + (X{\times}I) \quad .$

In the same way as for the natural numbers we can calculate the constructors. We find that $\mu\mathcal{S} \ = \ nil \triangledown snoc$, where nil and $snoc$ are defined as follows.

Definition 2.56 $\quad nil = \hookrightarrow \circ 1\!\!1 \quad .$

Definition 2.57 $\quad snoc \ = \ \hookleftarrow \ \circ \ \mu\mathcal{S}{\times}I \quad .$

We introduce the spec *cat* that combines two snoclists into one, i.e. its interpretation is:

$$x \; cat \; (y, z) \quad \equiv \quad x = y +\!\!+ z \quad ,$$

where the infix operator $+\!\!+$ denotes the concatenation of two lists.

In calculations with the spec *cat* we use two properties. The first property is:

$$(2.58) \qquad (snoc \; v) \circ cat \quad = \quad cat \circ I \times (snoc \; v) \qquad ,$$

which follows from the associativity of *cat*. Furthermore, we have for an arbitrary spec R that satisfies $R< \;\sqsubseteq\; \mu S$:

$$(2.59) \qquad R \quad = \quad cat \;\circ\; R \vartriangle (nil \circ \top\!\!\top) \qquad ,$$

which follows from the fact that the empty list is the identity of the catenation.

2.13 Summary

We argued for and presented a framework for the derivation of relational programs from specifications. In the next chapters we apply the rules that were listed above. We claimed that the rules we presented are especially suited for clear calculations; this, of course, can only be established by performing calculations with them. In this sense, the remainder of this report can be seen as a motivation for adopting a relational programming calculus.

Chapter 3

Morphisms

3.1 Introduction

In relational programming the programs one derives are relations and these are defined (or described) by equations. Therefore equations with unique solutions are of a special interest. The structure of such an equation determines the structure of the program defined by it. This means that we can divide the set of all programs into classes of programs sharing the same structure. We call such classes *morphisms*. There is a breathtaking number of morphisms in the spec calculus: there are catamorphisms, paramorphisms, zygomorphisms, mutumorphisms, etc. Relations between morphisms allow us to transform a program with a certain structure into a program with a different structure. The relation expresses the fact that these two programs are equivalent or correspond in some other way. Therefore, such relations offer us rules for transformational programming and allow reasoning about programming at a high level of abstraction. Furthermore, morphisms have powerful calculational properties that allow one to use equational reasoning instead of, for example, induction.
In section 3.2 we present a theorem that captures all we need to know about morphisms in the following chapters. A proof is given in section 3.3. In section 3.4 we show how the well known *catamorphisms* can be defined using our theorem. In section 3.5 we define the *paramorphisms*. In section 3.6 we introduce *zygomorphisms* and show their relation to a programming technique known as modularisation in functional programming. In the same fashion section 3.7 introduces *dependent tupling morphisms*, which are related to the technique of tupling. In section 3.8 we look at a relation between these two morphisms. Finally, section 3.9 contains proofs of some reductivity properties. Readers familiar with the theory of reductivity may wish to skip this section.

3.2 The morphism theorem

The theorem we present deals with equations of the following shape

$$X \quad :: \quad X \;=\; R \circ H.X \circ \alpha \qquad .$$

This is a (severe) restriction because there are many equations with unique solutions that do not have this structure. But all morphisms we will encounter in this report have this shape, so the theorem is general enough for our needs. The relator H will often be defined in terms of one or more other relators that define the datatype on which the relation is to act. In such cases, an expression containing the least fixpoints of these relators is a likely candidate for the monotype A that is mentioned in (3.1): our main theorem on morphisms. An introduction of the notion of *reductivity* that is used in the theorem can be found in [Backh1]; we state the theorems we use explicitly in the section on reductivity at the end of this chapter.

Theorem 3.1 (Morphisms) Let R be an arbitrary spec, H a relator, α a spec and A a monotype such that

$$H.A \circ \alpha \;=\; \alpha \circ A \quad\wedge\quad \alpha \circ A \text{ is } H\text{-reductive.}$$

Then the equation

(3.2) $\qquad X \;::\; X \;=\; R \circ H.X \circ \alpha \circ A$

has a unique solution which is equal to the least solution of the equation

(3.3) $\qquad X \;::\; X \;=\; R \circ H.X \circ \alpha \qquad .$

We denote this unique solution by $\|[R]\|$ and pronounce it as *morphism* R. The fact that $\|[R]\|$ is the unique solution of (3.2) for fixed A and α can be expressed in a way that is very suitable for calculation; it is this rule that we refer to as the *unique extension property* (UEP):

(3.4) $\qquad X = \|[R]\| \quad\equiv\quad X \;=\; R \circ H.X \circ \alpha \circ A \qquad .$

Furthermore, $\|[R]\|$ satisfies the following properties (where $\trianglelefteq\; \in \{\sqsubseteq, =, \sqsupseteq\}$):
a *weak* unique extension property

(3.5) $\qquad X \trianglelefteq \|[R]\| \quad\Leftarrow\quad X \;\trianglelefteq\; R \circ H.X \circ \alpha \circ A \qquad ;$

a *fusion* law

(3.6) $\qquad T \circ \|[R]\| \;\trianglelefteq\; \|[S]\| \quad\Leftarrow\quad T \circ R \;\trianglelefteq\; S \circ H.T \qquad ;$

a monotonicity property

(3.7) $\qquad \|[R]\| \sqsupseteq \|[S]\| \;\Leftarrow\; R \sqsupseteq S \qquad ;$

and *domain properties*

(3.8) $\qquad \|[R]\| = \|[R \circ H.B]\| = \|[B \circ R]\| \quad\Leftarrow\quad B \;\sqsupseteq\; (R \circ H.B)< \qquad ,$

(3.9) $\qquad \|[R]\| = \|[R \circ H.R<]\| \qquad ,$

(3.10) $\qquad \|[R]\| = \|[R \circ B]\| \quad\Leftarrow\quad B \;\sqsupseteq\; H.R< \qquad .$

3.3 The proof

We start by showing that the least solution $\|[R]\|$ of (3.3) is a solution of (3.2). Then we show that the second part of the assumption implies that equation (3.2) has a unique solution. Finally we prove the other four properties stated in the theorem separately.

Let $\|[R]\|$ be the least solution of (3.3). If we define $h.X \;=\; R \circ H.X \circ \alpha$ then this is equivalent to $\|[R]\| = \mu h$. Since h is a monotonic function (composition and relators are monotonic) such a least fixpoint exists.
A central rôle in the proof of theorem (3.1) is played by the following lemma:

Lemma 3.11 $\circ A \;\bullet\; h \;=\; h \;\bullet\; \circ A$.

Proof

$$
\begin{array}{ll}
 & \circ A \;\bullet\; h \;=\; h \;\bullet\; \circ A \\
\equiv & \qquad \{\text{def. } h;\ \forall Y\} \\
 & R \circ H.Y \circ \alpha \circ A \;=\; R \circ H.(Y \circ A) \circ \alpha \\
\Leftarrow & \qquad \{\text{relator } H;\ \text{monotonicity}\} \\
 & \alpha \circ A \;=\; H.A \circ \alpha \\
\equiv & \qquad \{\text{assumption}\} \\
 & \mathit{true} \quad .
\end{array}
$$

□

Lemma 3.12 $\|[R]\| = \|[R]\| \circ A$.

Proof

$$
\begin{array}{ll}
 & \mu h \circ A \;=\; \mu h \\
\Leftarrow & \qquad \{\mu\text{-fusion}\} \\
 & \circ A \;\bullet\; h \;=\; h \;\bullet\; \circ A \\
\equiv & \qquad \{(3.11)\} \\
 & \mathit{true} \quad .
\end{array}
$$

□

Lemma 3.13 $X \;=\; h.X \circ A \;\;\equiv\;\; X = X \circ A \;\wedge\; X = h.X$.

Proof

$$
\begin{array}{ll}
 & X = X \circ A \;\wedge\; X = h.X \\
\equiv & \qquad \{\text{substitution}\} \\
 & X = X \circ A \;\;\wedge\;\; X \;=\; h.(X \circ A)
\end{array}
$$

$$\equiv \quad \{(3.11)\}$$

$$X = X \circ A \quad \wedge \quad X \ = \ h.X \circ A$$

$$\equiv \quad \{\text{monotypes}\}$$

$$X \ = \ h.X \circ A \ \ .$$

□

Corollary 3.14 By combining $\|[R]\| = h.\|[R]\|$ with (3.12) and (3.13) we find that $\|[R]\|$ solves (3.2).

Lemma 3.15 The equation $X \ :: \ X = R \circ H.X \circ \alpha \circ A$ has a unique solution.

Proof This follows directly from the assumption that $\alpha \circ A$ is H-reductive and the hylomorphism UEP (cf. [Backh1]).
□

Corollary 3.16 (UEP) $X = \|[R]\| \quad \equiv \quad X \ = \ R \circ H.X \circ \alpha \circ A \ \ .$

By (3.16) and the Knaster-Tarski theorem, we also have the weaker

Corollary 3.17 (wUEP) $X \ \trianglelefteq \ \|[R]\| \quad \Leftarrow \quad X \ \trianglelefteq \ R \circ H.X \circ \alpha \circ A \ \ .$

Lemma 3.18 (Fusion) $T \circ \|[R]\| \ \trianglelefteq \ \|[S]\| \quad \Leftarrow \quad T \circ R \ \trianglelefteq \ S \circ H.T \ \ .$

Proof

$$T \circ \|[R]\| \ \trianglelefteq \ \|[S]\|$$

$$\Leftarrow \quad \{(\text{wUEP})\}$$

$$T \circ \|[R]\| \ \trianglelefteq \ S \circ H.(T \circ \|[R]\|) \circ \alpha \circ A$$

$$\equiv \quad \{(\text{UEP}), \text{relator } H\}$$

$$T \circ R \circ H.\|[R]\| \circ \alpha \circ A \ \trianglelefteq \ S \circ H.T \circ H.\|[R]\| \circ \alpha \circ A$$

$$\Leftarrow \quad \{\text{invariance of } \trianglelefteq \text{ under } \circ\}$$

$$T \circ R \ \trianglelefteq \ S \circ H.T \ \ .$$

□

Lemma 3.19 (Monotonicity) $\|[R]\| \sqsupseteq \|[S]\| \ \Leftarrow \ R \sqsupseteq S \ \ .$

Proof

$$\|[R]\| \ \sqsupseteq \ \|[S]\|$$

$$\Leftarrow \quad \{\text{Fusion}, T \ := \ I\}$$

$$R \ \sqsupseteq \ S \circ H.I$$

$$\Leftarrow \quad \{\text{relator } H\}$$

$$R \ \sqsupseteq \ S \ \ .$$

$\Box$

Lemma 3.20 $\quad [\![R]\!] = [\![R \circ H.B]\!] \quad \Leftarrow \quad B \sqsupseteq (R \circ H.B)<$.

Proof

$$
\begin{array}{ll}
& [\![R]\!] = [\![R \circ H.B]\!] \\
\Leftarrow & \quad \{\text{calculus}\} \\
& [\![R]\!] = B \circ [\![R \circ H.B]\!] \quad \wedge \quad B \circ [\![R \circ H.B]\!] = [\![R \circ H.B]\!] \\
\Leftarrow & \quad \{\text{fusion law applied twice}\} \\
& \;\; B \circ R \circ H.B \;=\; R \circ H.B \\
& \wedge\; B \circ R \circ H.B \;=\; R \circ H.B \circ H.B \\
\equiv & \quad \{\text{monotypes: } H.B \circ H.B \;=\; H.B\} \\
& B \circ R \circ H.B \;=\; R \circ H.B \\
\equiv & \quad \{\text{def. } <\} \\
& B \sqsupseteq (R \circ H.B)< \quad .
\end{array}
$$

$\Box$

Lemma 3.21 $\quad [\![R]\!] = [\![B \circ R]\!] \quad \Leftarrow \quad B \sqsupseteq (R \circ H.B)<$.

Proof

$$
\begin{array}{ll}
& [\![R]\!] \\
\equiv & \quad \{\text{lemma (3.20): } B \sqsupseteq (R \circ H.B)<\} \\
& [\![R \circ H.B]\!] \\
\equiv & \quad \{B \sqsupseteq (R \circ H.B)<\} \\
& [\![B \circ R \circ H.B]\!] \\
\equiv & \quad \{\text{lemma (3.20): } B \sqsupseteq (B \circ R \circ H.B)<\} \\
& [\![B \circ R]\!] \quad .
\end{array}
$$

$\Box$

Corollary 3.22 If we instantiate B with $R<$ in lemma (3.20) we find:

$$[\![R]\!] = [\![R \circ H.R<]\!] \quad .$$

Corollary 3.23 Using (3.19) and (3.22) we find:

$$[\![R]\!] = [\![R \circ B]\!] \quad \Leftarrow \quad B \sqsupseteq H.R< \quad .$$

3.4 Catamorphisms and anamorphisms

One of the best studied morphisms in the spec calculus is the class of the catamorphisms. As a first application of theorem (3.1) we give a formal introduction of these morphisms. The catamorphism $(\![R]\!)$ is defined as the least solution of the following equation:

$$(3.24) \qquad X \quad :: \quad X \; = \; R \circ F.X \qquad .$$

When necessary to avoid confusion we write $(\![F;\ R]\!)$ instead of $(\![R]\!)$.
To apply theorem (3.1) it is easier to write (3.24) as

$$(3.25) \qquad X \quad :: \quad X \; = \; R \circ F.X \circ I \qquad .$$

Using this definition we see that to apply theorem (3.1) we have to find a monotype A such that:

$$F.A \circ I \; = \; I \circ A$$

and

$I \circ A$ is F-reductive.

A good guideline is to try an expression containing the least fixed points of the relators used to define the morphism. In the case of the catamorphisms it is easy to see that μF satisfies both these requirements. Therefore we can instantiate all the properties of the theorem, like the UEP:

$$(3.26) \qquad X = (\![R]\!) \quad \equiv \quad X \; = \; R \circ F.X \circ \mu F \qquad .$$

From (3.26) we can see that

$$(3.27) \qquad (\![R]\!)> \; \sqsubseteq \; \mu F \qquad .$$

To get familiar with the properties of catamorphisms we give a short example of the derivation of a program that has this structure. Consider the following specification of a program S:

$$x \; S \; y \; \equiv \; x = a^y \qquad .$$

Since this is a problem concerning the natural numbers, we look at catamorphisms defined on this datatype, i.e. we instantiate F with $\mathcal{N}$. If we let Q denote $(\![R \triangledown S]\!)$ we have:

$$(3.28) \qquad Q \circ zero = R \circ 1\!\!1 \qquad \text{and}$$

$$(3.29) \qquad Q \circ succ = S \circ Q \qquad .$$

We prove (3.29); the proof of (3.28) has the same structure.

$$
\begin{array}{ll}
 & Q \circ succ \\
= & \quad \{(3.24)\ ;\ \text{def. } Q\} \\
 & R \triangledown S \;\circ\; \mathcal{N}.Q \;\circ\; succ \\
= & \quad \{\text{def. } \mathcal{N} \text{ and } succ\} \\
 & R \triangledown S \;\circ\; \mathbb{1} + Q \;\circ\; \hookleftarrow \;\circ\; \mu\mathcal{N} \\
= & \quad \{\text{comp. rule } +\} \\
 & R \triangledown S \;\circ\; \hookleftarrow \;\circ\; Q \;\circ\; \mu\mathcal{N} \\
= & \quad \{\text{comp. rule } \triangledown\ ;\ Q{>} \sqsubseteq \mu\mathcal{N}\} \\
 & S \circ Q \quad .
\end{array}
$$

We head for a definition of S that fits the pattern of (3.28) and (3.29), so we make the following calculations:

$$
\begin{array}{ll}
 & x\ S\ 0 \\
\equiv & \quad \{\text{specification of } S\} \\
 & x \,=\, a^{0} \\
\equiv & \quad \{\text{arithmetic}\} \\
 & x \,=\, 1
\end{array}
$$

and

$$
\begin{array}{ll}
 & x\ S\ (n+1) \\
\equiv & \quad \{\text{specification of } S\} \\
 & x \,=\, a^{n+1} \\
\equiv & \quad \{\text{arithmetic}\} \\
 & x \,=\, a{*}a^{n} \\
\equiv & \quad \{\text{spec. of } S\} \\
 & \exists(y :: \ x \,=\, a{*}y \ \wedge \ y\ S\ n) \\
\equiv & \quad \{\text{def. } \circ\} \\
 & x\ ((a*) \circ S)\ n \quad .
\end{array}
$$

As a result from these two calculations we have

$$S \circ zero \,=\, 1^{\bullet} \qquad \text{and}$$

$$S \circ succ \,=\, (a*) \circ S \qquad .$$

This neatly matches the pattern of (3.28) and (3.29), so we conclude that

$$S \,=\, (\!|1^{\bullet} \triangledown (a*)|\!)$$

is a program that satisfies the given specification.

There is another morphism that has strong connections to the class of the catamorphisms: the class of the *anamorphisms*. The anamorphism $[\![F;\ R]\!]$ is defined by

(3.30) $$[\![F;\ R]\!] \ = \ (\!|F;\ R^{\cup}|\!)^{\cup} \qquad .$$

From this definition we have the following property: $[\![F;\ R]\!]$ is the unique solution of the equation:

(3.31) $$X \ :: \ X \ = \ \mu F \circ F.X \circ R \qquad .$$

All the properties of the catamorphisms can be transformed into properties of anamorphisms in the same way. We have, for example,

(3.32) $$[\![F;\ R]\!] \ = \ F.[\![F;\ R]\!] \circ R \qquad .$$

3.5 Paramorphisms

Suppose you want to define the factorial, $n!$, on the natural numbers. We call this spec *fac* and calculate:

$$
\begin{array}{ll}
 & x \ fac \ 0 \\
\equiv & \qquad \{\text{specification of } fac\} \\
 & x \ = \ 0! \\
\equiv & \qquad \{\text{arithmetic}\} \\
 & x \ = \ 1 \quad .
\end{array}
$$

$$
\begin{array}{ll}
 & x \ fac \ (n+1) \\
\equiv & \qquad \{\text{specification of } fac\} \\
 & x \ = \ (n+1)! \\
\equiv & \qquad \{\text{arithmetic}\} \\
 & x \ = \ (n+1) \ * \ n! \\
\equiv & \qquad \{?\} \\
 & \ldots
\end{array}
$$

There is no way to rewrite the last line into the shape that fits the pattern of a catamorphism, due to the occurrence of the subexpression n in $(n+1)$. If you look again at (3.29) you see that we can only use the value of $n!$ to calculate $(n+1)!$. But we need the value of n too.

For this reason we introduce the *paramorphisms*. The paramorphism $[\![R]\!]$ is defined as the least solution of

(3.33) $$X \ :: \ X \ = \ R \circ F.(X \vartriangle I) \qquad .$$

We use the by now familiar recipe: we first rewrite (3.33) into a shape that eases application of theorem (3.1). In this case we find

$$(3.34) \qquad X :: X \;=\; R \circ F.(X{\times}I) \circ F.(I \vartriangle I) \quad .$$

The next step is the formulation of the proof obligations. For the paramorphisms we have to find a monotype A such that:

$$F.(A{\times}I) \circ F.(I \vartriangle I) \;=\; F.(I \vartriangle I) \circ A$$

and

$F.(I \vartriangle I) \circ A$ is $F \bullet (\times I)$-reductive.

Our first choice, since paramorphisms are defined in terms of the relator F, is the monotype μF. The proof of the first property is a simple application of the domain trading rule for the $\vartriangle$-operator. The reductivity property is proven in the section on reductivity.
As a result of theorem (3.1) we have the following UEP and fusion law:

$$X = [\![R]\!] \quad\equiv\quad X \;=\; R \circ F.(X \vartriangle I) \circ \mu F \quad ,$$

$$T \circ [\![R]\!] = [\![S]\!] \quad\Leftarrow\quad T \circ R \;=\; S \circ F.(T{\times}I) \quad .$$

After this very short introduction of the paramorphisms (it is theorem (3.1) that allows us to be so brief), we return to the problem of deriving a spec implementing the factorial.
If we instantiate F with $\mathcal{N}$ we find for $Q = [\![R \triangledown S]\!]$:

$Q \circ \mathit{zero} = R \circ 1\!\!1$ and

$Q \circ \mathit{succ} \;=\; S \circ Q \vartriangle I$.

One can calculate these two properties by following the same steps as in the proofs of (3.28) and (3.29) of the previous section. We now resume our calculation, but our goal has changed: we are now heading for a paramorphism:

$$\begin{array}{ll}
 & x\ \mathit{fac}\ (n+1) \\
\equiv & \qquad \{\text{specification of } \mathit{fac}\} \\
 & x = (n+1)! \\
\equiv & \qquad \{\text{arithmetic}\} \\
 & x \;=\; (n+1) * n! \\
\equiv & \qquad \{\text{spec. } \mathit{fac}\} \\
 & \exists(y :: x = (n+1){*}y \;\wedge\; y\ \mathit{fac}\ n) \\
\equiv & \qquad \{\text{calculus}\}
\end{array}$$

$$\exists(y,z :: \; x = (z+1)*y \;\wedge\; z = n \;\wedge\; y \; fac \; n)$$

$\equiv$ {def. $\vartriangle$ }

$$\exists(y,z :: \; x = (z+1)*y \;\wedge\; (y,z) \; (fac \vartriangle I) \; n)$$

$\equiv$ {def. $\circ$}

$$x \; (times \circ I\times succ \circ fac \vartriangle I) \; n \quad .$$

This time our calculation is successful; we are able to express *fac* as a paramorphism:

$$fac = [\![1^\bullet \triangledown (times \circ I\times succ)]\!] \quad .$$

3.6 Modularisation

The remainder of this chapter can be seen as one example showing how theorem (3.1) can be used to introduce morphisms that are needed for an application and how the different properties of morphisms are used in proofs. We look at two programming techniques known from functional programming and show that each corresponds with a morphism. We then prove a relation between these morphisms.

The following example will be used to introduce the programming techniques. Suppose we want to derive a relation with the following specification:

(3.35) $\quad x \; R \; y \;\equiv\; x = \Sigma(i : 0{\le}i{<}y : a^i) \quad .$

Because $\Sigma(i : 0{\le}i{<}0 : a^i) = 0$ we have $x \; R \; 0 \;\equiv\; x = 0$. For the other case we calculate:

$$x \; R \; (y+1)$$

$\equiv$ {spec. R}

$$x = \Sigma(i : 0{\le}i{<}y+1 : a^i)$$

$\equiv$ {range split}

$$x = \Sigma(i : 0{\le}i{<}y : a^i) + a^y$$

$\equiv$ {spec. R}

$$\exists(z :: \; x = z + a^y \;\wedge\; z \; R \; y)$$

$\equiv$ {$\bullet$ $w \; S \; y \;\equiv\; w = a^y$; def. *plus*}

$$\exists(w,z :: \; x \; plus \; (z,w) \;\wedge\; z \; R \; y \;\wedge\; w \; S \; y)$$

$\equiv$ {def. $\vartriangle$ }

$$\exists(w,z :: \; x \; plus \; (z,w) \wedge (z,w) \; R \vartriangle S \; y)$$

$\equiv$ {def. $\circ$}

$$x \; (plus \circ R \vartriangle S) \; y \quad .$$

We arrive at the following specification:

(3.36) $\quad R \circ zero = zero \qquad$ and

(3.37) $\quad R \circ \mathit{succ} \;=\; \mathit{plus} \circ R \vartriangle S \quad ,$

where

(3.38) $\quad x \; S \; y \;\equiv\; x = a^y \quad .$

If we encounter a subexpression that is not a permissible expression in our program notation, there are a number of programming techniques that can be applied to continue the derivation. The expression that is not permissible can be considered as a new problem, for which we have to derive a separate program. This approach is called *modularisation* in [Hooge]. In the case of our example, this boils down to constructing a spec S that satisfies (3.38). In the previous section we solved this problem and found a definition for S. However, it is not necessary to do so at this moment. The only decision we have to make is that S will be implemented by a catamorphism, say $(\!|T|\!)$ that satisfies (3.38). Then we have found the following definition of a program:

(3.39) $\quad R \circ \mathit{zero} = \mathit{zero}$ $\quad$ and

(3.40) $\quad R \circ \mathit{succ} \;=\; \mathit{plus} \circ R \vartriangle (\!|T|\!) \quad .$

Programs of this shape are zygomorphisms. The zygomorphism $[\![R, S]\!]$ is defined as (cf. [Voerm]) the least solution of

(3.41) $\quad X \;::\; X \;=\; R \circ F.(X \vartriangle (\!|S|\!)) \quad .$

We use the morphism theorem to show that the least solution of (3.41) is the unique solution of some equation and hence corresponds to a morphism. To ease the application of the theorem we rewrite (3.41) into the equivalent

(3.42) $\quad X \;::\; X \;=\; R \circ F.(I \times (\!|S|\!)) \circ F.(X \times I) \circ F.(I \vartriangle I) \quad .$

First observe that $F \bullet (\times I)$ is a relator because F and $(\times I)$ are relators. Next, it is easy to check that $A = \mu F$ satisfies the conditions of the theorem, i.e.

$$F.(\mu F \times I) \circ F.(I \vartriangle I) \;=\; F.(I \vartriangle I) \circ \mu F$$

and

$$F.(I \vartriangle I) \circ \mu F \;\text{ is } F \bullet (\times I)\text{-reductive.}$$

The first part is easily proved using the domain trading rule for $\vartriangle$. The proof of the second part is given in the section on reductivity.
We will state the UEP for zygomorphisms explicitly:

(3.43) $\quad X = [\![R, S]\!] \;\equiv\; X \;=\; R \circ F.(X \vartriangle (\!|S|\!)) \circ \mu F \quad .$

If we instantiate F with $\mathcal{N}$ and let Q denote $[\![R \triangledown S, T]\!]$ then we have

(3.44) $\quad Q \circ \mathit{zero} = R \circ \mathbb{1}$ $\quad$ and

(3.45) $\quad Q \circ \mathit{succ} \;=\; S \circ Q \vartriangle (\!|T|\!) \quad .$

We prove (3.45), the proof of (3.44) has the same structure.

$$
\begin{array}{ll}
 & Q \circ succ \\
= & \quad \{\text{def. Q; (3.41)}\} \\
 & R \triangledown S \;\circ\; \mathcal{N}.(Q \vartriangle (\![T]\!)) \;\circ\; succ \\
= & \quad \{\text{def. } \mathcal{N},\ succ\} \\
 & R \triangledown S \;\circ\; \mathbb{1} + (Q \vartriangle (\![T]\!)) \;\circ\; \hookleftarrow \;\circ\; \mu\mathcal{N} \\
= & \quad \{\text{computation rule } +\} \\
 & R \triangledown S \;\circ\; \hookleftarrow \;\circ\; Q \vartriangle (\![T]\!) \;\circ\; \mu\mathcal{N} \\
= & \quad \{\triangledown \text{ comp. rule, domains}\} \\
 & S \;\circ\; Q \vartriangle (\![T]\!) \;\; .
\end{array}
$$

So our program defined by (3.39) and (3.40) fits this pattern neatly and if we plug in the definition for S that we found in the previous section, we find that $R = [\![zero \triangledown plus,\, 1^{\bullet} \triangledown (a*)]\!]$ solves our problem. The advantage of this technique is that subproblems can be dealt with in isolation. A disadvantage is that it often leads to inefficient programs; we will see that the program of our example can be made more efficient by 'merging' the computation of $(\![S]\!)$ with the main computation. Therefore a combination with the second technique we present, which is a tool to increase the efficiency of programs, would be very interesting.

3.7 Tupling

The other programming technique we look at is *tupling* (cf. [Hooge]). Its purpose is to increase the efficiency of a program by calculating two (or more) relations that have the same domain and the same pattern of recursion in one computation. Therefore we introduce a new relation, with the same domain as the original ones, whose value is a tuple; the elements of this tuple are the values of the original relations. Applying this technique to our example means that we have a new specification:

(3.46) $\quad (x,y)\ R\ z \ \equiv\ x = \Sigma(i : 0{\le}i{<}z : a^i) \ \wedge\ y = a^z \quad .$

Again we will head for a definition in terms of the constructors of $\mathcal{N}$:

(3.47) $\quad R \circ zero = zero \vartriangle 1^{\bullet}$

is easily found by combining derivations we have already made. Since

$$
\begin{array}{ll}
 & (x,y)\ R\ (z+1) \\
\equiv & \quad \{\text{spec. } R\} \\
 & x = \Sigma(i : 0{\le}i{<}z+1 : a^i) \ \wedge\ y = a^{z+1} \\
\equiv & \quad \{\text{range split}\} \\
 & x = \Sigma(i : 0{\le}i{<}z : a^i) + a^z \ \wedge\ y = a{*}a^z \\
\equiv & \quad \{\text{spec. } R\} \\
 & \exists(p,u :: x = p+u \ \wedge\ (p,u)\ R\ z) \wedge \exists(q,v :: y = a{*}v \ \wedge\ (q,v)\ R\ z)
\end{array}
$$

$$\equiv \qquad \{\text{def. } \circ \text{ and } \gg\}$$

$$x\ (plus \circ R)\ z \ \wedge\ y\ ((a*) \circ \gg \circ R)\ z$$

$$\equiv \qquad \{\text{def. } \vartriangle\}$$

$$(x,y)\ ((plus \circ R) \vartriangle ((a*) \circ \gg \circ R))\ z \quad ,$$

we have

$$(3.48) \qquad R \circ succ = (plus \circ R) \vartriangle ((a*) \circ \gg \circ R) \quad .$$

A program of this shape is a *dependent tupling morphism* (dtm). This morphism can be found in [Voerm], although it is not explicitly named. A dtm $<R,S>$ is defined as the least solution of

$$(3.49) \qquad X \ :: \ X = (R \circ F.X) \vartriangle (S \circ F.(\gg \circ X)) \quad .$$

Again we will rewrite the defining equation to apply the theorem: (3.49) is equivalent to

$$(3.50) \qquad X \ :: \ X \ = \ R \times (S \circ F.\gg) \ \circ \ F.X \times F.X \ \circ \ I \vartriangle I \quad .$$

To satisfy the assumption of the morphism theorem we have to find a monotype A such that

$$F.A \times F.A \ \circ \ I \vartriangle I \ = \ I \vartriangle I \circ A$$

and

$$I \vartriangle I \circ A \ \text{ is } (X \mapsto F.X \times F.X)\text{-reductive.}$$

The first requirement is easily dealt with by substituting μF for A. This choice also satisfies the second requirement, as is shown in the section on reductivity. (If you are familiar with the theory of reductivity you might want to prove this requirement as an exercise and skip that section.) We state one of the results of the morphism theorem explicitly here: the UEP for dtm's.

$$(3.51) \qquad X = <R,S> \ \equiv \ X = (R \circ F.X) \vartriangle (S \circ F.(\gg \circ X)) \ \circ \ \mu F \quad .$$

If we instantiate F with $\mathcal{N}$ again and let Q denote $<R \triangledown S, T \triangledown U>$, we find that

$$(3.52) \qquad Q \circ zero = (R \circ 1\!\!1) \vartriangle (T \circ 1\!\!1) \qquad \text{and}$$

$$(3.53) \qquad Q \circ succ = (S \circ Q) \vartriangle (U \circ \gg \circ Q) \quad .$$

Aided by the previous examples, the reader should by now be able to prove this. Our definition of R has exactly this pattern, so $<zero \triangledown plus, 1^\bullet \triangledown (a*)>$ is a neatly defined program that solves our (new) specification. A solution to the original problem can easily be constructed using this program.

3.8 Relationships between morphisms

We have seen some examples of morphisms (zygomorphism, dtm) together with programs that belong to these classes. The derivations we have made and the techniques we have used are straightforward adaptations from the realm of functional programming, showing that such techniques are also applicable in relational programming. The identification of the program structures as morphisms enables us to calculate with programs at (yet) a higher level of abstraction. An example of this from functional programming is the *tail recursion theorem* (cf. [Hooge]). We illustrate this with an easier example relating the morphisms we have discussed.

Theorem 3.54 $<R,S> = [\![R,S]\!] \vartriangle (\!|S|\!)$.

Proof We use (3.51): let $X = [\![R,S]\!] \vartriangle (\!|S|\!)$. We have to prove

$$(R \circ F.X) \vartriangle (S \circ F.(\gg \circ X)) \;\circ\; \mu F \;=\; X \;.$$

We calculate

$$\begin{array}{ll}
& (R \circ F.X) \vartriangle (S \circ F.(\gg \circ X)) \quad \circ \quad \mu F \\
= & \qquad \{\gg \circ X \;=\; (\!|S|\!) \circ [\![R,S]\!]>\} \\
& (R \circ F.X) \vartriangle (S \;\circ\; F.(\!|S|\!) \;\circ\; F.[\![R,S]\!]>) \quad \circ \quad \mu F \\
= & \qquad \{\text{domain trading: see appendix}\} \\
& (R \circ F.X) \vartriangle (S \circ F.(\!|S|\!)) \quad \circ \quad \mu F \\
= & \qquad \{\vartriangle\text{-imp fusion}\} \\
& R \circ F.X \circ \mu F \quad \vartriangle \quad S \circ F.(\!|S|\!) \circ \mu F \\
= & \qquad \{\text{UEP's ; def. } X\} \\
& X \;.
\end{array}$$

□

How should we interpret this as a rule for program transformation? The answer to this question is that a program constructed using modularisation, which contains other (sub)programs with the same domain and the same recursion pattern, can be rewritten (without the need for new calculations) into a more efficient program that computes all properties in one pass.

Exercise Above we considered the tupling of two programs with a one-way dependency. An easier problem is the tupling of two independent programs. This is often done to calculate unrelated properties in one computation. Such a program can be defined as the least solution of

$$X \;::\; X \;=\; (R \circ F.(\ll \circ X)) \vartriangle (S \circ F.(\gg \circ X))$$

and could be called an *independent tupling morphism* (itm). Show that the least solution of this equation equals $(\!|R|\!) \vartriangle (\!|S|\!)$.

□

3.9 Reductivity proofs

We conclude this chapter with proofs of the reductivity properties that were used in the previous section. We used three such properties:

- μF is F-reductive;
- $F.(I \vartriangle I) \circ \mu F$ is $F \bullet (\times I)$-reductive;
- $I \vartriangle I \circ \mu F$ is $(X \mapsto F.X \times F.X)$-reductive.

For the readers who are not familiar with the work on reductivity as it is stated in [Backh1] we will state the three lemmas that will be used:

Lemma 3.55 μF is F-reductive for all relators F.

Lemma 3.56 If R is F-reductive and $S \sqsubseteq R$ then S is F-reductive.

Lemma 3.57 If Q is F-reductive and α is a spec such that for all monotypes A we have $\alpha \circ F.A \sqsubseteq G.A \circ \alpha$ then $\alpha \circ Q$ is G-reductive.

We see that the first reductivity property we want to prove follows directly from lemma (3.55). For the other two properties we have the following lemmas.

Lemma 3.58 $F.(I \vartriangle \top\!\!\top) \circ \mu F$ is $F \bullet (\times I)$-reductive.

Proof From lemma (3.55) we have that μF is F-reductive. Next we use lemma (3.57):

$$
\begin{array}{ll}
 & F.(I \vartriangle \top\!\!\top) \circ F.A \ \sqsubseteq\ F.(A \times I) \circ F.(I \vartriangle \top\!\!\top) \\
\equiv & \qquad \{\text{relator, } \vartriangle\text{-imp fusion and } \times\vartriangle\text{-fusion}\} \\
 & F.(A \ \vartriangle\ \top\!\!\top \circ A) \ \sqsubseteq\ F.(A \vartriangle \top\!\!\top) \\
\Leftarrow & \qquad \{\text{monotonicity of } F \text{ and } \vartriangle\} \\
 & \mathit{true} \quad .
\end{array}
$$

□

Corollary 3.59 $F.(I \vartriangle B) \circ \mu F$ is $F \bullet (\times I)$-reductive for all monotypes B. (Use lemma (3.56)).

Lemma 3.60 $B \vartriangle B \circ \mu F$ is G-reductive for all monotypes B, where $G.X = F.X \times F.X$.

Proof From lemma (3.55) we have that μF is F-reductive. Next we use lemma (3.57):

$$
\begin{array}{ll}
 & B \vartriangle B \circ F.A \ \sqsubseteq\ F.A \times F.A \ \circ\ B \vartriangle B \\
\equiv & \qquad \{\vartriangle\text{-imp and } \times\vartriangle\text{-fusion}\} \\
 & (B \circ F.A) \vartriangle (B \circ F.A) \ \sqsubseteq\ (F.A \circ B) \vartriangle (F.A \circ B) \\
\equiv & \qquad \{\text{monotypes}\} \\
 & \mathit{true} \quad .
\end{array}
$$

□

3.10 Summary

We introduced a powerful and useful theorem that greatly simplifies the introduction of a broad and interesting class of morphisms. We saw several examples of morphisms that will turn up again later in this report. Many techniques from functional programming were used to derive relational programs, showing their applicability in the spec calculus. We also saw how rules for transformational programming can be expressed in our calculus.

Chapter 4

Program inversion

4.1 Introduction

'Beauty is truth, truth beauty,' - that is all
Ye known on earth, and all ye need to know.
Keats, Ode on a Grecian Urn

In the previous chapter we introduced two elementary programming techniques that are well known in the realm of functional programming. In this chapter we introduce another programming technique, known as *program inversion.* This is not yet a well established technique, although the concept of program inversion can be traced back to a publication of E.W. Dijkstra and W.H.J. Feijen in 1978 (cf. [Dijkstra0]). Subsequently, the idea was explored in [Gries0] and used in, for example, [Gries1]. At that time the notion of program inversion was rather informal and the inverted programs still needed full proofs of correctness. It was [Chen] that provided a formal basis for program inversion. In this paper proof rules were introduced for the inversion of the constructs of Dijkstra's guarded command language. These rules support stepwise program inversion. The paper was followed by many other publications in which the same proof rules were used (e.g. [Schoen], [Sneps]).
More recently the concept of program inversion was introduced into the setting of a functional programming calculus (see for example [Harri] and [Moor]). Another recent development was the introduction of program inversion into the refinement calculus in [Back] and [Wright].
In this chapter we investigate how program inversion can be used to derive programs in the spec calculus.

4.2 Inverting programs

What do we mean when we say that a program is an inverse of another program? We start our search for an answer to this question in the world of imperative programming. In [Gries0] the inverse P^{-1} is specified by the requirement that 'running P followed by P^{-1} would be the same as not running any program at all.' This definition is very intuitive, but its use is rather restricted: only programs that yield different results for different inputs have an inverse in this sense. Consider for example the program that computes the square of its input, where the input is an integer. If the output of this program is 4, then the input is 2 or -2, but it is impossible to determine which of the two values was the original input.

So only 'injective' programs have an inverse according to this definition. In a functional setting we encounter the same problem: only an injective function from A to B has an inverse of the type from B to A. However, by allowing set-valued functions we can overcome this problem (see for example [Moor]). The inverse of a function f is denoted by f^{-1} and defined by

(4.1) $\quad f^{-1}.x \;=\; \{y \mid f.y = x\} \quad .$

In this definition there is an asymmetry between the type of the argument and the type of the result. The argument is an element of some set while the result is a set. Therefore, in [Harri] *all* functions are taken to be of the type 'from a set of values to a set of values,' so that inverses can be freely composed with any function.

In a relational setting such requirements are unnecessary. Every function is a relation; instead of $x = f.y$ we write $x\ f\ y$. In such a setting (4.1) becomes

(4.2) $\quad y\ f^{-1}\ x \;\equiv\; x\ f\ y \quad .$

Looking again at (4.2), one might recognise that it is in fact the interpretation of the reverse operator! So the spec calculus is equipped with an operator to specify the inverse of an expression. Properties of this operator correspond directly to rules for program inversion. Look for example at one of the defining axioms:

(4.3) $\quad (R \circ S)\cup \;=\; S\cup \circ R\cup \quad .$

This equation has a direct translation to a well known rule for the inversion of a composition of statements in an imperative program (cf. [Gries0]):

(4.4) $\quad (R; S)^{-1} \;=\; S^{-1}\ ;\ R^{-1} \quad .$

Another very nice property we have is that the reverse operator is its own inverse, i.e.

(4.5) $\quad (R\cup)\cup = R \quad .$

So R is the inverse of $R\cup$: inverting twice yields a program that is equivalent to the original program. As a last motivation for taking the reverse of a spec as its inverse, we take another look at Gries's definition with which we started this section: $P^{-1} \circ P \;=\; \mathit{skip}$. A direct translation in the spec calculus would be:

(4.6) $\quad P^{-1} \circ P \;=\; P> \quad .$

Since there is no value x such that $x\ P\ y$ holds if $y \notin P>$ we have to replace *skip* by $P>$, i.e. a partial skip. Substituting $\cup$ for $^{-1}$ in (4.6) this requirement becomes equivalent to the requirement that P is a co-imp:

$$
\begin{array}{ll}
 & \mathit{co\text{-}imp}.P \\
\equiv & \qquad \{\text{def. co-imp}\} \\
 & P\cup \circ P \;\sqsubseteq\; I \\
\equiv & \qquad \{R \sqsubseteq S \;\equiv\; R = R \sqcap S\} \\
 & P\cup \circ P \;=\; P\cup \circ P \;\sqcap\; I \\
\equiv & \qquad \{P> \;=\; P\cup \circ P \;\sqcap\; I\} \\
 & P\cup \circ P \;=\; P> \;.
\end{array}
$$

We have found a formal definition of the notion of an inverse of a program. But do we allow reverses of arbitrary specs as implementations or only as specifications of programs? So far, we have not made a clear distinction between specifications and implementations. As the name *imps* suggests, the intended interpretation of an imp is an implementation. It is, however, not the intention that *all* implementations are imps. In general, an expression of the shape $R^\cup$ is a specification of a program that might not offer information for a direct implementation. Its interpretation is

$$x \; R^\cup \; y \;\; \equiv \;\; y \; R \; x \quad ,$$

i.e. the output x is only *indirectly* characterised: it should be such that $y \; R \; x$ holds. Given a program R this would mean we have to try input values x until we find one that generates the output y .

For this reason we classify the reverse of an arbitrary spec as a specification that is not an implementation. Only reverses of the basic bijective constructs of our calculus, like the injections $\hookrightarrow$ and $\hookleftarrow$, are admitted as implementations. The construction of an implementation of an inverse of a spec is therefore often a process of 'eliminating the woks in a expression.'

The next question is how to use the existence of inverses in the derivation of programs. Such a derivation starts by specifying a requirement a spec has to satisfy. This specification is often of the following shape:

$$x \; R \; y \equiv p.x.y \qquad ,$$

where $p.x.y$ is a predicate in x and y. Instead of deriving R from this specification, we 'invert' the problem: we exchange the dummies x and y and derive a program S that satisfies this new specification:

$$x \; S \; y \equiv p.y.x \qquad .$$

After this derivation we construct the reverse of S, that is $S^\cup$. We can show that $S^\cup$ satisfies our original specification:

$$\begin{array}{ll} & x \; S^\cup \; y \\ \equiv & \qquad \{\text{interpretation of } S^\cup\} \\ & y \; S \; x \\ \equiv & \qquad \{\text{specification of } S\} \\ & p.x.y \quad . \end{array}$$

Clearly this technique is only of use if it is easier to construct a spec 'satisfying' $p.y.x$ than one that satisfies $p.x.y$. Consider for example $x = y^2$ and $y = x^2$. In fact there are only differences between these two predicates if a sense of direction is associated to the specs. $p.x.y$ should be interpreted as an assertion that has to be met for input y and output x. Then $x = y^2$ should be interpreted as 'the output is the square of the input' and $y = x^2$ as 'the output is such that squared it equals the input.' It is easier to compute the square of a number, than it is to compute a root. Part of this difference comes from the possibility that more than one root exists.

To conclude this section we summarise the technique of program inversion. Of some problems the inverse is a simpler problem. We can derive a solution for this simpler problem and construct its reverse. This reverse is a solution of the original problem.

In this chapter we give a two examples to show how program inversion is used. The examples illustrate two different methods for the construction the reverse of a spec, each with its own advantages and disadvantages. For both methods we develop some guidelines that will be of help in the construction of reverses. We start with an easy problem where the relation to be constructed is a bijection.

4.3 Number representations

Consider the following problem.

Problem Let N be a natural number that satisfies $N>0$. Derive a program P that takes as input a natural number x and gives as output a list that is the representation of x in the N-ary system. If we take $N=2$, for example, then *[11011]* P *27* should hold because *11011* is the binary representation of the number *27*.
□

This is not a very difficult problem, but the inverse problem of calculating the natural number represented by such a list is even simpler. The technique of program inversion allows one to 'invert' the resulting program into one that solves the original problem!

First we have to construct a program S that solves the 'inverted' problem. By constructing the reverse of S we find a program that satisfies the specification of program P that was given in the problem. The specification of S is:

(4.7) $\quad x\ S\ y \equiv \quad x =$ "the value represented by y".

A number is represented by exactly one list, that is, if we require that all list of length greater than 1 starting with a '0' are excluded. The list *[05]* is an example of a list that we do not allow because it represents the same value as *[5]*. We could decide to define our datatype representing the lists in such a way that this requirement is satisfied. However, this would lead to a complicated definition of the datatype that would obscure the main derivation. Instead we will use a simple definition of lists, even though this will force us to use the fact that there is no list with length greater than 1 that starts with a '0' in our derivation.

To define the datatype representing lists we first define the monotype D that contains the elements of such lists. Remember that $\mu\mathcal{N}$ represents the natural numbers. We define D as the set containing the natural numbers smaller than N. Hence $D \sqsubseteq \mu\mathcal{N}$. Next we introduce the relator F to define the non-empty snoclists over D.

Definition 4.8 $\quad F.X \;=\; D + (X\times D) \quad .$

F is a grounded relator, i.e. a sum relator with a constant relator as one component of the sum. The most important aspect of a sum relator F is that we can express the constructors of μF. They can be calculated with the following derivation:

$$\mu F$$

$$
\begin{array}{ll}
= & \quad \{\mu F \text{ is a fixed point of } F\} \\
& F.\mu F \\
= & \quad \{\text{def. } F\} \\
& D + (\mu F \times D) \\
= & \quad \{\text{def. } +\} \\
& (\hookrightarrow \circ D) \ \triangledown \ (\hookleftarrow \circ \mu F \times D) \quad .
\end{array}
$$

We introduce the names *wrap* and *snocp* for the constructors of F:

Definition 4.9 $\quad \mathit{wrap} = \hookrightarrow \circ D$

Definition 4.10 $\quad \mathit{snocp} \ = \ \hookleftarrow \circ \mu F \times D$

To conclude our introduction of the non-empty snoclists over D we give the instantiation of catamorphisms on such lists. For arbitrary specs R and T we have:

(4.11) $\quad (\!| R \triangledown T |\!) \circ \mathit{wrap} = R \circ D \quad$ and

(4.12) $\quad (\!| R \triangledown T |\!) \circ \mathit{snocp} \ = \ T \circ (\!| R \triangledown T |\!) \times D \quad .$

We prove (4.12), verification of (4.11) is left to the reader.

$$
\begin{array}{ll}
& (\!| R \triangledown T |\!) \circ \mathit{snocp} \\
= & \quad \{\text{catamorphisms}\} \\
& R \triangledown T \circ F.(\!| R \triangledown T |\!) \circ \mathit{snocp} \\
= & \quad \{\text{def. } F \text{ and } \mathit{snocp}\} \\
& R \triangledown T \ \circ \ D + ((\!| R \triangledown T |\!) \times D) \ \circ \ \hookleftarrow \ \circ \ \mu F \times D \\
= & \quad \{\text{computation rule for } +\} \\
& R \triangledown T \ \circ \ \hookleftarrow \ \circ \ (\!| R \triangledown T |\!) \times D \ \circ \ \mu F \times D \\
= & \quad \{\text{computation rule for } \triangledown\} \\
& T \ \circ \ (\!| R \triangledown T |\!) \times D \ \circ \ \mu F \times D \\
= & \quad \{\times\times \text{ fusion}\} \\
& T \circ ((\!| R \triangledown T |\!) \circ \mu F) \times (D \circ D) \\
= & \quad \{(\!| F;\ R |\!)> \sqsubseteq \mu F \ ; \text{monotypes}\} \\
& T \circ (\!| R \triangledown T |\!) \times D \quad .
\end{array}
$$

□

As stated earlier, applying the technique of program inversion is only useful when the inversion of the problem yields a simpler problem. For our example this certainly is the case: a definition for S can easily be derived following the examples of the previous chapter. We leave this as an exercise.

Exercise Show that $S = (\!|I \triangledown (plus \circ (N*)\times I)|\!)$ satisfies the specification of S given in (4.7).
□

To find a solution of the original problem we now calculate a reverse of S. Motivations for the first two steps will be given following the derivation.

$$
\begin{array}{ll}
 & (\!|I \;\triangledown\; plus \circ (N*)\times I|\!)^{\cup} \\
= & \quad \{(\!|R|\!) = (\!|R \circ F.I|\!)\ ;\ \triangledown + \text{fusion}\} \\
 & (\!|D \;\triangledown\; plus \circ (N*)\times D|\!)^{\cup} \\
= & \quad \{\text{domain restriction: see below}\} \\
 & (\!|D \;\triangledown\; D{>\!\bullet} \circ plus \circ (N*)\times D|\!)^{\cup} \\
= & \quad \{(\!|R|\!)^{\cup} = [\![R^{\cup}]\!]\} \\
 & [\![(D \;\triangledown\; D{>\!\bullet} \circ plus \circ (N*)\times D)^{\cup}]\!] \\
= & \quad \{(R \triangledown S)^{\cup} \;=\; \hookrightarrow \circ R^{\cup} \;\sqcup\; \hookleftarrow \circ S^{\cup}\} \\
 & [\![\hookrightarrow\circ D^{\cup} \;\sqcup\; \hookleftarrow \circ ((D{>\!\bullet}) \circ plus \circ (N*)\times D)^{\cup}]\!] \\
= & \quad \{A^{\cup} = A;\ (4.3)\} \\
 & [\![\hookrightarrow\circ D \;\sqcup\; \hookleftarrow \circ (plus \circ (N*)\times D)^{\cup} \circ D{>\!\bullet}]\!] \ .
\end{array}
$$

In the first line of the derivation we recognise an expression of the shape $(\!|R|\!)^{\cup}$. Our goal therefore is the application of the rule $(\!|R|\!)^{\cup} = [\![R^{\cup}]\!]$. A (cata-)morphism 'hides' a lot of information like domain restrictions, however, so we had better make this information more explicit before we start to construct a reverse of the subexpression R. This motivates the first step of the derivation.
In the second step we used the fact that there are no lists with length greater than 1 that start with a '0'. Hence, for $snoc.(x,a)$, i.e. the list obtained by adding the element a to the list x, we know that $S.x \geq 1$ and hence $N * S.x \;\geq\; N$.
Such reasoning could have been avoided had we chosen a definition of the datatype representing lists that excluded such lists. As said before this would have obscured the main argument too much, hence our choice. Note that the result of this step is again a domain restriction.
In the last line we encounter the subexpression $(plus \circ (N*)\times D)^{\cup}$. We study this expression in isolation:

$$
\begin{array}{ll}
 & (plus \circ (N*)\times D)^{\cup} \\
= & \quad \{(4.3)\} \\
 & ((N*)\times D)^{\cup} \circ plus^{\cup} \\
= & \quad \{(R \otimes S)^{\cup} \;=\; R^{\cup} \otimes S^{\cup} \text{ for binary relator } \otimes\} \\
 & (N*)^{\cup} \times D^{\cup} \;\circ\; plus^{\cup} \\
= & \quad \{\text{arithmetic, monotypes}\}
\end{array}
$$

$$(/N){\times}D \circ plus^{\cup}$$

$$= \qquad \{\text{domains; } \times\times\text{fusion}\}$$

$$(/N){\times}I \;\circ\; (/N)_{>} \times D \;\circ\; plus^{\cup} \quad .$$

Let's assume for a moment that we have an inverse of *plus*. This relation $plus^{\cup}$ satisfies

$$(x,y)\ plus^{\cup}\ z \;\equiv\; z = x + y \qquad .$$

So on input of a natural number, $plus^{\cup}$ returns a pair of natural numbers such that the sum of the components is equal to the input. In the subexpression $(/N)_{>}{\times}D \;\circ\; plus^{\cup}$, however, there are severe restrictions on the elements of such a pair. Using these domain restrictions enables us to avoid constructing $plus^{\cup}$. To show this we make the following point-wise calculation, in which we denote $(/N)_{>}$ by A for the sake of brevity.

$$(x,y)\ (A{\times}D \circ plus^{\cup})\ z$$

$$\equiv \qquad \{\text{interpretations}\}$$

$$x{\in}A \;\wedge\; y{\in}D \;\wedge\; x + y = z$$

$$\equiv \qquad \{\text{def. } D\}$$

$$x{\in}A \;\wedge\; 0{\le}y{<}N \;\wedge\; x + y = z$$

$$\equiv \qquad \{\text{def. } A\text{; arithmetic}\}$$

$$x = N{*}(z \ \underline{\mathsf{div}}\ N) \;\wedge\; y = z\ \underline{\mathsf{mod}}\ N$$

$$\equiv \qquad \{div = (\ \underline{\mathsf{div}}\ N),\ mod = (\ \underline{\mathsf{mod}}\ N)\text{; interpretation of } \vartriangle \text{ and } \circ\}$$

$$(x,y)\ (((N{*}) \circ div) \vartriangle mod)\ z \quad .$$

With this result we continue our point-free calculation:

$$(/N){\times}I \;\circ\; (/N)_{>}{\times}D \;\circ\; plus^{\cup}$$

$$= \qquad \{\text{calculation above}\}$$

$$(/N){\times}I \circ ((N{*}) \circ div) \vartriangle mod$$

$$= \qquad \{\times \vartriangle \text{ fusion}\}$$

$$((/N) \circ (N{*}) \circ div) \vartriangle mod$$

$$= \qquad \{(/N) \circ (N{*}) = \mu\mathcal{N} = div_{<}\ \}$$

$$div \vartriangle mod \quad .$$

We found that $(plus \circ (N{*}){\times}D)^{\cup} = div \vartriangle mod$ without constructing $plus^{\cup}$ itself. We use this rule to finish the main derivation; we start again by repeating the last line:

$$[\![\hookrightarrow{\circ}D \;\sqcup\; \hookleftarrow \circ (plus \circ (N{*}){\times}D)^{\cup} \circ D_{>\bullet}\ ]\!]$$

$$= \qquad \{\text{monotypes, } (plus \circ (N{*}){\times}D)^{\cup} = div \vartriangle mod\}$$

$$[\![\hookrightarrow{\circ}D{\circ}D \;\sqcup\; \hookleftarrow \circ div \vartriangle mod \circ D_{>\bullet}\ ]\!]$$

$$= \qquad \{\text{def. conditionals, } wrap\}$$

$$[\![wrap \lhd D \rhd (\hookleftarrow \circ div \vartriangle mod)]\!] \quad .$$

We found that $P = [\![wrap \lhd D \rhd (\hookleftarrow \circ div \vartriangle mod)]\!]$ solves the problem we stated at the beginning of this section. We conclude by rephrasing P in a shape that will be more familiar to most readers. To do so we consider a slightly more general expression.
Let $R = [\![wrap \lhd D \rhd \hookleftarrow \circ T]\!]$. We calculate

$$\begin{array}{ll}
 & R \\
= & \qquad \{[\![S]\!] \;=\; F.[\![S]\!] \circ S\} \\
 & D + (R{\times}D) \;\circ\; wrap \lhd D \rhd \hookleftarrow \circ T \\
= & \qquad \{\text{def. conditional}\} \\
 & D + (R{\times}D) \;\circ\; (wrap{\circ}D \;\sqcup\; \hookleftarrow \circ T \circ D{>}\!\bullet) \\
= & \qquad \{\text{distribution}\} \\
 & (D + (R{\times}D) \circ wrap \circ D) \;\sqcup\; (D + (R{\times}D) \circ \hookleftarrow \circ T \circ D{>}\!\bullet) \\
= & \qquad \{\text{def. } wrap \text{ ; computation rule for } + \text{ ; monotypes}\} \\
 & (\hookrightarrow \circ D \circ D) \;\sqcup\; (\hookleftarrow \;\circ\; R{\times}D \;\circ\; T \circ D{>}\!\bullet) \\
= & \qquad \{\text{def. } wrap\text{; } R{<} \sqsubseteq \mu F \text{ ; } {\times}{\times}\text{fusion}\} \\
 & (wrap \circ D) \;\sqcup\; (\hookleftarrow \;\circ\; \mu F{\times}D \;\circ\; R{\times}I \circ T \circ D{>}\!\bullet) \\
= & \qquad \{\text{def. } snocp\text{, conditional}\} \\
 & wrap \lhd D \rhd snocp \circ R{\times}I \circ T \quad .
\end{array}$$

Hence, for P, our solution of the problem, we have

$$P \;=\; wrap \lhd D \rhd (snocp \circ P{\times}I \circ div \vartriangle mod) \quad .$$

This property directly translates into a functional equivalent:

$$P.x \;=\; \underline{\mathsf{if}}\; x{<}N \;\to\; [x] \quad \Box \quad x{\geq}N \;\to\; snocp.(P.(x \;\underline{\mathsf{div}}\; N), x \;\underline{\mathsf{mod}}\; N)\; \underline{\mathsf{fi}} \quad .$$

4.4 A calculated witness

In the example of the previous section we saw that we do not always want to calculate reverses in a completely compositional way. Recall that we found an expression for $(/N){>}{\times}D \;\circ\; plus^{\cup}$ without constructing $plus^{\cup}$ itself. The alternative would have been to construct the reverse of $plus^{\cup}$, only to discover that in the presence of the monotype $(/N){>}{\times}D$ this complicated spec simplifies to $div \vartriangle mod$.

In the present section we take a closer look at this alternative method of constructing reverses. The example we use comes from [Backh0]. We start by giving a short introduction to the relevant theory (cf. [Backh0], [Weffers0], [Weffers1]).

An important question one can ask about datatypes is whether two datatypes are *isomorphic*, i.e. equivalent modulo implementation details. A second question is whether one datatype *simulates* another. For relators F and G we have that F simulates G if there is some spec, called the *witness*, that enables one to transform all computations involving G-structures to computations involving F-structures.

One way to prove that two relators F and G are naturally isomorphic is to construct two specs, one witnessing that F simulates G, and one witnessing that G simulates F, and then show that these specs are reverses of each other. This is done in [Backh0]. An alternative would be to construct a witness to one simulation, derive its reverse and finally check that the latter witnesses the other simulation. In this section we consider a simplified instance of the second approach: the construction of the reverse of a witness.

The spec of which we want to construct the reverse is a witness of the simulation of the relator $(F+G)\star$ by the relator H. First we introduce the definitions used to construct these relators.

Definition 4.13 ($\oplus_F$) Let F be an arbitrary relator and X and Y arbitrary specs. The binary relator $\oplus_F$ is defined by:

$$X \oplus_F Y \quad = \quad X + F.Y \quad .$$

Definition 4.14 ($F\star$) Let F be an arbitrary relator and X an arbitrary spec. The unary relator $F\star$ is defined by:

$$F\star.X \quad = \quad \mu(X\oplus_F) \qquad .$$

Definition 4.15 For relators F and G and an arbitrary spec X, the relator $(F+G)$ is defined by:

$$(F+G).X \quad = \quad F.X \;\; + \;\; G.X \qquad .$$

Definition 4.16 (H) The relator H is defined by $H \quad = \quad G\star \;\; \bullet \;\; (F \bullet G\star)\star \quad .$

In [Backh0] we find a witness R_0 for the simulation of $(F+G)\star$ by H, that is defined by (for reasons of simplicity we call it R):

Definition 4.17 (R) $R = (\!|I\oplus_{(F+G)};\; (\hookrightarrow \circ \hookrightarrow) \triangledown ((\hookrightarrow \circ \hookleftarrow) \triangledown \hookleftarrow)|\!) \quad .$

We shall construct $R^{\cup}$, but not in a compositional way. Instead we specify and solve an equation of which $R^{\cup}$ is the solution, using properties of R and the relators that we introduced above and a theorem that more or less reflects Gries's definition of an inverse. We first state the theorem.

Theorem 4.18 (reverse) Let γ be an arbitrary imp. Then

$$\delta = \gamma^{\cup} \quad \Leftarrow \quad \delta \circ \gamma = \gamma> \;\; \wedge \;\; \delta> \sqsubseteq \gamma< \quad .$$

Proof

$$
\begin{array}{ll}
 & \gamma^{\cup} \\
= & \qquad \{\gamma \;=\; \gamma \circ \gamma> \;;\; (4.3)\} \\
 & \gamma> \circ \gamma^{\cup} \\
= & \qquad \{\delta \circ \gamma = \gamma>\} \\
 & \delta \circ \gamma \circ \gamma^{\cup} \\
= & \qquad \{\mathit{imp}.\gamma \text{ hence } \gamma \circ \gamma^{\cup} \;=\; \gamma<\} \\
 & \delta \circ \gamma< \\
= & \qquad \{\delta> \;\sqsubseteq\; \gamma<\} \\
 & \delta \;\;.
\end{array}
$$

□

The first conjunct of the antecedent corresponds with Gries's definition; the second conjunct is needed to obtain the desired result.

A property of $F\star$ that we use is a fusion law for $I\oplus_F$ catamorphisms:

Lemma 4.19 $\quad (\!|I\oplus_F;\; R \triangledown S|\!) \;\circ\; F\star.X \quad = \quad (\!|I\oplus_F;\; (R \circ X) \triangledown S|\!) \;\;.$

Proof

$$
\begin{array}{ll}
 & (\!|R \triangledown S|\!) \;\circ\; F\star.X \\
= & \qquad \{\text{def. } F\star \;;\; \text{map fusion:} \\
 & \qquad\quad (\!|I\oplus_F;\; U|\!) \circ \mu(X\oplus_F) \;=\; (\!|I\oplus_F;\; U \circ X \oplus_F I|\!)\} \\
 & (\!|R \triangledown S \;\circ\; X + F.I|\!) \\
= & \qquad \{\triangledown + \text{fusion}\} \\
 & (\!|(R\circ X) \triangledown S \;\circ\; I + F.I|\!) \\
= & \qquad \{(\!|G;\; R \circ F.I|\!) = (\!|G;\; R|\!)\} \\
 & (\!|(R\circ X) \triangledown S|\!) \;\;.
\end{array}
$$

□

Lemma 4.20 For $I\oplus_F$ catamorphisms we have:

$$(\!|R \triangledown S|\!) \circ \hookrightarrow = R \quad \wedge \quad (\!|R \triangledown S|\!) \circ \hookleftarrow = S \circ F.(\!|R \triangledown S|\!) \;\;.$$

Proof

$$
\begin{array}{ll}
 & (\!|R \triangledown S|\!) \circ \hookrightarrow \\
= & \qquad \{\text{catamorphism}\} \\
 & R \triangledown S \;\circ\; I + F.(\!|R \triangledown S|\!) \;\circ\; \hookrightarrow \\
= & \qquad \{\triangledown + \text{fusion, comp. rule } \triangledown\} \\
 & R \;\;.
\end{array}
$$

$$(\!|R \triangledown S|\!) \circ \hookleftarrow$$

$$= \quad \{\text{catamorphism}\}$$

$$R \triangledown S \quad \circ \quad I + F.(\!|R \triangledown S|\!) \quad \circ \quad \hookleftarrow$$

$$= \quad \{\ \triangledown + \text{fusion, comp. rule } \triangledown\ \}$$

$$S \circ F.(\!|R \triangledown S|\!) \quad .$$

□

Lemma (4.20) will be the main motivation behind the assumptions we are going to make.

Properties of R that we can find in [Backh0] are:

- R is an imp,
- $R\!> \ = \ (F+G)\star\ .\ I$,
- $R\!< \ = H.I$, where $H \ = \ G\star \ \bullet \ (F \bullet G\star)\star$.

We use theorem (4.18) to construct the reverse of R. Since the first conjunct of the assumption seems to capture the notion of inverse, it is a logical point to start. Note that $R\!> \ = \ (F+G)\star\ .\ I \ = \ \mu(X \mapsto I+(F.X+G.X))$, so, if we substitute U for the unknown of theorem (4.18) we have to solve the following equation:

$$U \quad :: \quad U \circ R = \mu(X \mapsto I + (F.X + G.X)) \quad .$$

We calculate

$$U \circ R = \mu(X \mapsto I + (F.X + G.X))$$

$$\equiv \quad \{\mu F \ = \ (\!|F;\ F.I|\!);\ \text{def}\ +\}$$

$$U \circ R = (\!|\hookrightarrow \triangledown (\hookleftarrow \quad \circ \quad F.I + G.I)|\!)$$

$$\equiv \quad \{\text{def. } R\}$$

$$U \circ (\!|(\hookrightarrow \circ \hookrightarrow) \triangledown ((\hookrightarrow \circ \hookleftarrow) \triangledown \hookleftarrow)|\!) = (\!|\hookrightarrow \triangledown (\hookleftarrow \quad \circ \quad F.I + G.I)|\!)$$

$$\Leftarrow \quad \{\text{fusion law and } \triangledown \text{ cancellation}\}$$

$$U \circ \hookrightarrow \circ \hookrightarrow \ = \ \hookrightarrow \circ I$$

$$\wedge\ U \quad \circ \quad (\hookrightarrow \circ \hookleftarrow) \triangledown \hookleftarrow \quad = \quad \hookleftarrow \quad \circ \quad F.I + G.I \quad \circ \quad F.U + G.U$$

$$\equiv \quad \{\circ \text{ distributes over } \triangledown\ ;\ ++ \text{ fusion; def. } +\}$$

$$U \circ \hookrightarrow \circ \hookrightarrow \ = \ \hookrightarrow$$

$$\wedge\ (U \circ \hookrightarrow \circ \hookleftarrow) \triangledown (U \circ \hookleftarrow) \quad = \quad \hookleftarrow \quad \circ \quad (\hookrightarrow \circ F.U) \triangledown (\hookleftarrow \circ G.U)$$

$$\equiv \quad \{\circ \text{ distributes over } \triangledown\ ;\ \triangledown \text{ cancellation}\}$$

$$U \circ \hookrightarrow \circ \hookrightarrow \ = \ \hookrightarrow$$

$$\wedge\ U \circ \hookrightarrow \circ \hookleftarrow \quad = \quad \hookleftarrow \circ \hookrightarrow \circ F.U$$

$$\wedge\ U \circ \hookleftarrow \quad = \quad \hookleftarrow \circ \hookleftarrow \circ G.U \quad .$$

We have to construct U in such a way that it satisfies the equalities in the last line of the derivation. To do so we deal with each conjunct separately. We start with the last one:

$$
\begin{array}{ll}
 & U \circ \hookleftarrow \;=\; \hookleftarrow \circ \hookleftarrow \circ G.U \\
\equiv & \quad \{\bullet\ U = (\!|I \oplus_G;\ V \triangledown W|\!);\ \text{lemma (4.20)}\} \\
 & W \circ G.U \;=\; \hookleftarrow \circ \hookleftarrow \circ G.U \\
\equiv & \quad \{\bullet\ W = \hookleftarrow \circ \hookleftarrow\} \\
 & \mathit{true} \quad .
\end{array}
$$

Ignoring the conjunct that looks most complicated for the moment, we proceed with the first one. In this way, we collect as much information as possible about the unknown U. Information that is needed to deal with the more complicated second conjunct.

$$
\begin{array}{ll}
 & U \circ \hookrightarrow \circ \hookrightarrow = \hookrightarrow \\
\equiv & \quad \{U = (\!|I \oplus_G;\ V \triangledown W|\!);\ \text{lemma (4.20)}\} \\
 & V \circ \hookrightarrow = \hookrightarrow \\
\equiv & \quad \{\bullet\ V = (\!|I \oplus_L;\ \hookrightarrow \triangledown X|\!);\ \text{lemma (4.20)}\} \\
 & \mathit{true} \quad .
\end{array}
$$

Finally we deal with the second conjunct.

$$
\begin{array}{ll}
 & U \circ \hookrightarrow \circ \hookleftarrow \;=\; \hookleftarrow \circ \hookrightarrow \circ F.U \\
\equiv & \quad \{U = (\!|I \oplus_G;\ V \triangledown W|\!);\ \text{lemma (4.20)}\} \\
 & V \circ \hookleftarrow \;=\; \hookleftarrow \circ \hookrightarrow \circ F.U \\
\equiv & \quad \{V = (\!|I \oplus_L;\ \hookrightarrow \triangledown X|\!);\ \text{lemma (4.20)}\} \\
 & X \circ L.V \;=\; \hookleftarrow \circ \hookrightarrow \circ F.U \\
\Leftarrow & \quad \{\bullet\ X = \hookleftarrow \circ \hookrightarrow \circ Y\} \\
 & Y \circ L.V \;=\; F.U \\
\Leftarrow & \quad \{\bullet\ Y = F.Z \;\wedge\; L = F \bullet M\} \\
 & Z \circ M.V \;=\; U \\
\equiv & \quad \{\text{def } U\} \\
 & Z \circ M.V \;=\; (\!|I \oplus_G;\ V \triangledown W|\!) \\
\equiv & \quad \{\text{map fusion: (4.19)} \\
 & \quad \;\bullet\ M = G\star \;\wedge\; Z = (\!|I \oplus_G;\ I \triangledown W|\!)\} \\
 & \mathit{true} \quad .
\end{array}
$$

We do not deal with the second requirement of theorem (4.18): it follows from one of the properties one has to check to establish that our result is a natural simulation. This includes showing that $U> \,=\, H.I$, which implies $U> \;\sqsubseteq\; H.I$ (and can be found in [Backh0]).
Taking all the assumptions we made in the derivation together, we have the following corollary.

Corollary 4.21 ($R\cup$) $\quad U = (\!|I\oplus_{(F+G)};\ (\hookrightarrow \circ \hookrightarrow) \triangledown ((\hookrightarrow \circ \hookleftarrow) \triangledown \hookleftarrow)|\!)\cup \quad ,$

where

$$U = (\!|I\oplus_G;\ V \triangledown (\hookleftarrow \circ \hookleftarrow)|\!) \quad ,$$

$$V = (\!|I\oplus_{F\bullet G\star};\ \hookrightarrow \triangledown (\hookleftarrow \circ \hookrightarrow \circ F.Z)|\!) \quad ,$$

$$Z = (\!|I\oplus_G;\ I \triangledown (\hookleftarrow \circ \hookleftarrow)|\!) \quad .$$

4.5 Summary

In this chapter we introduced the notion of program inversion into our programming calculus. We gave a definition of 'an inverse of a program' and showed how to calculate an implementation of such an inverse. In general there are two methods for this construction: the compositional and the non-compositional.

The compositional method uses properties of the wok operator like (2.5) to express the wok of an expression in terms of (the reverses of) its subexpressions. On the other hand, the non-compositional method tries to construct a reverse of the complete expression and allows easy introduction of additional requirements.

For both methods we found that the use of domain restrictions is of great importance. If we do not recognise such restrictions we end up constructing complicated reverses that have to be simplified later on. Often these useful domain restrictions are implicitly present in the definition of a morphism. In such cases the domain trading rules have to be used to make them more explicit.

A disadvantage of the non-compositional method is the additional domain requirement. In theorem (4.18), for example, we have the conjunct $\delta> \sqsubseteq \gamma<$, where γ is a given spec and δ is the unknown we want to construct. The difficulty of proving such an inclusion is caused by the occurrence of the left-domain of some morphism. The spec calculus does not yet offer powerful enough rules to allow easy calculation of such domains. For non-trivial cases only the use of reductivity properties can help, but the theory on reductivity has not yet been developed far enough. As a consequence we are in a situation in which one can often 'see' (or rather prove by means other than the spec calculus) domain properties concerning the left-domain of a morphism that are actually very difficult, if not impossible, to prove.

Despite this problem the non-compositional method has its advantages and we use it in the next chapter to construct an algorithm that generates trees.

Chapter 5

Trees, traversals and precedence parsing

5.1 Introduction

Constructing a labeled binary tree from its inorder traversal alone or from both its inorder and preorder traversal is a topic that turns up every now and then in different publications. It is probably also the most used example to demonstrate the technique of program inversion (cf. [Chen], [Gries1], [Schoen], [Sneps]). It has some properties that make it very suitable for inversion: it is not a trivial problem, but it is easy to construct algorithms for the calculation of inorder and preorder traversals of trees.

We can also look at what happens if we restrict our attention to heaps, i.e. trees that satisfy a certain property. Can we derive an algorithm that constructs a heap from its inorder (and preorder) traversal? Such an algorithm would correspond directly to an algorithm for *precedence parsing*: a parsing method for a restricted type of grammars (cf. [Swier], [Floyd], [Aho]).

In this chapter we explore these problems in the spec calculus. In section 5.2 we introduce the labeled binary trees and define the inorder traversal, *it*, of such a tree. In section 5.3 we construct the reverse of *it* using the compositional method of chapter 4. Sections 5.4, 5.5 and 5.6 show that an inverse of *it* can be constructed in a non-compositional way that allows one to impose additional restrictions on the inverse being constructed. In section 5.7 we look at the restriction of the algorithm to heaps and in section 5.8 we investigate the connection of the resulting algorithm to precedence parsing.

5.2 Labeled binary trees

Labeled binary trees are used very often in programs because they allow efficient storage and retrieval of information. They are also the 'natural' choice for the implementation of data structures like parse trees.

In this chapter we use the following definition of the relator H to simplify expressions:

Definition 5.1 $\quad H.X \;=\; (X\times I)\times X$.

Instead of writing $(X\times I)\times X$ we occasionally drop the brackets when there is no confusion possible: we just write $X\times I\times X$. Labeled binary trees are defined using the relator $\mathcal{T}$:

Definition 5.2 ($\mathcal{T}$) $\quad \mathcal{T}.X \;=\; 1\!\!1 \;+\; H.X$.

Since $\mathcal{T}$ is a sum relator we can express the constructors of $\mu\mathcal{T}$:

$$
\begin{array}{ll}
 & \mu\mathcal{T} \\
= & \qquad \{\mu\mathcal{T} \text{ is a fixed point of } \mathcal{T}\} \\
 & \mathcal{T}.\mu\mathcal{T} \\
= & \qquad \{\text{def. } \mathcal{T}\} \\
 & 1\!\!1 \;+\; H.\mu\mathcal{T} \\
= & \qquad \{\text{def. } +\} \\
 & (\hookrightarrow \circ 1\!\!1) \;\triangledown\; (\hookleftarrow \circ H.\mu\mathcal{T}) \;\; .
\end{array}
$$

We introduce the names *empty* and *tak* for the constructors of $\mu\mathcal{T}$ ('tak' being the Dutch translation of 'branch').

Definition 5.3 $\quad empty = \hookrightarrow \circ 1\!\!1$.

Definition 5.4 $\quad tak \;=\; \hookleftarrow \;\circ\; H.\mu\mathcal{T}$.

One well-known function defined on trees is the *inorder traversal*. To define this function we denote the empty tree by $\emptyset$ and the non-empty tree constructed from binary trees l and r and label d by (l, d, r). The inorder traversal, $it.t$, then is the empty list if $t = \emptyset$ and $(it.l)d(it.r)$ if $t = (l, d, r)$ (cf. [Chen]). Or, more formally expressed in the spec calculus:

Definition 5.5 (it) $\quad it = (\!|\mathcal{T};\; nil \triangledown (cat \;\circ\; snoc{\times}I)|\!)$.

5.3 First attempt

In this section we calculate the reverse of it using the compositional approach of chapter 4. Without further comments we start with the calculation:

$$
\begin{array}{ll}
 & (\!|nil \triangledown (cat \;\circ\; snoc{\times}I)|\!)^{\cup} \\
= & \qquad \{(\!|R|\!)^{\cup} \;=\; [\![R^{\cup}]\!]\} \\
 & [\![(nil \triangledown (cat \;\circ\; snoc{\times}I))^{\cup}]\!] \\
= & \qquad \{(R \triangledown S)^{\cup} \;\;=\;\; \hookrightarrow \circ R^{\cup} \;\;\sqcup\;\; \hookleftarrow \circ S^{\cup}\} \\
 & [\![\hookrightarrow \circ\; nil^{\cup} \;\;\sqcup\;\; \hookleftarrow \circ\; (cat \;\circ\; snoc{\times}I)^{\cup}]\!] \\
= & \qquad \{(2.5)\} \\
 & [\![\hookrightarrow \circ\; nil^{\cup} \;\;\sqcup\;\; \hookleftarrow \circ\; (snoc{\times}I)^{\cup} \circ\; cat^{\cup}]\!] \\
= & \qquad \{nil^{\cup} \;\;=\;\; 1\!\!1 \circ \hookrightarrow^{\cup};\; \times \text{ is a relator; } I^{\cup} = I\} \\
 & [\![\hookrightarrow \circ\; 1\!\!1 \circ \hookrightarrow^{\cup} \;\;\sqcup\;\; \hookleftarrow \;\circ\; snoc^{\cup} \times I \;\;\circ\;\; cat^{\cup}]\!] \\
= & \qquad \{\text{def. } empty\} \\
 & [\![empty \circ \hookrightarrow^{\cup} \;\;\sqcup\;\; \hookleftarrow \;\circ\; snoc^{\cup} \times I \;\;\circ\;\; cat^{\cup}]\!] \;\; .
\end{array}
$$

We want to make two remarks. First, the result could have been improved if we had used that $D \sqsupseteq (cat \circ snoc \times I)_<$, where D is the set that contains all non-empty snoclists. This would have allowed us to rewrite the last line as:

$$[\![empty \lhd B \rhd (\hookleftarrow \circ snoc^{\cup} \times I \circ cat^{\cup})]\!] \quad ,$$

where B is the set containing only the empty snoclist.

Secondly, even with the improvement mentioned above the result is not a very pleasing one. Operationally seen, a non-empty list is transformed into a tree by splitting it in three parts: a list, an element and another list. On the sub-lists recursion is applied. The resulting trees, together with the element are used to construct a tree.

However, we would like an algorithm that transforms a list into a tree element by element, i.e. reading the list once from left to right.

We would also prefer that the construction of a tree is the non-deterministic part of the algorithm, instead of the 'destruction' of the list. In that way, it will be much easier to change the algorithm into one that creates only heaps.

For these reasons we do not bother to construct $cat^{\cup}$, but start again from scratch.

5.4 Second attempt

In this section we construct an inverse of it, called ti, using a non-compositional method comparable to the one we saw in theorem (4.18) at the end of chapter 4. We hope that the freedom that this method offers can be used to construct a program that is of more use than the one of the previous section.

We are looking for a program ti that constructs a tree from a list, t, such that the inorder traversal of this tree is equal to the list t. We can translate this specification into the following two requirements:

(5.6) $\quad it \circ ti = \mu\mathcal{S} \quad$ and

(5.7) $\quad ti_< \sqsubseteq \mu\mathcal{T} \ \wedge \ ti_> \sqsubseteq \mu\mathcal{S} \quad .$

We start our calculation with requirement (5.6):

$$\begin{array}{ll}
 & it \circ ti = \mu\mathcal{S} \\
\equiv & \qquad \{\bullet\ ti = (\!|\mathcal{S};\ X \triangledown Y|\!)\ ;\ \mu\mathcal{S} = (\!|\mathcal{S};\ \mu\mathcal{S}|\!)\} \\
 & it \circ (\!|\mathcal{S};\ X \triangledown Y|\!) = (\!|\mathcal{S};\ nil \triangledown snoc|\!) \\
\Leftarrow & \qquad \{\text{fusion}\} \\
 & it \circ X \triangledown Y \ = \ nil \triangledown snoc \ \circ \ \mathcal{S}.it \\
\equiv & \qquad \{\circ \text{ distr. over } \triangledown\text{; def. } \mathcal{S}\} \\
 & (it \circ X) \triangledown (it \circ Y) \ = \ nil \triangledown snoc \ \circ \ \mathbb{1} + (it \times I) \\
\equiv & \qquad \{\triangledown + \text{fusion; } \triangledown \text{ cancellation}\} \\
 & it \circ X = nil \ \ \wedge \ \ it \circ Y \ = \ snoc \circ it \times I
\end{array}$$

$$\equiv \qquad \{\bullet\ X = empty\}$$

$$it \circ Y \;=\; snoc \circ it{\times}I \quad .$$

In the first step we made our first decision: we choose *ti* to be a catamorphism on snoclists. This decision is made because we want the list to be transformed element-by-element into a tree. The resulting equation can be obtained using the fusion law for catamorphisms and the next steps are straightforward simplifications.

Now we take our next decision: we decide that Y will be a paramorphism, called *ins*. This decision is taken with an operational argument in mind. The program *ins* constructs a new tree from an old tree by inserting a new value, v. To be as general as possible, we want to be able to use the contents of the old tree as well as the structure to guide this process. Surely, if we want a program that constructs only heaps, this capability will be needed.

However, there is a problem with this choice. This problem is similar to the problem of finding a catamorphism for the append operation on conslists or for the addition on natural numbers (cf. [Doorn]). In general these problems are not easy to solve. It is possible to define *parameterised* catamorphisms (or paramorphisms), but this requires a great deal of theory and obscures the main derivation. We therefore choose an alternative: we define the section $(ins\ v)$ as a paramorphism.

Using sections, our specification for *ins* becomes:

$$it \circ (ins\ v) = (snoc\ v) \circ it \quad . \tag{5.8}$$

The lefthandside of equation (5.8) has the right shape to apply the paramorphism fusion law, but in order to do so the righthandside has to be expressed as a paramorphism. We first look at the more general problem of how we can show that an expression can be formulated as a paramorphism. Then we apply the lemma we find to $(snoc\ v) \circ it$ and construct a paramorphism for this expression. In the next section we apply the fusion law for paramorphisms to arrive at a definition for $(ins\ v)$.

Lemma 5.9 If R is a spec such that $R = R \circ \mu\mathcal{T}$ and X and Y are arbitrary specs then

$$R = [\![X \triangledown Y]\!] \quad \equiv \quad R \circ empty = X \circ \mathbb{1} \quad \wedge \quad R \circ tak \;=\; Y \circ H.(R \vartriangle I) \quad .$$

Proof

$$\begin{aligned}
& R = [\![X \triangledown Y]\!] \\
\equiv \quad & \qquad \{R \;=\; R \circ \mu\mathcal{T}\ ;\ \text{UEP}\} \\
& R \circ \mu\mathcal{T} \;=\; X \triangledown Y \;\circ\; \mathcal{T}.(R \vartriangle I) \;\circ\; \mu\mathcal{T} \\
\equiv \quad & \qquad \{\text{def. } \mu\mathcal{T}\ ;\ \triangledown\text{-cancellation}\} \\
& \quad R \circ empty \;=\; X \triangledown Y \;\circ\; \mathcal{T}.(R \vartriangle I) \;\circ\; empty \\
& \wedge\ R \circ tak \;=\; X \triangledown Y \;\circ\; \mathcal{T}.(R \vartriangle I) \;\circ\; tak \\
\equiv \quad & \qquad \{\text{def. } empty,\ tak,\ \mathcal{T};\ \text{comp.rules } \triangledown \text{ and } +\} \\
& \quad R \circ empty = X \circ \mathbb{1} \\
& \wedge\ R \circ tak \;=\; Y \circ H.(R \vartriangle I) \quad .
\end{aligned}$$

$\square$

To apply lemma (5.9) we first establish that $((snoc\ v) \circ it)> = \mu\mathcal{T}$.

$$
\begin{array}{ll}
 & ((snoc\ v) \circ it)> \\
= & \quad \{\text{domains}\} \\
 & ((snoc\ v)> \circ it)> \\
= & \quad \{(snoc\ v)> = \mu\mathcal{S}\} \\
 & (\mu\mathcal{S} \circ it)> \\
= & \quad \{\mu\mathcal{S} \sqsupseteq it<\} \\
 & it> \\
= & \quad \{it> = \mu\mathcal{T}\} \\
 & \mu\mathcal{T} \quad .
\end{array}
$$

Having established the assumption of lemma (5.9) we now instantiate R with $(snoc\ v) \circ it$. We find:

$$
\begin{array}{ll}
 & (snoc\ v) \circ it = [\![X \triangledown Y]\!] \\
\equiv & \quad \{\text{lemma (5.9)}\} \\
 & \quad (snoc\ v) \circ it \circ empty = X \circ 1\!\!1 \\
 & \wedge\ (snoc\ v) \circ it \circ tak \ = \ Y \circ H.(((snoc\ v) \circ it) \vartriangle I) \quad .
\end{array}
$$

We examine the first conjunct of the last line of the derivation above:

$$
\begin{array}{ll}
 & (snoc\ v) \circ it \circ empty \\
= & \quad \{\text{def. } it\} \\
 & (snoc\ v) \circ nil \quad .
\end{array}
$$

We introduce the notation $[v]$ for $(snoc\ v) \circ nil$; it symbolises the constructor of the singleton list containing the element v. Since $nil \circ 1\!\!1 = nil$ we have $[v] \circ 1\!\!1 = [v]$ and our result for the first conjunct is:

(5.10) $\quad (snoc\ v) \circ it \circ empty = [v] \circ 1\!\!1 \quad .$

We continue with the second conjunct that we have to establish: $(snoc\ v) \circ it \circ tak \ = \ Y \circ H.(((snoc\ v) \circ it) \vartriangle I)$. We start with the lefthandside of this equation and try to arrive at an expression that has the shape of the righthandside, thus finding a definition of Y. Our general strategy is to 'push' the relation $(snoc\ v)$ to the right.

$$
\begin{array}{ll}
 & (snoc\ v) \circ it \circ tak \\
= & \quad \{\text{def. } it\} \\
 & (snoc\ v) \ \circ \ cat \ \circ \ snoc{\times}I \ \circ \ H.it
\end{array}
$$

$$
\begin{array}{ll}
= & \{\text{property of } cat\text{: } (2.58)\} \\
& cat \;\circ\; I{\times}(snoc\ v) \;\circ\; snoc{\times}I \;\circ\; H.it \\
= & \{\times\times\text{-fusion; def. } H\} \\
& cat \;\circ\; snoc{\times}I \;\circ\; (I{\times}I){\times}(snoc\ v) \;\circ\; (it{\times}I){\times}it \\
= & \{\times\times\text{-fusion; } it{>} \sqsubseteq \mu\mathcal{T}\} \\
& cat \;\circ\; snoc{\times}I \;\circ\; it{\times}I{\times}I \;\circ\; (\mu\mathcal{T}{\times}I){\times}((snoc\ v) \circ it) \\
= & \{\gg {\times}I{\times} \ll \;\circ\; H.(X \vartriangle I) \;=\; X{>} \times I \times X\} \\
& cat \;\circ\; snoc{\times}I \;\circ\; it{\times}I{\times}I \;\circ\; \gg {\times}I{\times} \ll \;\circ\; H.(((snoc\ v) \circ it) \vartriangle I) \;\;.
\end{array}
$$

After four steps we have achieved our goal: the expression $(snoc\ v)$ has been pushed completely to the right. In the last step we had to find a way to 'massage' the expression into the right shape: an expression ending with $H.(((snoc\ v) \circ it) \vartriangle I)$. This was done by introducing the spec $\gg {\times}I{\times} \ll$. But are there other possibilities? Take for example the spec $\gg {\times}I{\times} \gg$. We calculate:

$$
\begin{array}{ll}
& \gg {\times}I{\times} \gg \;\circ\; H.(((snoc\ v) \circ it) \vartriangle I) \\
= & \{\gg {\times}I{\times} \gg \;\circ\; H.(X \vartriangle I) \;=\; X{>} \times I \times X{>}\} \\
& \mu\mathcal{T}{\times}I{\times}\mu\mathcal{T} \\
= & \{\text{def. } H\} \\
& H.\mu\mathcal{T} \\
= & \{\text{def. } tak\} \\
& tak{>} \;\;.
\end{array}
$$

So we find another solution:

$$
\begin{array}{ll}
& (snoc\ v) \circ it \circ tak \\
= & \{\text{domains}\} \\
& (snoc\ v) \circ it \circ tak \circ tak{>} \\
= & \{\text{calculation above}\} \\
& (snoc\ v) \;\circ\; it \;\circ\; tak \;\circ\; \gg {\times}I{\times} \gg \;\circ\; H.(((snoc\ v) \circ it) \vartriangle I) \;\;.
\end{array}
$$

We can combine the two solutions we found to get a more general one. To do so we use the idempotency of the cup: for arbitrary specs R, S, T and U such that $R = S \circ U$ and $R = T \circ U$ we have:

$$
\begin{array}{ll}
& R \\
= & \{\text{idempotency of } \sqcup\} \\
& R{\sqcup}R
\end{array}
$$

$$
\begin{array}{ll}
= & \{\text{assumptions}\} \\
 & (S \circ U) \sqcup (T \circ U) \\
= & \{\text{distribution} \circ \text{ over } \sqcup\} \\
 & (S \sqcup T) \circ U \quad .
\end{array}
$$

Remember that we were constructing a spec Y such that

$$(snoc\ v) \circ it \circ tak \;=\; Y \circ H.(((snoc\ v) \circ it) \vartriangle I) \;\;;$$

from the calculations above we conclude that the following definition of Y satisfies this requirement:

(5.11) $\quad Y = V \sqcup W \quad$, where

$$V \;=\; (snoc\ v) \circ it \circ tak \circ \gg \times I \times \gg \;\;, \text{and}$$

$$W \;=\; cat \circ snoc \times I \circ it \times I \times I \circ \gg \times I \times \ll \;\;.$$

Having dealt with both conjuncts of the righthandside of the equation of lemma (5.9) (i.e. (5.10) and (5.11)) we conclude that:

(5.12) $\quad (snoc\ v) \circ it = [\![[v] \triangledown (V \sqcup W)]\!] \quad .$

5.5 About halfway there

Remember that we were trying to construct a spec *ins* such that

$$it \circ (ins\ v) = (snoc\ v) \circ it$$

and that we have found a paramorphism for the expression $(snoc\ v) \circ it$. We proceed by applying the fusion law to find a paramorphism for the section $(ins\ v)$:

$$
\begin{array}{ll}
 & it \circ (ins\ v) = (snoc\ v) \circ it \\
\equiv & \{\bullet\ (ins\ v) = [\![X]\!];\ (5.12)\} \\
 & it \circ [\![X]\!] = [\![[v] \triangledown (V \sqcup W)]\!] \\
\Leftarrow & \{\text{fusion law}\} \\
 & it \circ X \;=\; [v] \triangledown (V \sqcup W) \;\circ\; \mathcal{T}.(it \times I) \\
\equiv & \{\text{def. } \mathcal{T};\ \triangledown + \text{ fusion}\} \\
 & it \circ X \;=\; [v] \;\triangledown\; ((V \sqcup W) \circ H.(it \times I)) \\
\equiv & \{\bullet\ X = Y \triangledown Z;\ \circ \text{ distrib. over } \triangledown\,;\ \triangledown\text{-cancellation}\} \\
 & it \circ Y = [v] \quad \wedge \quad it \circ Z \;=\; (V \sqcup W) \circ H.(it \times I) \;\;.
\end{array}
$$

If we introduce the section $(leaf\ v)$ such that

$$x \ (\mathit{leaf} \ v) \ y \quad \equiv \quad x = (\emptyset, v, \emptyset) \ \wedge \ y = 1\!\!1 \ ,$$

then we have for $[v]$:

(5.13) $\quad [v] = \mathit{it} \circ (\mathit{leaf} \ v)$

and hence we can take $(\mathit{leaf} \ v)$ as a definition for Y.

Now we only have to construct a spec Z such that

$$\mathit{it} \circ Z \quad = \quad (V \sqcup W) \circ H.(\mathit{it} \times I) \ .$$

Consider the expression $V \circ H.(\mathit{it} \times I)$. We calculate and try to arrive at an expression of the shape $\mathit{it} \circ R$ for some spec R. We first concentrate on a small part of V:

$$\begin{array}{ll} & \gg \times I \times \gg \ \circ H.(\mathit{it} \times I) \\ = & \qquad \{ \gg \ \circ \mathit{it} \times I \quad = \quad \gg \ \circ \mu\mathcal{T} \times I \} \\ & \gg \times I \times \gg \ \circ H.(\mu\mathcal{T} \times I) \ . \end{array}$$

We denote the monotype $H.(\mu\mathcal{T} \times I)$ by A and only use that $A \sqsupseteq H.(\mu\mathcal{T} \times I)$.
Furthermore, we have

$$\begin{array}{ll} & \mathit{snoc} \circ \mathit{it} \times I \\ = & \qquad \{ \text{property of } \mathit{cat}\text{: (2.59)} \} \\ & \mathit{cat} \ \circ \ (\mathit{snoc} \circ \mathit{it} \times I) \vartriangle (\mathit{nil} \circ \top\!\!\top) \\ = & \qquad \{ \mathit{nil} = \mathit{it} \circ \mathit{empty} \ ; \times \vartriangle \text{-fusion} \} \\ & \mathit{cat} \ \circ \ \mathit{snoc} \times I \ \circ \ \mathit{it} \times I \times \mathit{it} \ \circ \ (I \times I) \vartriangle (\mathit{empty} \circ \top\!\!\top) \\ = & \qquad \{ \text{def. } \mathit{it} \} \\ & \mathit{it} \circ \mathit{tak} \circ (I \times I) \vartriangle (\mathit{empty} \circ \top\!\!\top) \ . \end{array}$$

When we translate the result of the calculation above into an expression using sections, we find:

(5.14) $\quad (\mathit{snoc} \ v) \circ \mathit{it} = \mathit{it} \circ (\mathit{topl} \ v) \qquad ,$

where $(\mathit{topl} \ v)$ is defined by $(\mathit{topl} \ v).x = (x, v, \emptyset)$.

With this definition we conclude that:

(5.15) $\quad V \circ H.(\mathit{it} \times I) \ = \ \mathit{it} \circ P \circ A \qquad ,$

where $P \ = \ (\mathit{topl} \ v) \circ \mathit{tak} \circ \ \gg \times I \times \gg$.

In the same way as above we calculate with the expression $W \circ H.(\mathit{it} \times I)$. Again we first look at only a small part of this expression:

$$\gg \times I \times \ll \;\circ\; H.(it \times I)$$

$$= \quad \{\gg \circ\; it \times I \;=\; \gg \;\circ\; \mu\mathcal{T} \times I\}$$

$$(\gg \circ\; \mu\mathcal{T} \times I) \times I \times (it \circ \ll)$$

$$= \quad \{\times\times\text{-fusion}\}$$

$$I \times I \times it \;\circ\; \gg \times I \times \ll \;\circ\; (\mu\mathcal{T} \times I) \times I \times (I \times I) \quad .$$

Again we replace the domain restriction $(\mu\mathcal{T} \times I) \times I \times (I \times I)$ by a monotype, B, and only use that $B \sqsupseteq H.(\mu\mathcal{T} \times I)$. Next we look at the complete expression:

$$W \circ H.(it \times I)$$

$$= \quad \{\text{def. } W\text{; above}\}$$

$$cat \;\circ\; snoc \times I \;\circ\; it \times I \times I \;\circ\; I \times I \times it \;\circ\; \gg \times I \times \ll \;\circ\; B$$

$$= \quad \{\times\times\text{fusion; def. } it\}$$

$$it \circ tak \circ \gg \times I \times \ll \circ B \quad .$$

For brevity we introduce the spec Q to formulate our result:

(5.16) $\quad W \circ H.(it \times I) \;=\; it \circ Q \circ B \quad .$

where $Q \;=\; tak \circ \gg \times I \times \ll$.

Remember that we have to construct a spec Z such that

$$it \circ Z \;=\; (V \sqcup W) \circ H.(it \times I) \quad .$$

From (5.15) and (5.16) we find that we can define $Z \;=\; P \circ A \;\sqcup\; Q \circ B$. This means that we have finally achieved our main goal: finding a definition of the section $(ins\ v)$:

(5.17) $\quad (ins\ v) = [\![(leaf\ v) \;\triangledown\; (P \circ A \sqcup Q \circ B)]\!] \quad ,$

where

$$P \;=\; (topl\ v) \circ tak \circ \gg \times I \times \gg \quad \text{and}$$

$$Q \;=\; tak \circ \gg \times I \times \ll \quad .$$

5.6 Removing the domain restrictions

We can remove the domain restrictions A and B in definition (5.17) by use of the domain trading rules for morphisms. In fact we have the following lemma:

Lemma 5.18 $\quad [\![\mathcal{T}; R]\!] = [\![\mathcal{T}; R \;\circ\; \mathcal{T}.(\mu\mathcal{T} \times I)]\!] \quad \Leftarrow \quad \mu\mathcal{T} \sqsupseteq R{<}$

Proof

$$\begin{array}{ll}
 & [\![R]\!] = [\![R \;\circ\; \mathcal{T}.(\mu\mathcal{T}\times I)]\!] \\
\Leftarrow & \qquad \{\text{domain trading: (3.10)}\} \\
 & \mathcal{T}.(\mu\mathcal{T}\times I) \;\sqsupseteq\; \mathcal{T}.(R{<}\times I) \\
\Leftarrow & \qquad \{\text{monotonicity of relators}\} \\
 & \mu\mathcal{T} \sqsupseteq R{<} \quad .
\end{array}$$

□

If we take a closer look at the expression $(\mathit{leaf}\ v) \triangledown (P{\circ}A \sqcup Q{\circ}B)$ we see that it is in fact of the shape $\mathit{tak} \circ R$ for some spec R (use that both $(\mathit{leaf}\ v)$ and $(\mathit{topl}\ v)$ are sections of tak and that $\circ$ distributes over $\triangledown$ and $\sqcup$). For such expressions we have that $(\mathit{tak} \circ R){<} \;\sqsubseteq\; \mathit{tak}{<} = \mu\mathcal{T}$. It follows that

$$((\mathit{leaf}\ v) \triangledown (P{\circ}A \sqcup Q{\circ}B)){<} \;\sqsubseteq\; \mu\mathcal{T} \quad \text{, as well as}$$

$$((\mathit{leaf}\ v) \triangledown (P \sqcup Q)){<} \;\sqsubseteq\; \mu\mathcal{T} \quad .$$

These conclusions allow us to use lemma (5.18) to remove the domain restrictions as promised:

$$\begin{array}{ll}
 & [\![(\mathit{leaf}\ v) \triangledown (P{\circ}A \sqcup Q{\circ}B)]\!] \\
= & \qquad \{(5.18)\} \\
 & [\![(\mathit{leaf}\ v) \triangledown (P{\circ}A \sqcup Q{\circ}B) \;\circ\; \mathcal{T}.(\mu\mathcal{T}\times I)]\!] \\
= & \qquad \{\text{def. } \mathcal{T};\ A \sqsupseteq H.(\mu\mathcal{T}\times I)\ ;\ B \sqsupseteq H.(\mu\mathcal{T}\times I);\ \text{monotypes}\} \\
 & [\![(\mathit{leaf}\ v) \triangledown (P \sqcup Q) \;\circ\; \mathcal{T}.(\mu\mathcal{T}\times I)]\!] \\
= & \qquad \{(5.18)\} \\
 & [\![(\mathit{leaf}\ v) \triangledown (P \sqcup Q)]\!] \quad .
\end{array}$$

We arrive at the following inverse of the inorder traversal algorithm, called ti:

(5.19) $\quad \mathit{ti} = (\![\mathcal{S};\ \mathit{empty} \triangledown \mathit{ins}]\!) \quad$, where

$$(\mathit{ins}\ v) = [\![\mathcal{T}; (\mathit{leaf}\ v) \triangledown (P{\sqcup}Q)]\!] \quad ,$$

$$P \;=\; (\mathit{topl}\ v) \circ \mathit{tak} \circ \gg \times I \times \gg \quad \text{and}$$

$$Q \;=\; \mathit{tak} \circ \gg \times I \times \ll \quad .$$

With this definition it is easy to verify that (5.7) holds; we leave this as an exercise. A way to solve problems of this kind will be discussed in the next section.

5.7 Heaps

The *heaps* are a subset of the binary labeled trees that can be defined by:

(5.20) $\quad \emptyset \in heap \qquad$ and

(5.21) $\quad (l,d,r) \in heap \quad \Leftarrow \quad l \in heap \;\wedge\; r \in heap \;\wedge\; root.l \leq d \;\wedge\; root.r \leq d \quad ,$

where $root.\emptyset \;=\; 0$ and $root.(l,d,r) \;=\; d$.

We have already seen that sets are represented in the spec calculus by monotypes; here is the representation of heap:

Definition 5.22 (heap) $\quad heap \;=\; (\!|\mathcal{T};\; empty \triangledown (tak \circ E)|\!)$, where E is defined by

$$(l,d,r) \in E \quad\equiv\quad root.l \leq d \;\wedge\; root.r \leq d \;\;.$$

□

We want to change the program ti that we constructed, in such a way that only heaps are generated, i.e. $heap \;\sqsupseteq\; ti<$. We want to do so by restricting the non-determinism, but it is our aim not to change the right-domain of ti.

The restriction to heaps would be no problem if our program had the shape $(\!|heap \circ X|\!)$. In such a case we would have:

$$
\begin{array}{ll}
 & (\!|heap \circ X|\!)< \\
= & \qquad \{(\!|F;\; R|\!) \;=\; R \circ F.(\!|F;\; R|\!)\} \\
 & (heap \circ X \circ \mathcal{S}.(\!|heap \circ X|\!))< \\
\sqsubseteq & \qquad \{\text{domains}\} \\
 & heap< \\
= & \qquad \{A \;=\; A<\} \\
 & heap \quad .
\end{array}
$$

We can use the domain trading rule to rewrite ti in such a shape:

$$
\begin{array}{ll}
 & (\!|\mathcal{S};\; empty \triangledown ins|\!) \;=\; (\!|\mathcal{S};\; heap \;\circ\; empty \triangledown ins|\!) \\
\Leftarrow & \qquad \{\text{domain trading rule}\} \\
 & heap \;\sqsupseteq\; (empty \triangledown ins \;\circ\; \mathcal{S}.heap)< \quad .
\end{array}
$$

To achieve our goal, we only have to show $heap \;\sqsupseteq\; (empty \triangledown ins \;\circ\; \mathcal{S}.heap)<$. First, we try to simplify the righthandside of the inequality:

$$
\begin{array}{ll}
& (\mathit{empty} \triangledown \mathit{ins} \;\circ\; \mathcal{S}.\mathit{heap})< \\
= & \quad \{\text{def. } \mathcal{S};\ \triangledown + \text{fusion}\} \\
& (\mathit{empty} \triangledown (\mathit{ins} \circ \mathit{heap}\times I))< \\
= & \quad \{\text{def. } \triangledown\} \\
& (\mathit{empty} \circ \hookrightarrow\cup \;\sqcup\; \mathit{ins} \circ \mathit{heap}\times I \circ \hookleftarrow\cup)< \\
= & \quad \{< \text{ distributes over } \sqcup\} \\
& (\mathit{empty} \circ \hookrightarrow\cup)< \;\sqcup\; (\mathit{ins} \circ \mathit{heap}\times I \circ \hookleftarrow\cup)< \\
= & \quad \{\text{domains; } \hookrightarrow\cup < \;=\; \hookleftarrow\cup < \;=\; I\} \\
& \mathit{empty}< \;\sqcup\; (\mathit{ins} \circ \mathit{heap}\times I)< \quad .
\end{array}
$$

We return to our goal:

$$
\begin{array}{ll}
& \mathit{heap} \;\sqsupseteq\; (\mathit{empty} \triangledown \mathit{ins} \;\circ\; \mathcal{S}.\mathit{heap})< \\
\equiv & \quad \{\text{above}\} \\
& \mathit{heap} \;\sqsupseteq\; \mathit{empty}< \;\sqcup\; (\mathit{ins} \circ \mathit{heap}\times I)< \\
\equiv & \quad \{(2.1)\} \\
& \mathit{heap} \;\sqsupseteq\; \mathit{empty}< \;\wedge\; \mathit{heap} \;\sqsupseteq\; (\mathit{ins} \circ \mathit{heap}\times I)< \\
\equiv & \quad \{\text{def. } \mathit{heap}\} \\
& \mathit{heap} \;\sqsupseteq\; (\mathit{ins} \circ \mathit{heap}\times I)< \quad .
\end{array}
$$

The only proof obligation left is $\mathit{heap} \;\sqsupseteq\; (\mathit{ins} \circ \mathit{heap}\times I)<$. To let this correspond with the definition of the program we have to write this using the section $(\mathit{ins}\ v)$:

(5.23) $\qquad \mathit{heap} \;\sqsupseteq\; ((\mathit{ins}\ v) \circ \mathit{heap})< \qquad .$

This is our only proof obligation left. However, if we try to solve this inequality (which can de done by restricting the non-determinism of $(\mathit{ins}\ v)$), we encounter the same problems as in chapter 4. Again it is the occurrence of the left-domain of a morphism that causes these problems. It is not yet possible to give an easy to follow proof of this property within the limits of the spec calculus. Were we to allow the use of point-wise reasoning and induction then a short proof can be given. In fact we change the definition of $(\mathit{ins}\ v)$ into:

(5.24) $\qquad (\mathit{ins}\ v) = [\![\mathcal{T}; (\mathit{leaf}\ v) \triangledown (P\sqcup Q)]\!] \qquad ,$

$$P \;=\; (\mathit{topl}\ v) \circ \mathit{tak} \circ C \circ \;\gg \times I \times \gg \quad \text{and}$$

$$Q \;=\; \mathit{tak} \circ D \circ \;\gg \times I \times \ll \;\; ;$$

the monotypes C and D are introduced to restrict the non-determinism. The construction of a proof of the property $\mathit{heap} \;\sqsupseteq\; ((\mathit{ins}\ v) \circ \mathit{heap})<$ gives us definitions of C and D . Such a construction can be found in the appendix. As a consequence we have that with the following definitions of C and D the spec $(\mathit{ins}\ v)$ defined by (5.24) satisfies (5.23):

(5.25) $\qquad (l,d,r) \in C \;\equiv\; d \leq v \qquad$ and

$$(5.26) \qquad (l,d,r) \in D \;\equiv\; d \geq v \quad .$$

It is easy to see that every non-empty tree is an element of C or D. As a consequence we have that the right-domain of ti has not changed.

5.8 Precedence parsing

In [Swier] the claim is made that an algorithm like the one we constructed above, corresponds directly to a precedence parsing algorithm. To investigate this claim we take a look at the theory of precedence parsing.

A precedence parsing algorithm is one of the simplest instances of a *shift-reduce* parsing algorithm (cf. [Aho]) and its origins can be found in [Floyd]. Precedence-oriented parsing techniques were among the first techniques to be used in the construction of parsers for programming languages and a number of variants of precedence parsing have appeared in the literature. The grammars that can be parsed by this method are called *precedence* grammars. One can distinguish simple, weak, extended, mixed strategy and operator precedence grammars.

In general precedence grammars have the property that by consulting certain (precedence) relations that hold among the symbols appearing in the input we can decide whether a shift or a reduce action has to be applied. In a shift-action the current input symbol is pushed on a stack; in a reduce-action a portion of the stack is replaced by a nonterminal according to a production rule of the grammar.

In [Floyd] three precedence relations are defined for a precedence grammar. An *operator* precedence grammar is defined as a precedence grammar for which no more than one of these three relations holds between any ordered pair of terminal symbols.
Let $G = (V_N, V_T, S, Pr)$ be context-free grammar with the usual definition of the relation $\rightarrow$:

$$x \rightarrow y \equiv \exists(a,b,l,r :: \; x = lar \;\; \wedge \;\; y = lbr \;\; \wedge \;\; (a,b) \in Pr)$$

and the reflexive, transitive closure of $\rightarrow$ denoted by $\overset{*}{\rightarrow}$.
We define the following three precedence relations, some or all of which may hold between two terminal characters x and y (A and B are non-terminals, α, β and γ are strings):

- $x \doteq y$ if there is a production $A \rightarrow \alpha x y \beta$ or $A \rightarrow \alpha x B y \beta$.
- $x \gtrdot y$ if there is a production $A \rightarrow \alpha B y \beta$ and a derivation $B \overset{*}{\rightarrow} \gamma$, where x is the rightmost terminal character of γ.
- $x \lessdot y$ if there is a production $A \rightarrow \alpha x B \beta$ and a derivation $B \overset{*}{\rightarrow} \gamma$, where y is the leftmost terminal character of γ.

There are two ways to transform our heap generating algorithm into an algorithm for precedence parsing. The first method is to introduce two mappings, f and g from symbols to natural numbers, such that:

$$f.x < g.y \;\; \Leftarrow \;\; x \lessdot y \qquad ,$$

$$f.x = g.y \;\; \Leftarrow \;\; x \doteq y \qquad \text{and}$$

$$f.x > g.y \;\Leftarrow\; x \rightsquigarrow y \qquad .$$

In [Floyd] an algorithm is given to calculate such mappings f and g if they exist. Now we change the definition of C and D, i.e. (5.25) and (5.26) become:

$$(5.27) \qquad (l,d,r) \in C \;\equiv\; f.d \leq g.v \qquad \text{and}$$

$$(5.28) \qquad (l,d,r) \in D \;\equiv\; f.d > g.v \qquad .$$

Note that we have eliminated the last bit of non-determinism by replacing 'at least' by 'greater than' in (5.28). We have arrived at a program that corresponds with Floyd's program in [Floyd].

The second way to transform our program is to replace the 'at most' and 'at least' relations in the definition of C and D by the corresponding precedence relations. Of course this requires the presence of a table in which these precedence relations are defined for all combinations of the symbols of the grammar.

5.9 Summary

We tried both methods, introduced in chapter 4, to construct the inverse of an algorithm for the calculation of the inorder traversal of a tree. Using the non-compositional method we found an inverse of this algorithm. This reverse was adapted in two steps to yield a program for precedence parsing.

Chapter 6

Conclusions and suggestions for further research

In this report we have taken a brief look at different uses of a relational framework for program derivation. In particular we studied the *spec calculus*: a programming calculus designed for the construction of relational programs in a calculational style. After a brief introduction to our programming notation we started our exploration of the use of the spec calculus. We did so by studying the introduction of the technique of program inversion and by taking a first look at the derivation of some simple parsing algorithms. We encountered a lot of examples and techniques along the way, and it is these techniques and examples that allow us to judge whether the spec calculus lives up to the demands it was designed for.

Of course it is up to the readers to make such a judgement for themselves, because there will never be an accepted definition of concepts like 'a clear calculation'. But we can add the following remarks to help the reader.

- Programming techniques known from the realm of functional programming can also be used in the spec calculus. We saw illustrations of this fact throughout this report. The spec calculus, however, is also 'strong' enough to express theorems about classes of programs. As a topic for further research we suggested the expression of the tail recursion theorem for functional programs within our program notation.

- A theory of program inversion requires a relational setting. In a functional or imperative setting such a theory suffers from the difficulties one encounters trying to deal with non-determinism.

- In the construction of an inverse of an imperative program, assertions play a rôle as important as statements. Often a statement becomes an assertion and vice versa. The 'inversion' of a program boils down to first constructing the 'right' assertions and then applying the rules for inversion. In the spec calculus programs are not annotated with assertions: monotypes (and especially domain restrictions) take their place. However, as we have seen, the notion of program inversion can be defined in an elegant way.

- We have seen two different methods to construct implementations of reverses. The compositional way uses the properties of the reverse-operator to express the reverse of an expression in terms of the reverses of the subexpressions. The non-compositional method tries to find the reverse of an expression by solving an equation. This process permits more freedom in adding requirements on the reverse that is constructed, such as the requirement that it should be a catamorphism.

- For both methods of constructing reverses we found that it pays to make use of domain restrictions as much as possible. In cases where such restrictions are implicitly present, like in the definition of a morphism, it is important to state them explicitly using the trading rules presented in the morphism theorem.

- A problem that turned up twice in this report and that needs further attention is the calculation of left-domains of arbitrary morphisms. At this point it is sometimes impossible to prove (left-)domain properties within the spec calculus. More often, it is possible to give short and clear proofs of such properties by stepping outside the spec calculus and using means like induction and point-wise calculations. The theory of reductivity allows one to formulate theorems that ease calculations of left-domains of arbitrary morphisms, but this theory has not yet been developed far enough to solve these problems. This certainly is a subject for further research.

- We have only been able to take a brief look at the topic of the derivation of parsing algorithms. The recent developments in the research of functional parsing algorithms, together with our own results of which only a small part was ready for presentation in this report, support our opinion that this is an area in which much progress can be made by using a relational calculus.

- This report is the first graduating dissertation that was completely written using the Mathpad system. This system provides easy access to all the strange symbols used in this report and in other mathematical publications. Such a system is an invaluable help in the construction of both programs (or proofs) and their documentation.

Bibliography

[Aho] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation and Compiling. Vol I: Parsing.* Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Back] R.J.R. Back and J. von Wright. Statement inversion and strongest postcondition. In *Science of Computer Programming* 20 (1993) 223-251, Elsevier.

[Backh] C.J. Aarts, R.C. Backhouse, P. Hoogendijk, T.S. Voermans and J. van der Woude. A relational theory of datatypes. September 1992.

[Backh0] R.C. Backhouse. Monad Decomposition. November 26, 1992.

[Backh1] R.C. Backhouse and H. Doornbos. Induction and Recursion on Datatypes. In *Categories of Relations in Computer Science*, Oxford, July 1993.

[Bird] R. Bird and O. de Moor. Relational program derivation and context-free language recognition. 1993.

[Chen] W. Chen and J.T. Udding. Program inversion: more than fun! Rept. CS 8903, Department of Mathematics and Computer Science, University of Groningen, 1988.

[Dijkstra0] E.W. Dijkstra. Program inversion. EWD671. Eindhoven University of Technology, 1978.

[Doorn] H. Doornbos. Parameterised Relational Catamorphisms. January 20, 1992.

[Floyd] R.W. Floyd. Syntactic Analysis and Operator Precedence. In *Journal of the ACM*, vol 10, no. 3 (July 1963), pp. 316-333. Also appeared in B.W. Pollack (ed.). *Compiler Techniques*, Auerbach Publishers, 1972.

[Gries0] D. Gries. *The science of programming.* Springer-Verlag, New York, 1981.

[Gries1] D. Gries and J.L.A. van de Snepscheut. Inorder Traversal of a Binary Tree and its Inversion. In E.W. Dijkstra (ed.). *Formal Development of Programs and Proofs*, University of Texas at Austin Year of Programming Series, Addison-Wesley, Amsterdam, 1990.

[Harri] P.G. Harrison and H. Khoshnevisan. On the synthesis of function inverses. In *Acta Informatica* 29 (1992) 211-239, Springer-Verlag.

[Hoare] C.A.R. Hoare and J. He. The weakest prespecification. In *Fundamenta Informaticae*, 9:51-84, 217-252, 1986.

[Hooge] R. Hoogerwoord. *The design of functional programs: a calculational approach.* Ph.D. thesis, Eindhoven University of Technology, 1989.

[Leerm0] R. Leermakers. *The functional treatment of Parsing.* Kluwer Academic Publishers, February 1993.

[Leerm1] R. Leermakers. Recursive ascent parsing: from Earley to Marcus. In *Theoretical Computer Science*, 104 (1992) 299-312.

[Leerm2] R. Leermakers, L. Augusteijn and F.E.J. Kruseman Aretz. A functional LR parser. In *Theoretical Computer Science*, 104 (1992) 313-323.

[Moor] O. de Moor. Inverses in Program Synthesis. In *International summer school on constructive algorithmics*, part II, Hollum - Ameland, September 1989.

[Rietman] F.J. Rietman. An Aggregated Segment Sum Theorem in the Relational System. In *EURICS Workshop, Calculational Theories of Program Structure*, part I, Hollum - Ameland, September 1991.

[Schoen] B. Schoenmakers. Inorder Traversal of a Binary Heap and its Inversion in Optimal Time and Space. Eindhoven University of Technology.

[Sneps] J.L.A. van de Snepscheut. Inversion of a recursive tree traversal. Caltech-CS-TR-91-07, California Institute of Technology, 1991.

[Swier] D. Swierstra and O. de Moor. Virtual Data Stuctures.

[Tarski] A. Tarski. On the calculus of relations. In *Journal of Symbolic Logic*, 6(3):73-89, 1941.

[Voerm] E. Voermans and J. van der Woude. Relational Tupling. In *EURICS Workshop, Calculational Theories of Program Structure*, part II, Hollum - Ameland, September 1991.

[Weffers0] H.T.G. Weffers. *Implementing Elements of the Spec-Calculus.* Graduating Dissertation. Eindhoven University of Technology, 1993.

[Weffers1] H.T.G. Weffers. The Monad Decomposition Theorem exemplified. Internal note, 1993.

[Wright] J. von Wright. Program inversion in the refinement calculus. In *Information Processing Letters* 37 (1991) 95-100, North-Holland.

Appendix A

Trees, traversals and precedence parsing

In chapter 5 we encountered the following property

$$heap \;\; \sqsupseteq \;\; ((ins\ v) \circ heap)_< \quad .$$

We claimed that a short proof could be given were we to allow the use of induction and point-wise calculation. In this section we present the proof we had in mind.

First we use the unique extension property of paramorphisms to get a recursive definition of $(ins\ v)$:

$$\begin{array}{ll}
 & (ins\ v) \\
= & \qquad \{\text{UEP}\} \\
 & (leaf\ v) \triangledown (P \sqcup Q) \;\circ\; \mathcal{T}.((ins\ v) \vartriangle I) \;\circ\; \mu\mathcal{T} \\
= & \qquad \{\text{def. } \mathcal{T}\} \\
 & (leaf\ v) \triangledown (P \circ H.((ins\ v) \vartriangle I) \;\sqcup\; Q \circ H.((ins\ v) \vartriangle I)) \;\circ\; \mu\mathcal{T} \\
= & \qquad \{\text{below}\} \\
 & (leaf\ v) \triangledown ((topl\ v) \circ tak \;\;\sqcup\;\; tak \circ I{\times}I{\times}(ins\ v)) \;\circ\; \mu\mathcal{T} \\
= & \qquad \{\text{def. } \mu\mathcal{T};\ R \triangledown S \circ (T \triangledown U)^{\cup} \;=\; R \circ T^{\cup} \sqcup S \circ U^{\cup}\} \\
 & (leaf\ v) \circ empty^{\cup} \;\sqcup\; (topl\ v) \circ tak \circ tak^{\cup} \;\sqcup\; tak \circ I{\times}I{\times}(ins\ v) \circ tak^{\cup} \\
= & \qquad \{(leaf\ v) = (topl\ v) \circ empty\} \\
 & (topl\ v) \;\;\sqcup\;\; tak \circ I{\times}I{\times}(ins\ v) \circ tak^{\cup} \;\; .
\end{array}$$

The correctness of the third step is established by the following calculation:

$$\begin{array}{ll}
 & Q \circ H.((ins\ v) \vartriangle I) \\
= & \qquad \{\text{def. } Q\} \\
 & tak \;\circ\; \gg{\times}I{\times}\ll \;\circ\; H.((ins\ v) \vartriangle I)
\end{array}$$

$$
\begin{array}{ll}
= & \quad \{\text{comp. rule } \vartriangle\} \\
& tak \circ (ins\ v)_{>} \times I \times (ins\ v) \\
= & \quad \{(ins\ v)_{>} \sqsubseteq \mu\mathcal{T}\} \\
& tak \circ I{\times}I{\times}(ins\ v) \quad .
\end{array}
$$

The calculation for $P \circ H.((ins\ v) \vartriangle I)$ is left to the reader.
The recursive definition allows us to use induction. We calculate:

$$
\begin{array}{ll}
& (topl\ v).t \in heap \\
\equiv & \quad \{\text{interpretation } (topl\ v)\} \\
& (t, v, \emptyset) \in heap \\
\equiv & \quad \{\text{def. } heap\} \\
& t{\in}heap \ \wedge \ \emptyset{\in}heap \ \wedge \ root.t \leq v \ \wedge \ root.\emptyset \leq v \\
\equiv & \quad \{\text{def. } heap \text{ and } root\} \\
& t{\in}heap \ \wedge \ root.t \leq v \quad .
\end{array}
$$

From the above calculation we have the base step of the induction: $\emptyset \in heap$. For the induction step we have:

$$
\begin{array}{ll}
& (tak \circ I{\times}I{\times}(ins\ v) \circ tak^{\cup}).(l, d, r) \in heap \\
\equiv & \quad \{\text{interpretation}\} \\
& (l, d, (ins\ v).r){\in}heap \\
\equiv & \quad \{\text{def. } heap\} \\
& l{\in}heap \ \wedge \ (ins\ v).r \in heap \ \wedge \ root.l \leq d \ \wedge \ root.(ins\ v).r \leq d \\
\Leftarrow & \quad \{\text{induction}\} \\
& l{\in}heap \ \wedge \ root.l \leq d \ \wedge \ root.(ins\ v).r \leq d \\
\Leftarrow & \quad \{root.(ins\ v).r = v \ \vee \ root.(ins\ v).r = root.r\} \\
& l{\in}heap \ \wedge \ root.l \leq d \ \wedge \ root.r \leq d \ \wedge \ v{\leq}d \\
\Leftarrow & \quad \{\text{def. } heap\} \\
& (l, d, r){\in}heap \wedge v{\leq}d \quad .
\end{array}
$$

$\square$