Functional Polytypic Programming

Patrik Jansson

Department of Computing Science
Chalmers University of Technology and Göteborg University
Göteborg, 2000

Thesis for the Degree of Doctor of Philosophy

Functional Polytypic Programming

Patrik Jansson

Department of Computing Science
Chalmers University of Technology and Göteborg University
Göteborg, Sweden, May 2000

Functional Polytypic Programming
Patrik Jansson
ISBN 91-7197-895-X

Doktorsavhandlingar vid Chalmers Tekniska Högskola
Ny serie nr 1584
ISSN 0346-718x

Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden

Printed at Chalmers, Göteborg, 2000

Functional Polytypic Programming
Patrik Jansson
Department of Computing Science
Chalmers University of Technology and Göteborg University

Abstract

Many algorithms have to be implemented over and over again for different datatypes, either because datatypes change during the development of programs, or because the same algorithm is used for several datatypes. Examples of such algorithms are equality tests, pretty printers, and pattern matchers, and polytypic programming is a paradigm for expressing such algorithms. This dissertation introduces polytypic programming for functional programming languages, shows how to construct and prove properties of polytypic algorithms, presents the language extension PolyP for implementing polytypic algorithms in a type safe way, and presents a number of applications of polytypic programming. The applications include a library of basic polytypic building blocks, PolyLib, and two larger applications of polytypic programming: rewriting and data conversion.

PolyP extends a functional language (a subset of Haskell) with a construct for defining polytypic functions by induction on the structure of user-defined datatypes. Programs in the extended language are translated to Haskell.

PolyLib contains powerful structured recursion operators like catamorphisms, maps and traversals, as well as polytypic versions of a number of standard functions from functional programming: *sum*, *length*, *zip*, $(==)$, $(\leqslant)$, etc. Both the specification of the library and a PolyP implementation are presented.

The first larger application is a framework for polytypic programming on terms. We show that an interface of four functions is sufficient to express polytypic functions for pattern matching, unification and term rewriting. Using this framework, a term rewriting function is specified and transformed into an efficient and correct implementation.

In the second application, a number of functions for polytypic data conversion are implemented and proved correct. The conversions considered include pretty printing, parsing, packing and unpacking of structured data. The conversion functions are expressed in an embedded domain specific language for data conversion (a hierarchy of Haskell's constructor classes).

Keywords: Programming languages, Functional programming, Algebraic datatypes, Polytypic programming, Generic programming
AMS 1991 subject classification 68N15, 68N20

This dissertation is based on the following publications:

1. Patrik Jansson and Johan Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In *Workshop on Generic Programming*, 2000.

2. Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. Submitted for publication, 2000.

3. P. Jansson and J. Jeuring. Polytypic compact printing and parsing. In Doaitse Swierstra, editor, *Proceedings of the 8th European Symposium on Programming, ESOP'99*, volume 1576 of *LNCS*, pages 273–287. Springer-Verlag, 1999.

4. R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1999.

5. Patrik Jansson and Johan Jeuring. Functional pearl: Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, September 1998.

6. P. Jansson and J. Jeuring. PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand, June 1998.

7. PolyP — a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

8. J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming '96*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, 1996.

Available from the Polytypic programming WWW page [49].

`http://www.cs.chalmers.se/~patrikj/poly/`

Contents

1 Introduction **1**
1.1 What is a polytypic function? 2
1.2 Polymorphism and polytypism 3
1.3 Why polytypic programming? 4
1.4 Scope 5
1.5 Approaches to writing polytypic programs 6
1.6 The PolyP system 7
1.7 Overview 7

2 Prelude **9**
2.1 Context 9
2.2 The function type 9
2.3 The disjoint sum type 10
2.4 The unit type 11
2.5 The pair type 11
2.6 The Haskell bottom 12
2.7 Booleans, truth values and predicates 13
2.8 Computations that may fail 15
2.9 Polymorphic lists 16
2.10 Overloading and classes 17
2.11 Fixed points 17
2.11.1 Fixed point induction 18
2.11.2 Explaining fixed point induction 20

3 Basic polytypic programming **23**
3.1 The structure of lists 24
3.2 Catamorphisms and fusion for lists 26
3.3 The structure of trees 29
3.4 Pattern functors 31
3.5 In and out of a regular datatype 32
3.6 The **polytypic** construct 33
3.7 Catamorphisms and maps 35
3.8 Catamorphisms on specific datatypes 37

3.9 Separate: a simple PolyP program . . . 38
3.10 Polytypic laws . . . 40

4 PolyP — a polytypic programming language extension 45
4.1 Type inference . . . 45
4.1.1 The core language . . . 46
4.1.2 The polytypic language extension . . . 47
4.1.3 Unification . . . 50
4.1.4 Type checking the **polytypic** construct . . . 50
4.2 Semantics . . . 52
4.3 Code generation . . . 55
4.4 Implementation . . . 56
4.5 Conclusions and future work . . . 56

5 PolyLib — a polytypic function library 59
5.1 Describing polytypic functions . . . 60
5.1.1 Notation and naming . . . 60
5.1.2 Library overview . . . 60
5.2 Recursion operators . . . 61
5.3 Zips . . . 64
5.4 Monad operations . . . 67
5.5 Flatten functions . . . 69
5.6 Miscellaneous . . . 70
5.7 Conclusions . . . 71

6 Rewriting 73
6.1 Introduction . . . 73
6.1.1 An example rewriting system . . . 74
6.1.2 Polytypic rewriting . . . 75
6.2 A term interface . . . 76
6.2.1 Terms . . . 76
6.2.2 Polytypic *Term* instances . . . 78
6.2.3 Combinators on terms . . . 79
6.2.4 Laws for term combinators . . . 80
6.3 Substitutions, matching and unification . . . 82
6.3.1 Substitutions . . . 82
6.3.2 Matching . . . 83
6.3.3 Unification . . . 84
6.4 Rewriting . . . 86
6.4.1 One step rewriting . . . 87
6.4.2 Rewriting to normal form . . . 88
6.4.3 Concrete fixed points . . . 90
6.4.4 Improving rewriting . . . 90

6.4.5 Efficiency comparison . . . 95
6.5 Proofs . . . 96
6.5.1 Proofs of term combinator laws . . . 97
6.5.2 Proofs of rewriting transformations . . . 101
6.6 Conclusions . . . 104

7 Polytypic Data Conversion Programs **105**
7.1 Introduction . . . 105
7.1.1 Data conversion programs . . . 106
7.1.2 Constructing data conversion programs . . . 107
7.2 Shape . . . 109
7.2.1 Function *separate* . . . 109
7.2.2 Function *combine* . . . 110
7.2.3 Function *combine* is the inverse of *separate* . . . 111
7.3 Arrows and laws . . . 111
7.3.1 Basic definitions and laws for arrows . . . 111
7.3.2 A class for arrows . . . 114
7.3.3 An inverse law for arrow products . . . 116
7.3.4 Fixed point induction and arrows . . . 117
7.4 Arrow maps . . . 117
7.4.1 The arrow maps are inverses . . . 118
7.5 Packing . . . 121
7.5.1 The construction of the packing function . . . 123
7.6 Pretty printing . . . 128
7.6.1 More arrow classes . . . 128
7.6.2 Definition of *pshow* and *pread* . . . 131
7.7 Generating arrow instances . . . 137
7.7.1 An instance for *ArrowReadShow* . . . 140
7.8 Results and conclusions . . . 141

8 Related work **145**
8.1 BMF ≅ Squiggol . . . 145
8.2 Theories of datatypes . . . 146
8.3 Beyond regular datatypes . . . 146
8.4 Specific polytypic functions . . . 147
8.5 Type systems . . . 148
8.6 Implementations . . . 149
8.7 Polytypic transformations and proofs . . . 149
8.8 Imperative Polytypic Programming . . . 150

Acknowledgments **153**

Bibliography **155**

A An implementation of PolyLib **165**
A.1 Structured recursion operators 165
A.2 *Crush* 166
A.3 Monadic recursion operators 167
A.4 *Thread* 169
A.5 *ThreadFuns* 170
A.6 *Propagate* 171
A.7 *Zip* 171
A.8 *Equal* 174
A.9 *Compare* 175
A.10 *Flatten* 176
A.11 *Sum* 177
A.12 *CrushFuns* 177
A.13 *ConstructorName* 178

Chapter 1

Introduction

The ability to name and reuse common patterns of computation as higher-order functions is at the heart of the power of functional languages. Higher-order functions like maps and catamorphisms capture very general programming idioms that are useful in many contexts. This kind of polymorphic functions enables us to abstract away from the unimportant details of an algorithm and concentrate on its essential structure.

The type of a polymorphic function has type parameters, but all monomorphic instances of the function can use identical code. A generalization is to parametrize also the function definition on types. Functions that are parametrized in this way are called *polytypic functions* [61]. Equality functions, pretty printers and parsers, traversal functions and other recursion combinators are all examples of polytypic functions.

While a normal polymorphic function is an algorithm that is independent of the type parameters, the class of instances of a polytypic function contains functions that are different, but which share a common structure. Any algorithm in the class can be obtained by instantiating a template algorithm with (the structure of) a datatype.

Other terms used for polytypism in the literature are *structural polymorphism* (Ruehr [94]), *type parametric programming* (Sheard [97]), *generic programming* (Bird, de Moor and Hoogendijk [7]), *polynomial polymorphism* (Jay [55]), *shape polymorphism* (Jay [56]) and *type indexed functions* (Hinze [35]). A detailed overview of polytypic programming in related work is presented in Chapter 8.

In the sequel we will assume that the reader has some knowledge of a functional programming language, preferably Haskell [90]. This chapter explains briefly what polytypic functions are, why they are useful and how they can be implemented. It also describes the scope of this dissertation and presents an overview of the following chapters.

1.1 What is a polytypic function?

To give an example of what a polytypic function is we show that the definitions of the function *sum* on different datatypes share a common structure. The *sum* function takes a structure containing integers and returns the sum of all the integers in the structure. The normal *sum* function for lists can be defined as follows in the functional language Haskell:

$$\begin{array}{lcl} sum :: [\,Int\,] & \rightarrow & Int \\ sum\ [\,] & = & 0 \\ sum\ (x : xs) & = & x + sum\ xs \end{array}$$

We define *sum* on the following datatypes:

$$\begin{array}{lcl} \mathbf{data}\ [\,a\,] & = & [\,] \mid a : [\,a\,] \\ \mathbf{data}\ Tree\ a & = & Leaf\ a \mid Bin\ (Tree\ a)\ (Tree\ a) \\ \mathbf{data}\ Maybe\ a & = & Nothing \mid Just\ a \\ \mathbf{data}\ Rose\ a & = & Fork\ a\ [\,Rose\ a\,] \end{array}$$

We can define the function *sum* for all of these datatypes (instantiated on integers) using catamorphisms. A catamorphism is a function that recursively replaces constructors with functions. We write $cata_D\ \{\,C_i \mapsto e_i\,\}$ for the catamorphism on the datatype $D\ a$ that replaces the constructors C_i with the expressions e_i.

$$\begin{array}{lcl} sum_{[\,]} & :: & [\,Int\,] \rightarrow Int \\ sum_{[\,]} & = & cata_{[\,]}\ \{\,[\,] \mapsto 0, (:) \mapsto (+)\,\} \\ sum_{Tree} & :: & Tree\ Int \rightarrow Int \\ sum_{Tree} & = & cata_{Tree}\ \{\,Leaf \mapsto id, Bin \mapsto (+)\,\} \\ sum_{Maybe} & :: & Maybe\ Int \rightarrow Int \\ sum_{Maybe} & = & cata_{Maybe}\ \{\,Nothing \mapsto 0, Just \mapsto id\,\} \\ sum_{Rose} & :: & Rose\ Int \rightarrow Int \\ sum_{Rose} & = & cata_{Rose}\ \{\,Fork \mapsto \lambda a\ l \rightarrow a + sum_{[\,]}\ l\,\} \end{array}$$

We can already see some patterns in the parameters of the catamorphism: the two nullary constructors $[\,]$ and *Nothing* are both replaced by 0 and the two unary constructors *Leaf* and *Just* are replaced by the identity function *id*. The binary constructors ((:), *Bin* and *Fork*) are replaced by functions that sum the subexpressions. All the definitions of *sum* are instances of the following polytypic definition of *psum*:

$$\begin{array}{l} psum :: Regular\ d\ \Rightarrow\ d\ Int \rightarrow Int \\ psum\ =\ cata\ fsum \end{array}$$

$$
\begin{array}{lll}
\textbf{polytypic}\ \mathit{fsum} & :: & f\ \mathit{Int}\ \mathit{Int} \to \mathit{Int} \\
\quad = \textbf{case}\ f\ \textbf{of} & & \\
\qquad g + h & \to & \mathit{either}\ \mathit{fsum}\ \mathit{fsum} \\
\qquad g * h & \to & \lambda(x, y) \to \mathit{fsum}\ x + \mathit{fsum}\ y \\
\qquad \mathit{Empty} & \to & \lambda x \to 0 \\
\qquad \mathit{Par} & \to & \mathit{id} \\
\qquad \mathit{Rec} & \to & \mathit{id} \\
\qquad d@g & \to & \mathit{psum} \circ \mathit{pmap}\ \mathit{fsum} \\
\qquad \mathit{Const}\ t & \to & \lambda x \to 0
\end{array}
$$

Figure 1.1: The definition of *fsum*

Function *fsum* is defined (in Figure 1.1) by induction over the pattern functor f that captures the structure of the regular type constructor d. The polytypic definition of function *cata* and the explanation of function *fsum* will have to wait until the **polytypic** construct is defined in Section 3.6.

Higher-order functions and polytypic functions can be used together to obtain even more general definitions. Exactly the same structure as that used for *psum*, can be used to define the polytypic function *conc*, which concatenates all lists in a structure of type $d\ [a]$. We just replace 0 by $[\,]$ and $(+)$ with list concatenation $(+\!\!+)$ in the definition of *fsum* to obtain *fconc*.

$$
\begin{array}{l}
\mathit{conc} :: \mathit{Regular}\ d \Rightarrow d\ [a] \to [a] \\
\mathit{conc} = \mathit{cata}\ \mathit{fconc}
\end{array}
$$

Both *psum* and *conc* are polytypic functions and thus parametrized on the type constructor d. By abstracting over the operator and its unit, we can generalize *psum* (*fsum*) and *conc* (*fconc*) to the polytypic function *crush* (*fcrush*).

$$
\begin{array}{l}
\mathit{crush} :: \mathit{Regular}\ d \Rightarrow (a \to a \to a) \to a \to d\ a \to a \\
\mathit{crush}\ \mathit{op}\ e = \mathit{cata}\ (\mathit{fcrush}\ \mathit{op}\ e)
\end{array}
$$

where $\mathit{fsum} = \mathit{fcrush}\ (+)\ 0$ and $\mathit{fconc} = \mathit{fcrush}\ (+\!\!+)\ [\,]$. These functions, and many others, are described in PolyLib (Chapter 5 and Appendix A).

1.2 Polymorphism and polytypism

A parametric polymorphic function such as

$$\mathit{head} :: [a] \to a$$

can be seen as a family of functions — one for each instance of a as a monomorphic type. Parametricity implies that *head* can make no assumptions about the type a. Thus all the functions in the family are essentially the same.

An *ad hoc* polymorphic function such as

$$(+) :: \mathit{Num}\ a \Rightarrow a \to a \to a$$

is also a family of functions, one for each instance in the *Num* class. These instances may be completely unrelated and each instance is defined separately. In almost all cases, automatic type inference can be used to find the appropriate instance for any given occurrence of the (+) operator.

The polymorphism of a polytypic function such as

$$\mathit{psum} :: \mathit{Regular}\ d \Rightarrow d\ \mathit{Int} \to \mathit{Int}$$

is somewhere in between parametric and *ad hoc* polymorphism. A polytypic function can be seen as a type indexed family of functions. A single definition of *psum* suffices, but *psum* has different instances in different contexts. Here the compiler generates instances from the definition of the polytypic function and the type in the context where it is used. A polytypic function may also be parametric polymorphic: function $\mathit{size} :: \mathit{Regular}\ d \Rightarrow d\ a \to \mathit{Int}$, which returns the size of a value of an arbitrary datatype, is both polytypic and parametric polymorphic.

Meertens [76] gives a nice example of the power of parametric polymorphism: Suppose we want a function to swap two integers: $\mathit{swap} :: (\mathit{Int}, \mathit{Int}) \to (\mathit{Int}, \mathit{Int})$. This is not a very hard problem to solve, but there are infinitely many type correct but wrong solutions. (Two are *id* and $\lambda(x, y) \to (y + 1, x)$.) If we generalize this function to the polymorphic function $\mathit{swap} :: (a, b) \to (b, a)$, then we get a much more useful program and we *can't* make it wrong while type correct. (Strictly speaking this is true only in a strongly normalizing language. If we have bottoms, or non-terminating computations, as in *CPO* and in Haskell, then we can still write a few non-terminating (wrong) versions.) Similarly, even when a function may be needed only for one specific datatype, it may be helpful to define it polytypically to reduce the risk of making a mistake.

1.3 Why polytypic programming?

Polytypic programming offers a number of benefits:

Reusability: Polytypism extends the power of polymorphic functions to allow classes of related algorithms to be described in one definition. For example, the class of printing functions for different datatypes can be expressed as one polytypic *show* function. Thus polytypic functions are very well suited for building program libraries. PolyLib (Chapter 5) is an example of such a library.

Adaptivity: Polytypic programs automatically adapt to changing datatypes. For example, if we add a constructor *Node* (*Tree* a) a (*Tree* a) to the datatype *Tree* a, then the same polytypic sum function can still be used to sum all integers in elements of the (new) tree type. This adaptivity reduces the need for time consuming and boring rewrites of trivial functions and eliminates the associated risk of making mistakes.

Closure and orthogonality: Currently some polytypic functions can be *used* but not *defined* in ML (the equality function(s)) and Haskell (the members of the derived classes). This asymmetry can be removed by extending these languages with polytypic definitions.

Applications: Some problems are polytypic by nature: maps and traversals (Section 5.4), pretty printing and parsing (Section 7.6), data compression (Section 7.5), matching (Section 6.3.2), unification (Section 6.3.3), term rewriting (Section 6.4), ...

Provability: More general functions means more general proofs. If we consider polytypic proofs, then each of the earlier benefits obtains an additional interpretation: we get reusable proofs, adaptive proofs, less *ad hoc* semantics of programming languages and new proofs of properties of printing and parsing (Section 7.6), packing (Section 7.5), term rewriting (Section 6.4) etc.

1.4 Scope

As the title suggests this dissertation is about polytypic programming for functional programming languages. More specifically, the programs in this dissertation are written in the functional programming language Haskell 98 [90] extended with with support for polytypic definitions provided by the authors language extension PolyP (Chapter 4).

A polytypic function can be applied to values of a large class of datatypes, but some restrictions apply. We require that a polytypic function is applied to values of *regular* datatypes only. A datatype $D\ a$ is regular if it is not mutually recursive, contains no function spaces, and if the arguments of the datatype constructor on the left- and right-hand side in its definition are the same. The collection of regular datatypes contains most conventional recursive datatypes, such as *Nat*,

$[a]$, and different kinds of trees. We use the constructor class *Regular* to represent the collection of regular datatypes.

Polytypic functions can be defined on a larger class of datatypes, including multiple parameter datatypes [58], mutually recursive datatypes [14, 35, 45], datatypes with function spaces [26, 78] and nested datatypes [8, 34] but we will not discuss these extensions.

1.5 Approaches to writing polytypic programs

There are various ways to implement polytypic programs in a typed language. (Polytypic programs can be implemented in untyped languages like Lisp or C but without any (static) type safety. We only consider strongly typed languages in this dissertation.) Three possibilities are:

- using a universal datatype;
- using higher-order polymorphism and constructor classes;
- using a special syntactic construct.

Polytypic functions can be implemented by defining a universal datatype, on which we define the functions we want to have available for large classes of datatypes. These polytypic functions can be used on a specific datatype if we provide translation functions to and from the universal datatype. An advantage of using a universal datatype for implementing polytypic functions is that we do not need a language extension for writing polytypic programs. However, using universal datatypes has several disadvantages: type information is lost in the translation phase to the universal datatype, and type errors can occur when programs are run. Furthermore, different people will use different universal datatypes, which will make program reuse more difficult.

If we use higher-order polymorphism and constructor classes for defining polytypic functions (as in Jones [65]), then type information is preserved, and we can use a functional language such as Haskell for implementing polytypic functions. In this style all regular datatypes are represented by the type

$$\textbf{data}\ \mathit{Mu}\ f\ a\ =\ \mathit{In}\ (f\ a\ (\mathit{Mu}\ f\ a))$$

and the class system is used to overload functions like *map* and *cata*. However, writing such programs is rather cumbersome: programs become cluttered with instance declarations, and type declarations become cluttered with contexts. And the user still has to write all translation functions.

Because the first two solutions to writing polytypic functions are unsatisfactory, we have extended (a subset of) Haskell with a syntactic construct for defining polytypic functions. We will use the name PolyP both for the extension and the resulting language.

1.6 The PolyP system

PolyP is an extension of a functional language that allows programmers to define and use polytypic functions. The underlying language in this dissertation is a subset of Haskell and hence lazy, but this is not essential for the polytypic extension. The extension introduces a new kind of (top level) definition, the **polytypic** construct, used to define functions by induction over the structure of datatypes. Because datatype definitions can express sum-, product-, parametric and recursive types, the **polytypic** construct must handle these cases.

PolyP type checks polytypic value definitions and, when using polytypic values, types are automatically inferred. (Just as in Haskell, sometimes explicit type annotations are needed to resolve overloading.) The type inference algorithm is based upon Jones' theories of qualified types [64] and higher-order polymorphism [66]. The semantics of PolyP is defined by adding type arguments to polytypic functions in a dictionary passing style. We give a type based translation from PolyP to Haskell that uses partial evaluation to remove all dictionary values at compile time. Thus we avoid run time overhead for creating instances of polytypic functions.

The compiler for PolyP is still under development, and has a number of limitations. Polytypic functions can only be applied to values of regular datatypes. The underlying subset of Haskell lacks many useful constructs such as modules and instance declarations. Extensions to handle multiple type arguments, mutually recursive datatypes and all of Haskell are planned for the forthcoming successor of PolyP: Generic Haskell [33].

1.7 Overview

The dissertation contains an introduction to polytypic programming, a description of the language extension PolyP and its library PolyLib and two larger polytypic applications: term rewriting and data conversion.

Chapter 2 is a non-polytypic prelude to the rest of the dissertation. It defines notation, useful functions and laws to be used in the sequel.

Chapter 3 is an introduction to functional polytypic programming. This chapter is the one you should read if you want to learn how to write and use polytypic functions: it defines catamorphisms, polytypic *map* functions, function *psum* used in the preceding example and presents the **polytypic** construct which is used for defining polytypic functions by induction over the structure of user-defined datatypes. This chapter also presents some polytypic proof rules and uses these rules to prove properties about polytypic functions.

Chapter 4 briefly describes the theory and implementation of PolyP: the type system that preserves Haskell-like type inference provided the **polytypic** construct is explicitly typed, and the semantics in terms of a translation of PolyP-programs into Haskell. The theory from this chapter is not essential for reading the rest of the dissertation. The chapter is based on the POPL'97 paper *PolyP — a polytypic programming language extension* [46].

Chapter 5 presents a library of polytypic building blocks that can be used in applications. Each function is presented with its type and a brief description of what it does and how it is related to other polytypic functions. The chapter is a revised version of the paper *PolyLib — a polytypic function library* [51]. An implementation of PolyLib in the language extension PolyP is incuded in Appendix A.

Chapter 6 presents the first larger polytypic application: term rewriting. This chapter presents an interface for polytypic programming on terms, and uses this interface to describe polytypic algorithms for matching, unification and efficient term rewriting together with some correctness proofs. The chapter is an extended version of the article *A framework for polytypic programming on terms, with an application to rewriting* [52].

Chapter 7 is the second larger polytypic application: data conversion. It presents polytypic functions for maps and traversals, data compression, and pretty printing. For each conversion, a pair of inverse functions is constructed together with a proof of correctness. The conversion functions are expressed in an embedded domain specific language for data conversion. The embedded language is defined as a hierarchy of Haskell's constructor classes, based on Hughes' *Arrow*s [42].

Chapter 8 gives an overview of polytypism in related work. It describes the origins of polytypism, the different approaches used to express, type check and implement polytypism and gives many references to further reading about polytypism.

Chapter 2

Prelude

This chapter is for the dissertation what the standard prelude is for Haskell: a collection of common resources which can be used everywhere without explicitly having to define them locally or import them. The prelude is divided into sections that present some notation and a few basic datatypes with associated operations and laws.

2.1 Context

For specifications, program code and proofs, we use Haskell [90] notation with a few typographical enhancements to improve readability. In a few places these enhancements clash with the formal syntax for Haskell. For example, we use (;) for forward composition (that is, $f \,;\, g = g \circ f$) although Haskell uses the semicolon only as a separator. Where possible, included program code is automatically pretty printed from Haskell or PolyP source code to avoid errors.

In category theory, a functor is a mapping between categories that preserves the algebraic structure of the category. Because a category consists of objects (types) and arrows (functions), a functor consists of two parts: a definition on types, and a definition on functions. We normally work in the category CPO of complete partial orders and continuous functions between them.

2.2 The function type

The Haskell type of partial functions from a to b is written $a \to b$ and a lambda expression with pattern a and body b is written $\lambda a \to b$. The identity function, constant function and function composition are defined as follows:

$$
\begin{array}{lcl}
\mathit{id} & :: & a \to a \\
\mathit{id}\ x & = & x \\
\mathit{const} & :: & a \to b \to a \\
\mathit{const}\ k\ \_ & = & k \\
(\circ) & :: & (b \to c) \to (a \to b) \to (a \to c) \\
(f \circ g)\ x & = & f\ (g\ x)
\end{array}
$$

Functions of multiple arguments are normally curried in contrast to languages like Ada, Java and SML where functions normally take a tuple of arguments. The functions *curry* and *uncurry* convert between these two views:

$$
\begin{array}{lcl}
\mathit{curry} & :: & ((a, b) \to c) \to (a \to b \to c) \\
\mathit{curry}\ f\ x\ y & = & f\ (x, y) \\
\mathit{uncurry} & :: & (a \to b \to c) \to ((a, b) \to c) \\
\mathit{uncurry}\ f\ p & = & f\ (\mathit{fst}\ p)\ (\mathit{snd}\ p)
\end{array}
$$

2.3 The disjoint sum type

The disjoint sum type *Either a b* in Haskell consists of left-tagged elements of type a, and right-tagged elements of type b, and has constructors *Left* and *Right*, which inject elements into the left and right component of a sum respectively.

$$\textbf{data}\ \mathit{Either}\ a\ b\ =\ \mathit{Left}\ a \mid \mathit{Right}\ b$$

$$
\begin{array}{l}
\mathit{Left} :: a \to \mathit{Either}\ a\ b \\
\mathit{Right} :: b \to \mathit{Either}\ a\ b
\end{array}
$$

Function $l \triangledown r$ (written *either* l r in Haskell) is a shorthand notation for case analysis. Function $(\triangledown)$ is the catamorphism on *Either*. It takes a function l of type $a \to c$ and a function r of type $b \to c$, and replaces *Left* with l and *Right* with r:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{(\triangledown) :: (a \to c) \to (b \to c) \to (\mathit{Either}\ a\ b \to c)} \\
(l \triangledown r)\ (\mathit{Left}\ x) & = & l\ x \\
(l \triangledown r)\ (\mathit{Right}\ y) & = & r\ y
\end{array}
$$

The operator $(+)$ is used to apply either l or r inside *Left* or *Right*. It is a two-argument mapping function on *Either*.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{(+) :: (a \to c) \to (b \to d) \to (\mathit{Either}\ a\ b \to \mathit{Either}\ c\ d)} \\
(l + r)\ (\mathit{Left}\ x) & = & \mathit{Left}\ (l\ x) \\
(l + r)\ (\mathit{Right}\ y) & = & \mathit{Right}\ (r\ y)
\end{array}
$$

The following functional definition of $(+)$ is equivalent and easier to calculate with:

$$l + r \;=\; (\mathit{Left} \circ l) \triangledown (\mathit{Right} \circ r)$$

Function $(+)$ satisfies two functor laws and operator $(\triangledown)$ satisfies two fusion laws:

$$\begin{array}{lcl} id + id & = & id \\ (f + g) \circ (h + i) & = & (f \circ h) + (g \circ i) \\ f \circ (g \triangledown h) & = & (f \circ g) \triangledown (f \circ h) \\ (f \triangledown g) \circ (h + i) & = & (f \circ h) \triangledown (g \circ i) \end{array}$$

2.4 The unit type

The nullary product type and its only constructor are both written as ():

$$\textbf{data}\ () \;=\; ()$$

2.5 The pair type

The binary product type and its elements are written as pairs (a, b). Functions *fst* and *snd* are the two projections.

$$\begin{array}{lcl} \textbf{data}\ (a, b) & = & (a, b) \\ \mathit{fst}\ (a, b) & = & a \\ \mathit{snd}\ (a, b) & = & b \end{array}$$

The duals of $(\triangledown)$ and $(+)$ are $(\vartriangle)$ and $(*)$, respectively.

$$\begin{array}{l} (\vartriangle) :: (a \to b) \to (a \to c) \to (a \to (b, c)) \\ (f \vartriangle s)\ x \;=\; (f\ x, s\ x) \end{array}$$

The operator $(*)$ is the analogue of map on products.

$$\begin{array}{l} (*) :: (a \to c) \to (b \to d) \to ((a, b) \to (c, d)) \\ (f * s)\ (x, y) \;=\; (f\ x, s\ y) \end{array}$$

By analogy with the definition of $(+)$ we have an equivalent function level definition:

$$f * g \;=\; (f \circ \mathit{fst}) \vartriangle (g \circ \mathit{snd})$$

Function ($\divideontimes$) satisfies two bifunctor laws and operator ($\vartriangle$) satisfies two fusion laws:

$$
\begin{array}{lcl}
id \divideontimes id & = & id \\
(f \divideontimes g) \circ (h \divideontimes i) & = & (f \circ h) \divideontimes (g \circ i) \\
(f \vartriangle g) \circ h & = & (f \circ h) \vartriangle (g \circ h) \\
(f \divideontimes g) \circ (h \vartriangle i) & = & (f \circ h) \vartriangle (g \circ i)
\end{array}
$$

2.6 The Haskell bottom

All Haskell types have a bottom element denoting a non-terminating computation and we can define a polymorphic value $\bot$ by the following trivial recursive definition:

$$
\begin{array}{l}
\bot :: a \\
\bot = \bot
\end{array}
$$

In contrast to most theoretical frameworks the function type, the empty and the binary product type in Haskell are all lifted:

$$
\begin{array}{lcl}
(\lambda x \to \bot) & \neq & \bot :: a \to b \\
() & \neq & \bot :: () \\
(\bot, \bot) & \neq & \bot :: (a, b)
\end{array}
$$

Among other things, this means that for Haskell:

- η-expansion is not semantics preserving: if $f = \bot :: a \to b$, then

$$\lambda x \to f\ x \;=\; \lambda x \to \bot\ x \;=\; \lambda x \to \bot \;\neq\; \bot \;=\; f\ .$$

- The type () is not a terminal object as it has two elements: $\bot$ and ().
- And we do not have surjective pairing: if $p = \bot :: (a, b)$, then

$$(\mathit{fst}\ p, \mathit{snd}\ p) \;=\; (\mathit{fst}\ \bot, \mathit{snd}\ \bot) \;=\; (\bot, \bot) \;\neq\; \bot \;=\; p\ .$$

To sum up — almost no laws from *CPO* hold in Haskell! As this would lead to considerable problems in the detailed proofs, we restrict ourselves to the unlifted versions of these types. As we use Haskell for the implementations this means, strictly speaking, that most of the results presented in this dissertation are not proved for the actual running code but for idealized versions. This has not turned out to be a problem in practice.

2.7 Booleans, truth values and predicates

The boolean values *False* and *True* are the constructors of the type *Bool*:

$$\mathbf{data}\ \mathit{Bool} \;=\; \mathit{False} \mid \mathit{True}$$

Note that the Haskell type *Bool* contains a least value, $\perp$, in addition to the two truth values. When we really need only truth values we use the type $\mathit{Truth} = \{\mathit{False}, \mathit{True}\}$ and convert from *Bool* to *Truth* by identifying *False* and $\perp$:

$$\begin{array}{lcl} \lfloor\rfloor :: \mathit{Bool} & \to & \mathit{Truth} \\ \lfloor \mathit{True} \rfloor & = & \mathit{True} \\ \lfloor \_ \rfloor & = & \mathit{False} \end{array}$$

The expression $\lfloor b \rfloor$ means "the calculation of b terminates with the value *True*" and is pronounced "b is true" for short.

We have the common operations implication $(\Rightarrow)$, and $(\wedge)$, or $(\vee)$, and negation $(\neg)$ for calculating with *Bool*eans and with *Truth* values, and **if**-expressions to select between two expressions. We use the same syntax for operations on *Bool* and operations on *Truth*.

We often work with predicates instead of booleans to simplify calculations. We often use the same syntax for the pointwise lifted operations.

$$\begin{array}{lcl} \mathit{false}, \mathit{true} & :: & a \to \mathit{Bool} \\ \mathit{false} & = & \mathit{const}\ \mathit{False} \\ \mathit{true} & = & \mathit{const}\ \mathit{True} \\ (\Rightarrow), (\wedge), (\vee) & :: & (a \to \mathit{Bool}) \to (a \to \mathit{Bool}) \to (a \to \mathit{Bool}) \\ p \Rightarrow q & = & \lambda x \to p\ x \Rightarrow q\ x \\ p \wedge q & = & \lambda x \to p\ x \wedge q\ x \\ p \vee q & = & \lambda x \to p\ x \vee q\ x \\ \mathbf{iff}\ p\ \mathbf{then}\ t\ \mathbf{else}\ e & = & \lambda x \to \mathbf{if}\ p\ x\ \mathbf{then}\ t\ x\ \mathbf{else}\ e\ x \\ \lfloor\rfloor & :: & (a \to \mathit{Bool}) \to (a \to \mathit{Truth}) \\ \lfloor p \rfloor & = & \lambda x \to \lfloor p\ x \rfloor \end{array}$$

As an example of the use of the lifted boolean operations we can specify pre- and post-conditions for a function f:

$$\lfloor \mathit{pre} \rfloor \Rightarrow \lfloor \mathit{post} \circ f \rfloor\ .$$

Expanding the definitions of the lifted operators this is equivalent to:

$$\lambda x \to \lfloor \mathit{pre}\ x \rfloor \Rightarrow \lfloor \mathit{post}\ (f\ x) \rfloor\ .$$

If this predicate equals *true* (that is, for all x the body is *True*), then f satisfies its specification.

The Haskell equality test $(==) :: Eq\ a \Rightarrow a \to a \to Bool$ is also lifted:

$$\begin{array}{lcl} (===) & :: & Eq\ b \Rightarrow (a \to b) \to (a \to b) \to (a \to Bool) \\ f === g & = & \lambda x \to f\ x == g\ x \end{array}$$

The lifted version of the law $(f\ x == f\ y) \Leftarrow (x == y)$ becomes:

Lemma 2.1 *Cancel* $(f\circ)$*:*

$$(f \circ g === f \circ h) \quad \Leftarrow \quad (g === h)$$

We will often reason about functions that are equal when restricted to a subset of their domains:

Definition 2.2 *Function equality on a subset:*

$$\begin{array}{l} (\ \overset{\cdot}{===}\ ) :: Eq\ a \Rightarrow (b \to Bool) \to (b \to a) \to (b \to a) \to (b \to Truth) \\ f \overset{p}{===} g \ = \ \lambda x \to \lfloor p\ x \rfloor \Rightarrow \lfloor f\ x == g\ x \rfloor \end{array}$$

or, equivalently, using the lifted operations:

$$f \overset{p}{===} g \ = \ \lfloor p \rfloor \Rightarrow \lfloor f === g \rfloor$$

We will later use the following property of $(\ \overset{p}{===}\ )$:

Lemma 2.3 *Factor* $(\ \overset{p}{===}\ )$*:*

$$g \circ f \overset{p \circ f}{===} h \circ f === (g \overset{p}{===} h) \circ f$$

The lifted version of $\lfloor\,\rfloor$ satisfies the following laws:

Lemma 2.4 *Factor out* f *from* $\lfloor\,\rfloor$*:*

$$\lfloor p \circ f \rfloor === \lfloor p \rfloor \circ f$$

Lemma 2.5 *Cancel* $(\circ f)$*:*

$$\lfloor p \circ f \rfloor \Leftarrow \lfloor p \rfloor$$

Simple laws for booleans lift immediately to predicates:

Law 2.6 *(exp1):* $(a \lor b) \Rightarrow c \;\equiv\; (a \Rightarrow c) \land (b \Rightarrow c)$

Law 2.7 *(exp2):* $a \Rightarrow (b \lor c) \;\equiv\; (a \Rightarrow b) \lor (a \Rightarrow c).$

Laws for **iff then else** :

Lemma 2.8 iff then else *-fusion:*
For all strict f:

$$f\ (\mathbf{iff}\ b\ \mathbf{then}\ p\ \mathbf{else}\ q) \;=\; \mathbf{iff}\ b\ \mathbf{then}\ f\ p\ \mathbf{else}\ f\ q$$

Lemma 2.9 iff p **then** p

$$(\mathbf{iff}\ p\ \mathbf{then}\ \lfloor p \rfloor\ \mathbf{else}\ x) \;=\; (\mathbf{iff}\ p\ \mathbf{then}\ \mathit{true}\ \mathbf{else}\ x)$$

Lemma 2.10 *Expressing* $(\lor)$ *using* **iff then else** *:*

$$\lfloor p \lor q \rfloor \;=\; \mathbf{iff}\ \lfloor p \rfloor\ \mathbf{then}\ \mathit{true}\ \mathbf{else}\ \lfloor q \rfloor$$

2.8 Computations that may fail

The datatype *Maybe a* is used to model computations that may fail to give a result.

$$\mathbf{data}\ \mathit{Maybe}\ a \;=\; \mathit{Nothing} \mid \mathit{Just}\ a$$

For example, we can define the expression *divide m n* to be equal to *Nothing* if n equals zero, and *Just* $(m\ /\ n)$ otherwise. A function that handles values of type *Maybe a* consists of two components: a component that deals with *Nothing*, and a component that deals with values of the form *Just x*.

$$\begin{array}{l}
\mathit{maybe} :: b \to (a \to b) \to \mathit{Maybe}\ a \to b \\
\mathit{maybe}\ n\ j\ \mathit{Nothing} \;=\; n \\
\mathit{maybe}\ n\ j\ (\mathit{Just}\ x) \;=\; j\ x
\end{array}$$

Function *maybe* is an example of a catamorphism. Function *mapM* takes a function f, and a value of type *Maybe a*, and returns *Nothing* in case the argument equals *Nothing*, and *Just* $(f\ x)$ in case the argument equals *Just x*.

$$\begin{array}{l}
\mathit{mapM} :: (a \to b) \to \mathit{Maybe}\ a \to \mathit{Maybe}\ b \\
\mathit{mapM}\ f \;=\; \mathit{maybe}\ \mathit{Nothing}\ (\mathit{Just} \circ f)
\end{array}$$

Function *mapM* satisfies two functor laws and function *maybe* satisfies two fusion laws:

$$
\begin{array}{lcl}
\mathit{mapM}\ \mathit{id} & === & \mathit{id} \\
\mathit{mapM}\ f \circ \mathit{mapM}\ g & === & \mathit{mapM}\ (f \circ g) \\
f \circ \mathit{maybe}\ n\ j & === & \mathit{maybe}\ (f\ n)\ (f \circ j) \\
\mathit{maybe}\ n\ j \circ \mathit{mapM}\ f & === & \mathit{maybe}\ n\ (j \circ f)
\end{array}
$$

It is sometimes useful to have predicates to test for *Just* and *Nothing*:

$$
\begin{array}{lcl}
\mathit{isJust}, \mathit{isNothing} & :: & \mathit{Maybe}\ a \to \mathit{Bool} \\
\mathit{isJust}\ (\mathit{Just}\ \_) & = & \mathit{True} \\
\mathit{isJust}\ \mathit{Nothing} & = & \mathit{False} \\
\mathit{isNothing}\ \mathit{Nothing} & = & \mathit{True} \\
\mathit{isNothing}\ (\mathit{Just}\ \_) & = & \mathit{False}
\end{array}
$$

2.9 Polymorphic lists

The polymorphic list datatype in Haskell is written $[a]$ and has a constructor $[\,]$ for the empty list and an infix $(:)$ for prepending a value to a list. There is also syntactic sugar for lists: for example $[1,2,3]$ means $1:2:3:[\,]$.

$$\mathbf{data}\ [a]\ =\ [\,] \mid a : [a]$$

The syntax for the list constructors is a little different from other datatypes. We will sometimes use a definition more in line with other user-defined datatypes:

$$\mathbf{data}\ \mathit{List}\ a\ =\ \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

You can think about this just as a different syntax for the built-in lists.

The Haskell function $\mathit{foldr}\ (\oplus)\ e$ is a catamorphism for lists — it replaces uses of the constructor $(:)$ with $(\oplus)$ and uses of $[\,]$ with e:

$$
\begin{array}{l}
\mathit{foldr}\ (\oplus)\ e\ [\,]\ =\ e \\
\mathit{foldr}\ (\oplus)\ e\ (a : as)\ =\ a \oplus \mathit{foldr}\ (\oplus)\ e\ as
\end{array}
$$

Function $\mathit{map}\ f$ maps the function f over all elements in a list:

$$\mathit{map}\ f\ =\ \mathit{foldr}\ ((:) \circ f)\ [\,]$$

Function *map* satisfies two functor laws and function *foldr* satisfies two fusion laws: (h is strict)

$$
\begin{array}{lll}
map\ id & === & id \\
map\ f \circ map\ g & === & map\ (f \circ g) \\
h \circ foldr\ f\ e & === & foldr\ g\ (h\ e) \Leftarrow \forall\ x\ y.\ h\ (f\ x\ y) == g\ x\ (h\ y) \\
foldr\ f\ e \circ map\ g & === & foldr\ (f \circ g)\ e
\end{array}
$$

For reference we present a few other list functions here as well:

$$
\begin{array}{lll}
null & :: & [a] \to Bool \\
null\ [\,] & = & True \\
null\ (\_ : \_) & = & False \\
nil & :: & b \to [a] \\
nil\ x & = & [\,] \\
singleton & :: & a \to [a] \\
singleton\ x & = & [x] \\
(+\!\!+) & :: & [a] \to [a] \to [a] \\
xs +\!\!+ ys & = & foldr\ (:)\ ys\ xs
\end{array}
$$

2.10 Overloading and classes

We will often use Haskell's class system [66] to write generic overloaded code. This is visible in types as $context \Rightarrow normaltype$ where *context* lists the class constraints the variables in *normaltype* must satisfy. An example is $sort :: Ord\ a \Rightarrow [a] \to [a]$ where a is restricted to be in the class *Ord* of types with a comparison operator. We use the Haskell class *Monad* for monadic computations [102].

$$
\begin{array}{l}
\textbf{class}\ Monad\ m\ \textbf{where} \\
\quad \begin{array}{lll}
return & :: & a \to m\ a \\
(\gg\!=) & :: & m\ a \to (a \to m\ b) \to m\ b \\
(\gg) & :: & m\ a \to m\ b \to m\ b \\
fail & :: & String \to m\ a
\end{array}
\end{array}
$$

2.11 Fixed points

For calculations and proofs involving recursively defined values, we often use an explicit fixed point combinator to express recursive definitions. Haskell has built in support for recursive definitions over all types and we can directly define a fixed point combinator *fix*:

$$
\begin{aligned}
&\mathit{fix} :: (a \to a) \to a \\
&\mathit{fix}\ f \ = \ f\ (\mathit{fix}\ f)
\end{aligned}
$$

We call f an *improvement function* — it takes an approximation of the fixed point to a better approximation.

2.11.1 Fixed point induction

Theorem 2.11 *Fixed point fusion:*

$$f \circ g \ = \ h \circ f \ \Rightarrow f\ (\mathit{fix}\ g) == \mathit{fix}\ h$$

The requirement of the fixed point fusion law is often too strong — a weaker requirement can be obtained by observing that the equality is only needed for a chain of finite approximations of g:

$$\forall\ i.\ f\ (g\ a_i) \ = \ h\ (f\ a_i)\ \textbf{where}\ a_i \ = \ g^i\ \bot$$

This in turn can be expressed inductively:

$$
\begin{aligned}
&P\ (\bot) \wedge \forall\ x.\ P\ (x) \Rightarrow P\ (g\ x) \\
&\quad \textbf{where}\ P\ (x) \ = \ f\ (g\ x) == h\ (f\ x)
\end{aligned}
$$

The rolling rule [84] is a simple application of fixed point fusion:

Lemma 2.12 *The rolling rule: for all functions $f :: a \to b$ and $g :: b \to a$*

$$\mathit{fix}\ (f \circ g) === f\ (\mathit{fix}\ (g \circ f))$$

In the sequel we will use a powerful fixed point law that relates n fixed points. For the formulation of the fixed point law we need to introduce the concept of an *inclusive* relation as defined in Schmidt [95] (other names used in the literature are "admissible" and "limit closed").

Definition 2.13 *A relation P is* inclusive *iff for all chains of tuples $(a_1^i, ..., a_n^i)$*

$$(\forall\ i.\ P\ (a_1^i, ..., a_n^i)) \Rightarrow P\ (\bigsqcup_i a_1^i, ..., \bigsqcup_i a_n^i)$$

Inclusive relations are used to prove properties about fixed points from properties of finite approximations of these fixed points. The expression $\bigsqcup_i a_i$ denotes the *least upper bound* of the chain a_i with respect to the approximation ordering ($\sqsubseteq$) of the *CPO*. Tuples are ordered pointwise. A useful source of inclusive relations is the following theorem.

Theorem 2.14 *A class of inclusive relations: [95, def. 6.28]*

A relation P is inclusive if $P\ (f_1, ..., f_n)$ has the form:

$$\forall\ d_1 \in D_1, ..., d_m \in D_m.\ \bigwedge_{i=1}^{k} (\bigvee_{j=1}^{l}\ Q_{ij})$$

where Q_{ij} can be either

1. *A predicate using only the d_i as free identifiers.*
2. *An inclusion $e_1 \sqsubseteq e_2$ where e_1 and e_2 are expressions using continuous functions and only the f_i and the d_i as free identifiers.*

A function is continuous if it is monotone with respect to the ($\sqsubseteq$) ordering and if it preserves least upper bounds. All constructions in a functional language like Haskell are continuous, but some operators in the semantic domain are not. On *Truth*, the operators ($\wedge$) and ($\vee$) are continuous but negation ($\neg$) is not even monotone and as $a \Rightarrow b\ =\ \neg\ a\ \vee\ b$, neither is ($\Rightarrow$).

Two examples of inclusive relations are

$$\begin{array}{lcl} r_1 & :: & \mathit{Bool} \rightarrow \mathit{Truth} \\ r_1\ (b) & = & \lfloor b \rfloor \\ r_2 & :: & (\mathit{Truth}, \mathit{Truth}) \rightarrow \mathit{Truth} \\ r_2\ (a, b) & = & a \Rightarrow b \end{array}$$

Proof: Functions r_1 and r_2 are inclusive, because we can rewrite their definitions to match the form of Theorem 2.14:

$$\begin{array}{lcl} r_1\ (b) & = & \lfloor b \rfloor\ =\ (b \sqsubseteq \mathit{True})\ \wedge\ (\mathit{True} \sqsubseteq b) \\ r_2\ (a, b) & = & a \Rightarrow b\ =\ a \sqsubseteq b\ . \end{array}$$

$\square$

Theorem 2.15 *Fixed point induction: [95, def. 6.26]*

For every inclusive *relation P, and for all improvement functions $i_1, ..., i_n$:*

$$(P\ (\bot, ..., \bot)\ \wedge\ \forall\ f_1 \ldots f_n.\ P\ (f_1, ..., f_n)\ \Rightarrow\ P\ (i_1\ f_1, ..., i_n\ f_n))$$
$$\Rightarrow P\ (\mathit{fix}\ i_1, ..., \mathit{fix}\ i_n)$$

A typical example application of this theorem is found in proving that two functions $g = \mathit{fix}\ ig$ and $h = \mathit{fix}\ ih$ are equal on the set where a predicate $p = \mathit{fix}\ ip$ holds. Note that the predicate is also defined as a fixed point. We use fixed point induction with $n = 3$, the relation $P\ (x, y, z) = x \stackrel{z}{===} y$ and improvement functions ig, ih and ip.

The base case is easy: the predicate $\bot$ is never true, and all functions are trivially equal on the empty set, so what is left is the following:

Theorem 2.16 *fix-equality:*

$$(\forall\ x\ y\ z.\ x \stackrel{z}{===} y\ \Rightarrow\ ig\ x \stackrel{ip\ z}{===} ih\ y)\ \Rightarrow\ \mathit{fix}\ ig \stackrel{\mathit{fix}\ ip}{===} \mathit{fix}\ ih$$

2.11.2 Explaining fixed point induction

To prove a property of a fixed point definition using fixed point induction we have to identify a relation that implies the desired property if instantiated with the fixed points, and which holds for all approximations of the fixed point as well. The proof of such a property is similar to a proof by normal induction and consists of a series of steps. We formulate a relation P_0 to be proved, we prove the base case and we start working on the inductive case until we need a property that we cannot prove without some side condition. Assuming that the original theorem is true (and provable) it should be possible to prove the side condition together with P_0. So then we formulate a relation P_1 that implies the side condition, and strengthen the inductive hypothesis to $P = P_0 \wedge P_1$. This means extra work in proving a new base case and inductive case for P_1, but on the other hand the inductive case for P_0 makes a good leap forward. We repeat this procedure until we have an inductive proof of $P = P_0 \wedge \ldots \wedge P_n$ — this trivially implies P_0 and we are done.

If, more specifically, we want to prove that a function has a certain property when restricted to a particular set, where both the functions and the set are defined as fixed points, then the new relations to prove are of two kinds — those relating all the parameters and those restricting only the set. An example (covered in detail in Chapter 6) is proving that a rewriting function always produces a term in normal form when restricted to the set of normalizing terms. As we are interested in proving a number of properties for the same set but with different function definitions, the set-only properties can be proven separately and reused for all the proofs. This can be viewed as specializing the fixed point induction principle to an equality over a specific set, or rather specializing the inductive step to a known set improvement function i.

A very useful set-only property is

$$\begin{aligned} &\mathit{InLim} :: ((a \to \mathit{Bool}) \to (a \to \mathit{Bool})) \to (a \to \mathit{Bool}) \to (a \to \mathit{Truth}) \\ &\mathit{InLim}_i\ p \ = \ \lfloor p \rfloor \Rightarrow \lfloor \mathit{fix}\ i \rfloor \end{aligned}$$

This restricts the sets we need to consider to subsets of the fixed point (for example finite terms, or normalizing terms). Without this restriction the inductive step has to be proven for an arbitrary p and that is often hard. Fortunately, *InLim* itself is easy to prove inductively:

Lemma 2.17 *InLim:*

If $i :: (a \to \mathit{Bool}) \to (a \to \mathit{Bool})$ is an improvement function for predicates that is monotone in the following sense:

$$\forall\, p, q.\ (\lfloor p \rfloor \Rightarrow \lfloor q \rfloor) \Rightarrow (\lfloor i\ p \rfloor \Rightarrow \lfloor i\ q \rfloor)$$

then InLim_i can be used as a fixed point induction side condition:

$$\mathit{InLim}_i\ \bot \ \wedge\ (\forall\, p.\ \mathit{InLim}_i\ p \Rightarrow \mathit{InLim}_i\ (i\ p))$$

Base case: $\mathit{InLim}_i\ \bot \ = \ \lfloor \bot \rfloor \Rightarrow \lfloor \mathit{fix}\ i \rfloor \ = \ \mathit{true}.$

Inductive case: By calculation:

$$\begin{aligned} & \mathit{InLim}_i\ p \\ \equiv\quad & \{\ \text{Definition of } \mathit{InLim}\ \} \\ & \lfloor p \rfloor \Rightarrow \lfloor \mathit{fix}\ i \rfloor \\ \Rightarrow\quad & \{\ \text{Monotonicity of } i\ \} \\ & \lfloor i\ p \rfloor \Rightarrow \lfloor i\ (\mathit{fix}\ i) \rfloor \\ \equiv\quad & \{\ \text{Definition of } \mathit{fix}\text{: } \mathit{fix}\ i \ = \ i\ (\mathit{fix}\ i)\ \} \\ & \lfloor i\ p \rfloor \Rightarrow \lfloor \mathit{fix}\ i \rfloor \\ \equiv\quad & \{\ \text{Definition of } \mathit{InLim}\ \} \\ & \mathit{InLim}_i\ (i\ p) \end{aligned}$$

Chapter 3

Basic polytypic programming

The essence of funtional polytypic programming is that functions can be defined by induction on the structure of datatypes. The structure of a datatype is described by means of a *pattern functor* that captures the top level structure of elements of the datatype. Just as in imperative languages where it is preferable to use structured iteration constructs such as **while**-loops and **for**-loops instead of unstructured **goto**s, it is often advantageous to use structured recursion operators instead of unrestricted recursion when using a functional language. Structured programs are easier to reason about and more amenable to (possibly automatic) optimizations than their unstructured counterparts. Two very useful structured recursion operators are the catamorphism operator *cata* and the polytypic mapping function *pmap*. This chapter defines not only *cata* and *pmap*, but also a construct with which it is possible to define new recursion operators, tailored for specific needs. (Examples of such operators are the monadic traversal functions in Chapter 5 and the arrow maps and data conversion programs in Chapter 7.)

This chapter is organized as follows: Sections 3.1–3.3 explain the structure of two example datatypes (lists and binary trees) in terms of pattern functors. These sections also introduce catamorphisms, maps and fusion laws for the example datatypes, and use fusion to prove a few laws in a calculational style. Section 3.4 defines regular datatypes and shows how pattern functors are used to capture the structure of regular datatypes. Section 3.5 defines the isomorphisms *inn* and *out* that convert values between a regular datatype and the top level structure of that datatype. Section 3.6 introduces the **polytypic** construct to express polytypic functions by induction over pattern functors. The definition of the polytypic sum function *psum* from the introduction is used as an example. Section 3.7 defines polytypic catamorphisms and maps and Section 3.8 explains how use a catamorphism as an evaluator for a small expression language. Section 3.9 presents a self-contained polytypic program, together with the code that the PolyP generates for that program. Finally, Section 3.10 states and proves some polytypic function laws.

3.1 The structure of lists

Consider the datatype $List\ a$ that is defined by

$$\textbf{data}\ List\ a\ =\ Nil \mid Cons\ a\ (List\ a)\ .$$

This datatype can be viewed as the fixed point with respect to the second argument of the datatype $FList\ a\ r$ defined by

$$\textbf{data}\ FList\ a\ r\ =\ FNil \mid FCons\ a\ r\ .$$

The datatype $FList\ a\ r$ describes the structure of the datatype $List\ a$. Note that $FList$ has one argument more than $List$. The extra argument is used to represent the recursive occurrence of the datatype $List\ a$ in the right-hand side of its definition. Because we are only interested in the structure of $List\ a$, the names of the constructors of $FList$ are not important. As an element of $FList$ is either a nullary constructor or a binary constructor with its two arguments, we can instead represent the type $FList$ by:

$$\textbf{type}\ FList\ a\ r\ =\ Either\ ()\ (a, r)$$

We call $FList$ a *pattern functor* as it captures the recursion pattern of a datatype.

We now abstract from the arguments a and r to obtain a variable free description of $FList$. We represent the first argument by the pattern functor Par and the second argument by Rec.

$$\begin{aligned}
&\textbf{type}\ Par\ a\ r\ =\ a\\
&\textbf{type}\ Rec\ a\ r\ =\ r
\end{aligned}$$

The type constructors in $FList$ are lifted to work on pattern functors: $Either$ is lifted to $+$, the pair type constructor $(\_,\_)$ is lifted to $*$ and the unit type $()$ is lifted to $Empty$.

$$\begin{aligned}
\textbf{type}\ (f+g)\ a\ r\ &=\ Either\ (f\ \ a\ r)\ (g\ \ a\ r)\\
\textbf{type}\ (f*g)\ a\ r\ &=\ (f\ \ a\ r, g\ \ a\ r)\\
\textbf{type}\ Empty\ a\ r\ &=\ ()
\end{aligned}$$

As usual, $*$ binds stronger than $+$. Using these pattern functor constructors we can express $FList$ in a variable free form.

$$FList\ =\ Empty + Par * Rec$$

The initial object in the category of *FList* a-algebras (that is, the fixed point of *FList* with respect to its second component) models the datatype *List* a. The initial object consists of two parts: the datatype *List* a, and a single strict constructor function inn_{List}, that combines the constructors *Nil* and *Cons*.

$$\begin{array}{l} inn_{List} :: FList\ a\ (List\ a) \to List\ a \\ inn_{List}\ =\ const\ Nil\ \triangledown\ uncurry\ Cons \end{array}$$

As an example, the list containing only the integer 3, *Cons* 3 *Nil*, is represented by $inn_{List}\ (Right\ (3, inn_{List}\ (Left\ ())))$. Function out_{List} is the inverse of function inn_{List}.

$$\begin{array}{lcl} out_{List} :: List\ a & \to & FList\ a\ (List\ a) \\ out_{List}\ Nil & = & Left\ () \\ out_{List}\ (Cons\ a\ b) & = & Right\ (a, b) \end{array}$$

In the polytypic programming system PolyP these functions are automatically supplied by the system for each user-defined datatype.

The pattern functor *FList* takes two types and returns a type. *FList* is a bifunctor, which is witnessed by the existence of a corresponding action, called $fmap2_{FList}$, on functions. Function $fmap2_{FList}$ takes two functions and returns a function.

$$\begin{array}{l} fmap2_{FList} :: (a \to c) \to (b \to d) \to (FList\ a\ b \to FList\ c\ d) \\ fmap2_{FList}\ p\ r\ =\ id \mathbin{-\!\!+} p \mathbin{-\!\!*} r \end{array}$$

That $fmap2_{FList}$ is indeed a bifunctor follows immediately from the corresponding laws for $(\ \mathbin{-\!\!+}\ )$ and $(\ \mathbin{-\!\!*}\ )$.

$$\begin{array}{lcl} fmap2_{FList}\ id\ id & === & id \\ fmap2_{FList}\ f\ g \circ fmap2_{FList}\ h\ i & === & fmap2_{FList}\ (f \circ h)\ (g \circ i) \end{array}$$

As an example of a program written using the combinators defined so far we show $map_{List}\ f\ xs$ that applies function f to all elements of the list xs:

$$\begin{array}{lcl} map_{List} & :: & (a \to b) \to (List\ a \to List\ b) \\ map_{List}\ f & = & inn_{List} \circ fmap2_{FList}\ f\ (map_{List}\ f) \circ out_{List} \end{array}$$

Function map_{List} is really the same function as *map* in Haskell but we define it differently here to allow for a simple generalization to the polytypic case.

Just as *FList* and $fmap2_{FList}$ form a functor, so do *List* and the function map_{List}:

$$\begin{array}{lcl} map_{List}\ id & === & id \\ map_{List}\ f \circ map_{List}\ g & === & map_{List}\ (f \circ g) \end{array}$$

3.2 Catamorphisms and fusion for lists

Function $size_{List}$ returns the number of elements in a $List\ a$ (corresponding to the function *length* in Haskell). The result of applying $size_{List}$ to an argument list can be computed by replacing uses of the constructor *Nil* by 0, and uses of the constructor *Cons* by 1+.

$$\begin{array}{llll} Cons\ 17 & (Cons\ 3 & (Cons\ 8 & Nil\ )) \\ 1+ & (1+ & (1+ & 0\ \ )) \end{array}$$

Thus the size of this list is 3. We use a higher-order function to describe functions that replace constructors by functions: the catamorphism. The catamorphism is a basic structured recursion operator and on lists it is equivalent to function *foldr* in Haskell:

$$\begin{array}{l} foldr\ f\ e\ =\ cata_{List}\ \phi \\ \quad \textbf{where}\ \phi :: FList\ a\ b \to b \\ \qquad\quad \phi\ =\ const\ e \triangledown uncurry\ f \end{array}$$

The catamorphism $cata_{List}\ \phi$ replaces *Nil* by e, and *Cons* by f.

$$\begin{array}{llllllll} Cons & 17 & (Cons & 3 & (Cons & 8 & Nil &)) \\ f & 17 & (f & 3 & (f & 8 & e &)) \end{array}$$

Function $cata_{List}$ is defined using function out_{List} to avoid a definition by pattern matching. Function $fmap2_{FList}\ id\ (cata_{List}\ f)$ applies $cata_{List}\ f$ recursively to the rest of the list.

$$\begin{array}{l} cata_{List} :: (FList\ a\ b \to b) \to List\ a \to b \\ cata_{List}\ f\ =\ f \circ fmap2_{FList}\ id\ (cata_{List}\ f) \circ out_{List} \end{array}$$

The theoretical justification for this definition is that in the category of $FList\ a$-algebras the $FList\ a$-algebra $(List\ a, inn_{List})$ is an initial object. This means that there is a unique arrow from $(List\ a, inn_{List})$ to every $FList\ a$-algebra (b, f). This unique arrow is the function $cata_{List}\ f$. The initiality of this algebra also means that $cata_{List}\ inn_{List}$ is the identity function on $List\ a$.

As examples we use function $cata_{List}$ to define the function $size_{List}$ (corresponding to $length :: [a] \to Int$ in Haskell) and list concatenation $(+\!\!+)$.

$$\begin{array}{lcl} size_{List} & :: & List\ a \to Int \\ size_{List} & = & cata_{List}\ (const\ 0 \triangledown inc) \\ & & \quad \textbf{where}\ inc\ (\_, n)\ =\ 1 + n \\ (+\!\!+) & :: & List\ a \to List\ a \to List\ a \\ xs +\!\!+ ys & = & cata_{List}\ (const\ ys \triangledown uncurry\ Cons)\ xs \end{array}$$

Function $cata_{List}$ satisfies the so-called *fusion law*. The fusion law gives conditions under which intermediate values produced by a catamorphism can be eliminated.

Law 3.1 *List-fusion: for strict h,*

$$h \circ cata_{List}\ f \ = \ cata_{List}\ g \quad \Leftarrow \quad h \circ f \ = \ g \circ fmap2_{FList}\ id\ h \ .$$

Using *List*-fusion we can prove a lemma relating $size_{List}$ and $(+\!\!+)$.

Lemma 3.2 *The $size_{List}$-$(+\!\!+)$-lemma:*

$$size_{List}\ (xs +\!\!+ ys) \ = \ size_{List}\ xs + size_{List}\ ys \ .$$

Proof: In the calculations we abbreviate $size_{List}$ with #.

$$
\begin{array}{ll}
 & \#\ (xs +\!\!+ ys) \ = \ \#\ xs + \#\ ys \\
\Leftarrow & \quad \{\text{ Abstract from } xs \ \} \\
 & \# \circ (+\!\!+ ys) \ = \ (+(\#\ ys)) \circ \# \\
\Leftarrow & \quad \{\text{ Assume both sides can be written as a catamorphism } \} \\
 & \# \circ (+\!\!+ ys) \ = \ cata_{List}\ (n \triangledown c) \ = \ (+(\#\ ys)) \circ \# \\
\Leftarrow & \quad \{\text{ Two subcalculations using } List\text{-fusion } \} \\
 & True
\end{array}
$$

In the first subcalculation we fuse # with $(+\!\!+ ys)$.

$$
\begin{array}{ll}
 & \# \circ (+\!\!+ ys) \ = \ cata_{List}\ (n \triangledown c) \\
\equiv & \quad \{\text{ Definition of } (+\!\!+ ys) \ \} \\
 & \# \circ cata_{List}\ (const\ ys \triangledown uncurry\ Cons) \ = \ cata_{List}\ (n \triangledown c) \\
\Leftarrow & \quad \{\text{ Fusion } \} \\
 & \# \circ (const\ ys \triangledown uncurry\ Cons) \ = \ (n \triangledown c) \circ fmap2_{FList}\ id\ \# \\
\equiv & \quad \{\text{ Definition of } fmap2_{FList} \ \} \\
 & \# \circ (const\ ys \triangledown uncurry\ Cons) \ = \ (n \triangledown c) \circ (id + (id \ast \#))
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{\text{Laws for } (\triangledown)\} \\
& (\# \circ \mathit{const}\ ys) \triangledown (\# \circ \mathit{uncurry}\ \mathit{Cons}) = n \triangledown (c \circ (\mathit{id} \ast \#)) \\
\equiv & \{\text{Split the } (\triangledown)\text{s and simplify}\} \\
& \# \circ \mathit{const}\ ys = n \wedge \# \circ \mathit{uncurry}\ \mathit{Cons} = c \circ (\mathit{id} \ast \#) \\
\equiv & \{\text{Introduce arguments: () and } (x, n)\} \\
& \#\, ys = n\ () \wedge \#\, (\mathit{Cons}\ x\ xs) = c\ (x, \#\, xs) \\
\equiv & \{\text{Let } n = \mathit{const}\ (\#\, ys)\} \\
& \mathit{True} \wedge 1 + \#\, xs = c\ (x, \#\, xs) \\
\equiv & \{\text{Let } c = \mathit{inc}\} \\
& \mathit{True}
\end{array}
$$

In the second subcalculation we let $m = \#\, ys$ and we fuse $(+m)$ with $\#$.

$$
\begin{array}{ll}
& (+m) \circ \# = \mathit{cata}_{\mathit{List}}\ (n \triangledown \mathit{inc}) \\
\equiv & \{\text{Definition of } \#\} \\
& (+m) \circ \mathit{cata}_{\mathit{List}}\ (\mathit{const}\ 0 \triangledown \mathit{inc}) = \mathit{cata}_{\mathit{List}}\ (n \triangledown \mathit{inc}) \\
\equiv & \{\text{Fusion}\} \\
& (+m) \circ (\mathit{const}\ 0 \triangledown \mathit{inc}) = (n \triangledown \mathit{inc}) \circ \mathit{fmap2}_{\mathit{FList}}\ \mathit{id}\ (+m) \\
\equiv & \{\text{Definition of } \mathit{fmap2}_{\mathit{FList}}\} \\
& (+m) \circ (\mathit{const}\ 0 \triangledown \mathit{inc}) = (n \triangledown \mathit{inc}) \circ (\mathit{id} + (\mathit{id} \ast (+m))) \\
\equiv & \{\text{Laws for } (\triangledown)\} \\
& ((+m) \circ \mathit{const}\ 0) \triangledown ((+m) \circ \mathit{inc}) = n \triangledown (\mathit{inc} \circ (\mathit{id} \ast (+m))) \\
\equiv & \{\text{Split the } (\triangledown)\text{s and simplify}\} \\
& (+m) \circ \mathit{const}\ 0 = n \wedge (+m) \circ \mathit{inc} = \mathit{inc} \circ (\mathit{id} \ast (+m)) \\
\equiv & \{\text{Introduce arguments: () and } (x, n)\} \\
& m = n\ () \wedge (\mathit{inc}\ (x, n)) + m = \mathit{inc}\ (x, n + m) \\
\equiv & \{\text{Definitions of } n \text{ and } \mathit{inc}\} \\
& m = m \wedge 1 + n + m = 1 + n + m \\
\equiv & \{\text{Trivially}\} \\
& \mathit{True}
\end{array}
$$

□

3.3 The structure of trees

The datatype *Tree a* is defined by

$$\mathbf{data}\ \mathit{Tree}\ a \ = \ \mathit{Leaf}\ a \mid \mathit{Bin}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

Applying the same procedure as for the datatype *List a*, we obtain the following functor that describes the structure of the datatype *Tree a*.

$$\mathit{FTree} \ = \ \mathit{Par} + \mathit{Rec} * \mathit{Rec}$$

Functions $\mathit{inn}_{\mathit{Tree}}$ and $\mathit{out}_{\mathit{Tree}}$ are defined in the same way as functions $\mathit{inn}_{\mathit{List}}$ and $\mathit{out}_{\mathit{List}}$.

$$\begin{aligned}
&\mathit{inn}_{\mathit{Tree}} :: \mathit{FTree}\ a\ (\mathit{Tree}\ a) \to \mathit{Tree}\ a \\
&\mathit{inn}_{\mathit{Tree}} \ = \ \mathit{Leaf} \triangledown \mathit{uncurry}\ \mathit{Bin}
\end{aligned}$$

$$\begin{aligned}
\mathit{out}_{\mathit{Tree}} :: \mathit{Tree}\ a \quad &\to \quad \mathit{FTree}\ a\ (\mathit{Tree}\ a) \\
\mathit{out}_{\mathit{Tree}}\ (\mathit{Leaf}\ a) \quad &= \quad \mathit{Left}\ a \\
\mathit{out}_{\mathit{Tree}}\ (\mathit{Bin}\ a\ b) \quad &= \quad \mathit{Right}\ (a, b)
\end{aligned}$$

The functions $\mathit{map}_{\mathit{Tree}}$ and $\mathit{cata}_{\mathit{Tree}}$ are defined in terms of functions $\mathit{inn}_{\mathit{Tree}}$, $\mathit{out}_{\mathit{Tree}}$ and $\mathit{fmap2}_{\mathit{FTree}}$:

$$\begin{aligned}
&\mathit{fmap2}_{\mathit{FTree}} :: (a \to c) \to (b \to d) \to (\mathit{FTree}\ a\ b \to \mathit{FTree}\ c\ d) \\
&\mathit{fmap2}_{\mathit{FTree}}\ p\ r \ = \ p \mathbin{-\!\!+\!\!-} r \mathbin{-\!\!*\!\!-} r
\end{aligned}$$

$$\begin{aligned}
&\mathit{map}_{\mathit{Tree}} :: (a \to b) \to (\mathit{Tree}\ a \to \mathit{Tree}\ b) \\
&\mathit{map}_{\mathit{Tree}}\ f \ = \ \mathit{inn}_{\mathit{Tree}} \circ \mathit{fmap2}_{\mathit{FTree}}\ f\ (\mathit{map}_{\mathit{Tree}}\ f) \circ \mathit{out}_{\mathit{Tree}}
\end{aligned}$$

$$\begin{aligned}
&\mathit{cata}_{\mathit{Tree}} :: (\mathit{FTree}\ a\ b \to b) \to (\mathit{Tree}\ a \to b) \\
&\mathit{cata}_{\mathit{Tree}}\ f \ = \ f \circ \mathit{fmap2}_{\mathit{FTree}}\ \mathit{id}\ (\mathit{cata}_{\mathit{Tree}}\ f) \circ \mathit{out}_{\mathit{Tree}}
\end{aligned}$$

Note that the definitions of $\mathit{map}_{\mathit{Tree}}$ and $\mathit{cata}_{\mathit{Tree}}$ are almost identical to the definitions $\mathit{map}_{\mathit{List}}$ and $\mathit{cata}_{\mathit{List}}$, only the indices are different. Function $\mathit{size}_{\mathit{Tree}}$ is defined by

$$\begin{aligned}
\mathit{size}_{\mathit{Tree}} \quad &:: \quad \mathit{Tree}\ a \to \mathit{Int} \\
\mathit{size}_{\mathit{Tree}} \quad &= \quad \mathit{cata}_{\mathit{Tree}}\ (\mathit{const}\ 1 \triangledown \mathit{uncurry}\ (+))
\end{aligned}$$

The function $\mathit{flatten}_{\mathit{Tree}}$, which returns a list containing the elements of the argument tree, can also be defined using function $\mathit{cata}_{\mathit{Tree}}$:

$$\begin{aligned}
\mathit{flatten}_{\mathit{Tree}} \quad &:: \quad \mathit{Tree}\ a \to [\,a\,] \\
\mathit{flatten}_{\mathit{Tree}} \quad &= \quad \mathit{cata}_{\mathit{Tree}}\ (\mathit{singleton} \triangledown \mathit{uncurry}\ (+\!\!+))
\end{aligned}$$

The fusion law for trees looks the same as the fusion law for lists:

Law 3.3 *Tree-fusion: for strict h,*

$$h \circ cata_{Tree}\ f \;=\; cata_{Tree}\ g \quad\Leftarrow\quad h \circ f \;=\; g \circ fmap2_{FTree}\ id\ h\ .$$

We can use this law to prove that $size_{List} \circ flatten_{Tree} \;=\; size_{Tree}$.

$$\begin{array}{cl}
& \# \circ flatten_{Tree} \;=\; size_{Tree} \\
\equiv & \{\text{By definition, introducing the abbreviations } \phi \text{ and } \sigma\} \\
& \# \circ cata_{Tree}\ \phi \;=\; cata_{Tree}\ \sigma \\
\Leftarrow & \{\text{Fusion}\} \\
& \# \circ \phi \;=\; \sigma \circ (fmap2_{FTree}\ id\ \#) \\
\equiv & \{\text{By definition of } fmap2_{FTree}\} \\
& \# \circ \phi \;=\; \sigma \circ (id + (\# \ast \#)) \\
\equiv & \{\text{New abbreviations: } \sigma \;=\; \sigma_1 \triangledown \sigma_2 \text{ and } \phi \;=\; \phi_1 \triangledown \phi_2\} \\
& \# \circ (\phi_1 \triangledown \phi_2) \;=\; (\sigma_1 \triangledown \sigma_2) \circ (id + (\# \ast \#)) \\
\equiv & \{\text{Laws for } (\,\triangledown\,)\} \\
& (\# \circ \phi_1) \triangledown (\# \circ \phi_2) \;=\; (\sigma_1 \circ id) \triangledown (\sigma_2 \circ (\# \ast \#)) \\
\equiv & \{\text{Split the } (\,\triangledown\,)\text{s and simplify}\} \\
& \# \circ \phi_1 \;=\; \sigma_1 \;\wedge\; \# \circ \phi_2 \;=\; \sigma_2 \circ (\# \ast \#) \\
\equiv & \{\text{Introduce arguments, implicitly } \forall\text{-quantified}\} \\
& \#\,(\phi_1\ x) \;=\; \sigma_1\ x \;\wedge\; \#\,(\phi_2\ (l, l')) \;=\; \sigma_2\ (\#\,l, \#\,l') \\
\equiv & \{\text{Definition of } \phi_i \text{ and } \sigma_i\} \\
& \#\,[x] \;=\; 1 \;\wedge\; \#\,(l \mathbin{+\!\!\!+} l') \;=\; \#\,l + \#\,l' \\
\equiv & \{\text{Lemma 3.2}\} \\
& True \;\wedge\; True
\end{array}$$

3.4 Pattern functors

A pattern functor captures the (top level) structure of a datatype. We represent a pattern functor in a variable free form by means of a number of functor constructors. We have already introduced *Par* for the datatype parameter, *Rec* for the recursive parameter, *Empty* for the empty product and $(+)$ and $(*)$ for lifted versions of *Either* and $(,)$ and we have used them to define the pattern functors for lists and trees. In general, PolyP's pattern functors are generated by the following grammar:

$$f, g, h \ ::= \ g + h \mid g * h \mid \mathit{Empty} \mid \mathit{Par} \mid \mathit{Rec} \mid d@g \mid \mathit{Const}\ t$$

where d generates regular datatype constructors, and t generates monomorphic types. We note the following about the functor constructors:

- The pattern functor for a datatype with more than two constructors is represented by a nested binary sum associating to the right. Therefore, in the concrete syntax, the constructor $+$ is right-associative, so that $f+g+h$ means $f+(g+h)$. Constructor $+$ may only occur at top level, so $f*(g+h)$ is an illegal functor. This restriction corresponds to the syntactic restriction in Haskell which says that the vertical bar | that separates constructors may only occur at the top level of datatype definitions.

- Constructor $*$ is right-associative and binds stronger than $+$.

- The constructor *Empty* is the empty or nullary product.

- Composition of functors d and g is denoted by $d@g$ and is only defined for a unary functor d and a binary functor g. Functor composition is used to describe the structure of types that are defined in terms of other user-defined datatypes, such as the datatype of *rose-trees*:

  $$\begin{aligned} \textbf{data}\ \mathit{Rose}\ a \ &= \ \mathit{Fork}\ a\ (\mathit{List}\ (\mathit{Rose}\ a)) \\ \text{-- } \mathit{FRose} \ &= \ \mathit{Par} * (\mathit{List}@\mathit{Rec}) \end{aligned}$$

- The pattern functor *Const* t denotes a constant pattern functor with value t. The t stands for a monotype such as *Bool*, *Char* or $(\mathit{Int}, [\mathit{Float}])$. This is used when a datatype definition mentions a type other than the type parameter and datatype itself. An example is the structure of the following simple datatype of types:

  $$\begin{aligned} \textbf{data}\ \mathit{Type}\ a \ &= \ \mathit{Con}\ \mathit{String} \mid \mathit{Var}\ a \mid \mathit{Fun}\ (\mathit{Type}\ a)\ (\mathit{Type}\ a) \\ \text{-- } \mathit{FType} \ &= \ \mathit{Const}\ \mathit{String} + \mathit{Par} + \mathit{Rec} * \mathit{Rec} \end{aligned}$$

$$
\begin{array}{lllll}
\Phi_{List} & = & FList & = & Empty + Par * Rec \\
\Phi_{Tree} & = & FTree & = & Par + Rec * Rec \\
\Phi_{Rose} & = & FRose & = & Par * (List@Rec) \\
\Phi_{Type} & = & FType & = & Const\ String + Par + Rec * Rec
\end{array}
$$

Figure 3.1: Examples of pattern functors.

The type context $Bifunctor\ f\ \ \Rightarrow$ is used to indicate that f is a pattern functor.

Every regular recursive datatype $d\ a$ in Haskell is implicitly defined as a fixed point of a pattern functor $\Phi_d\ a$, that is $d\ a \cong \mu(\Phi_d\ a)$. PolyP provides a type constructor *FunctorOf* d (we use Φ_d as a shorthand) for this pattern functor. Pattern functors for the types defined in this chapter are summarized in Figure 3.1. A datatype $d\ a$ is regular (satisfies $Regular\ d$) if it contains no function spaces, and if the argument of the type constructor d is the same on the left- and right-hand side of its definition. For each regular datatype $d\ a$, PolyP automatically generates Φ_d using roughly the same steps as those used manually for *FList* and *FTree* in previous sections. Pattern functors are *only* constructed for datatypes defined by means of the **data** construct. If, somewhere in a program, a polytypic function is applied to a value of type $Maybe\ (List\ a)$, then PolyP will generate an instance of the polytypic function on the datatype $Maybe\ b$ (with $b\ =\ List\ a$), not on the type $(Maybe@List)\ a$.

A regular datatype is defined as the fixed point of a pattern functor. The pattern functor Φ_d may, in turn, refer to other (previously defined) regular datatypes in the $d@g$ case. Thus the descriptions of regular datatypes and pattern functors are mutually recursive. In practice, this means that most polytypic definitions are given as two mutually recursive bindings — one for the datatype level and one for the pattern functor level. Similarly, laws for polytypic functions are often proved by mutual induction over the grammars for regular datatypes and pattern functors. This induction is well-founded as we don't allow mutually recursive datatypes and thus a datatype can only refer to a datatype that is defined earlier.

In the rest of the paper we always assume that $d\ a$ is a regular datatype and that f is a pattern functor, and we often omit the contexts ($Regular\ d\ \ \Rightarrow$ or $Bifunctor\ f\ \ \Rightarrow$) from the types for brevity. This is purely a notational convention in the dissertation, explicit types in actual PolyP programs must contain the proper context.

3.5 In and out of a regular datatype

In the definition of a function that works for an arbitrary (as yet unknown) datatype we cannot use the constructors to build values, nor pattern match against values. Instead, we use two built-in functions, *inn* and *out*, to construct and destruct a value of an arbitrary datatype from and to its top level components. Functions *inn* and *out* are the fold and unfold isomorphisms showing $d\ a \cong \Phi_d\ a\ (d\ a)$.

$$\begin{aligned}
&inn :: Regular\ d \Rightarrow \Phi_d\ a\ (d\ a) \to d\ a\\
&out :: Regular\ d \Rightarrow d\ a \to \Phi_d\ a\ (d\ a)
\end{aligned}$$

Theorem 3.4 *Functions inn and out are inverses.*

For every Regular datatype d a:

$$\begin{aligned}
&inn \circ out === id :: d\ a \to d\ a\\
&out \circ inn === id :: \Phi_d\ a\ b \to \Phi_d\ a\ b
\end{aligned}$$

Note that functions *inn* and *out* are only defined for *Regular* datatypes $d\ a$. PolyP automatically generates instances of *inn* and *out* for all regular datatypes. Example instances were given in Sections 3.1 and 3.3.

3.6 The polytypic construct

PolyP introduces a new construct **polytypic** for defining polytypic functions by induction over pattern functors:

$$\textbf{polytypic}\ p :: [\mathit{Bifunctor}\ f \Rightarrow]\ t\ =\ [\lambda x_1 \ldots x_n \to]\ \textbf{case}\ f\ \textbf{of}\ \{f_i \to e_i\}$$

Here p is the name of the value being defined, t is its type, f is a functor variable, f_i are functor patterns and e_i are PolyP expressions. The optional function abstraction $\lambda x_1 \ldots x_n \to$ is syntactic sugar for a **polytypic** definition with this abstraction in each of the branches e_i. The explicit type in the **polytypic** construct is needed because we cannot in general infer the type from the cases. As the case analysis is over pattern functors, f must be restricted by the context *Bifunctor* $f \Rightarrow$, but it is optional in the syntax.

The informal meaning is that we define a function that takes (a representation of) a pattern functor as its first argument. This function selects the expression in the first branch of the case matching the functor, and the expression may in turn use the polytypic function (on subfunctors). Thus the polytypic construct is a (recursive) template for constructing instances of polytypic functions given the pattern functor of a datatype. The functor argument of the polytypic function need not (and cannot) be supplied explicitly but is inserted by the compiler during type inference.

$$
\begin{array}{l}
psum :: Regular\ d \Rightarrow d\ Int \rightarrow Int \\
psum\ =\ cata\ fsum
\end{array}
$$

$$
\begin{array}{llll}
\textbf{polytypic}\ fsum & :: & f\ Int\ Int \rightarrow Int \\
\quad =\ \textbf{case}\ f\ \textbf{of} & & \\
\qquad g + h & \rightarrow & fsum\ \triangledown\ fsum \\
\qquad g * h & \rightarrow & \lambda(x, y) \rightarrow fsum\ x + fsum\ y \\
\qquad Empty & \rightarrow & \lambda x \rightarrow 0 \\
\qquad Par & \rightarrow & id \\
\qquad Rec & \rightarrow & id \\
\qquad d@g & \rightarrow & psum \circ pmap\ fsum \\
\qquad Const\ t & \rightarrow & \lambda x \rightarrow 0
\end{array}
$$

Figure 3.2: The definition of *psum*

As an example we take the polytypic sum function discussed already in the introduction. Function *psum* (defined in Figure 3.2) sums the integers in a structure with integers. The definitions of *cata* and *pmap* are given later in Section 3.7. When *psum* is used on an element of type *Tree Int*, the compiler produces the code in Figure 3.3 for $psum_{Tree}$ and $fsum_{FTree}$. Together with the code generated

$$
\begin{array}{lll}
psum_{Tree} & :: & Tree\ Int \rightarrow Int \\
psum_{Tree} & = & cata_{Tree}\ fsum_{FTree} \\
fsum_{FTree} & :: & Either\ Int\ (Int, Int) \rightarrow Int \\
fsum_{FTree} & = & fsum_{Par}\ \triangledown\ fsum_{Rec*Rec} \\
fsum_{Par} & :: & Int \rightarrow Int \\
fsum_{Par} & = & id \\
fsum_{Rec*Rec} & :: & (Int, Int) \rightarrow Int \\
fsum_{Rec*Rec} & = & \lambda(x, y) \rightarrow fsum_{Rec}\ x + fsum_{Rec}\ y \\
fsum_{Rec} & :: & Int \rightarrow Int \\
fsum_{Rec} & = & id
\end{array}
$$

Figure 3.3: Generated Haskell code for $psum_{Tree}$ and $fsum_{FTree}$

for $cata_{Tree}$ (presented later in Figure 3.5), this is a complete definition of the instance $psum_{Tree}$. Function $fsum_{Rec*Rec}$ can be rewritten as *uncurry* (+) and, if we inline all the instances of *fsum*, then we obtain the following function for summing a tree:

$$\begin{aligned} &psum_{Tree} :: Tree\ Int \to Int \\ &psum_{Tree} \ = \ cata_{Tree}\ (id\ \triangledown\ uncurry\ (+)) \end{aligned}$$

As expected, $psum_{Tree}$ is a *Tree*-catamorphism that replaces the constructor *Leaf* with *id* and the constructor *Bin* with $(+)$.

3.7 Catamorphisms and maps

This section defines the functions *cata* and *pmap* that were used in the definition of function *psum* in Figure 3.2.

The catamorphism, or generalized fold, on a datatype takes as many functions as the datatype has constructors (combined into a single argument by means of function $(\ \triangledown\ )$), and recursively replaces constructor functions with corresponding argument functions. It is a generalization to arbitrary regular datatypes of the function *foldr* that is defined on lists. In spite of its generality, function *cata* can be defined in just one line in terms of the functor map, *fmap2* (defined later in Figure 3.4):

$$\begin{aligned} &cata :: Regular\ d \Rightarrow (\Phi_d\ a\ b \to b) \to (d\ a \to b) \\ &cata\ f \ = \ f \circ fmap2\ id\ (cata\ f) \circ out \end{aligned}$$

Function *out* makes the top level structure of the input explicit, *fmap2* applies $(cata\ f)$ recursively to the immediate substructures, and f combines the results of the recursive calls into the final result. Except for the indices, the definition of the polytypic *cata* is the same as the instances on $List\ a$ and $Tree\ a$. Similarly, we can define a polytypic version of *map*:

$$\begin{aligned} &pmap :: Regular\ d \Rightarrow (a \to b) \to (d\ a \to d\ b) \\ &pmap\ p \ = \ inn \circ fmap2\ p\ (pmap\ p) \circ out \end{aligned}$$

Function $pmap\ p$ applies function p to all elements of type a in a value of type $d\ a$. Function *out* takes the argument apart, *fmap2* applies f to parameters and $(pmap\ f)$ recursively to substructures and *inn* puts the parts back together again. We call it *pmap* to avoid a name clash with the normal Haskell function *map*. The types of *cata* and *pmap* are best explained by commuting diagrams:

$$\begin{tikzcd} d\ a \arrow[r, "out"] \arrow[d, "cata\ f"'] & \Phi_d\ a\ (d\ a) \arrow[d, "fmap2\ id\ (cata\ f)"] \\ b & \Phi_d\ a\ b \arrow[l, "f"] \end{tikzcd} \qquad \begin{tikzcd} d\ a \arrow[r, "out"] \arrow[d, "pmap\ f"'] & \Phi_d\ a\ (d\ a) \arrow[d, "fmap2\ f\ (pmap\ f)"] \\ d\ b & \Phi_d\ b\ (d\ b) \arrow[l, "inn"] \end{tikzcd}$$

As explained in the prelude, a functor is a mapping between categories that preserves the algebraic structure of the category. As a category consists of objects (types) and arrows (functions), a functor consists of two parts: a definition on types, and a definition on functions. A pattern functor in PolyP is a function that takes two types and returns a type. The part of the functor that takes two functions and returns a function is called *fmap2*, see Figure 3.4.

$$
\begin{array}{l}
\textbf{polytypic}\ \mathit{fmap2} :: (a \to c) \to (b \to d) \to (f\ a\ b \to f\ c\ d) \\
\quad = \lambda p\ r \to \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad \begin{array}{lcl}
g + h & \to & \mathit{fmap2}\ p\ r \mathbin{-\!\!+\!\!-} \mathit{fmap2}\ p\ r \\
g * h & \to & \mathit{fmap2}\ p\ r \mathbin{-\!\!*\!\!-} \mathit{fmap2}\ p\ r \\
\mathit{Empty} & \to & \mathit{id} \\
\mathit{Par} & \to & p \\
\mathit{Rec} & \to & r \\
d @ g & \to & \mathit{pmap}\ (\mathit{fmap2}\ p\ r) \\
\mathit{Const}\ t & \to & \mathit{id}
\end{array}
\end{array}
$$

Figure 3.4: The definition of *fmap2*.

Function $\mathit{fmap2}_g$ is the function action of the pattern functor g, and we can show that pmap_d is the function action of the type constructor d, viewed as a functor. As an example of an instance, Figure 3.5 presents the code generated by PolyP for $\mathit{cata}_{\mathit{Tree}}$. Function $\mathit{out}_{\mathit{Tree}}$ was defined in Section 3.3.

$$
\begin{array}{lcl}
\mathit{cata}_{\mathit{Tree}} & :: & (\mathit{Either}\ a\ (b, b) \to b) \to \mathit{Tree}\ a \to b \\
\mathit{cata}_{\mathit{Tree}}\ i & = & i \circ \mathit{fmap2}_{\mathit{FTree}}\ \mathit{id}\ (\mathit{cata}_{\mathit{Tree}}\ i) \circ \mathit{out}_{\mathit{Tree}} \\
\mathit{fmap2}_{\mathit{FTree}} & :: & (a \to b) \to (c \to d) \to \mathit{Either}\ a\ (c, c) \to \mathit{Either}\ b\ (d, d) \\
\mathit{fmap2}_{\mathit{FTree}} & = & \lambda p\ r \to \mathit{fmap2}_{\mathit{Par}}\ p\ r \mathbin{-\!\!+\!\!-} \mathit{fmap2}_{\mathit{Rec*Rec}}\ p\ r \\
\mathit{fmap2}_{\mathit{Par}} & :: & (a \to b) \to (c \to d) \to a \to b \\
\mathit{fmap2}_{\mathit{Par}} & = & \lambda p\ r \to p \\
\mathit{fmap2}_{\mathit{Rec*Rec}} & :: & (a \to b) \to (c \to d) \to (c, c) \to (d, d) \\
\mathit{fmap2}_{\mathit{Rec*Rec}} & = & \lambda p\ r \to \mathit{fmap2}_{\mathit{Rec}}\ p\ r \mathbin{-\!\!*\!\!-} \mathit{fmap2}_{\mathit{Rec}}\ p\ r \\
\mathit{fmap2}_{\mathit{Rec}} & :: & (a \to b) \to (c \to d) \to c \to d \\
\mathit{fmap2}_{\mathit{Rec}} & = & \lambda p\ r \to r
\end{array}
$$

Figure 3.5: Generated code for $\mathit{cata}_{\mathit{Tree}}$ and $\mathit{fmap2}_{\mathit{FTree}}$

Function *fmap2* and function *pmap* are mutually recursive through the $d@g$ case. This recursive dependence is only in the code generation phase. Take the instance

$pmap_{Rose}$ as an example: The generated instances of $pmap$ and $fmap2$ are shown in Figures 3.6 and 3.7 respectively (except for functions $fmap2_{Par}$, $fmap2_{Rec}$, inn_{List} and out_{List} which have been defined already). Function $pmap_{Rose}$ uses $fmap2_{FRose}$ and $fmap2_{FRose}$ uses $pmap_{List}$. We see that the instances are not mutually recursive as $pmap$ is instantiated on different types.

$$
\begin{array}{lcl}
pmap_{Rose} & :: & (a \to b) \to Rose\ a \to Rose\ b \\
pmap_{Rose}\ f & = & inn_{Rose} \circ fmap2_{FRose}\ f\ (pmap_{Rose}\ f) \circ out_{Rose} \\
fmap2_{FRose} & :: & (a \to b) \to (c \to d) \to (a, List\ c) \to (b, List\ d) \\
fmap2_{FRose} & = & \lambda p\ r \to fmap2_{Par}\ p\ r \mathbin{-\!\!*\!\!-} fmap2_{List@Rec}\ p\ r \\
fmap2_{List@Rec} & :: & (a \to b) \to (c \to d) \to List\ c \to List\ d \\
fmap2_{List@Rec} & = & \lambda p\ r \to pmap_{List}\ (fmap2_{Rec}\ p\ r) \\
inn_{Rose} & :: & (a, List\ (Rose\ a)) \to Rose\ a \\
inn_{Rose} & = & uncurry\ Fork
\end{array}
$$

$$
\begin{array}{lcl}
out_{Rose} :: Rose\ a & \to & (a, List\ (Rose\ a)) \\
out_{Rose}\ (Fork\ a\ b) & = & (a, b)
\end{array}
$$

Figure 3.6: Generated code for $pmap_{Rose}$

$$
\begin{array}{lcl}
pmap_{List} & :: & (a \to b) \to List\ a \to List\ b \\
pmap_{List}\ f & = & inn_{List} \circ fmap2_{FList}\ f\ (pmap_{List}\ f) \circ out_{List} \\
fmap2_{FList} & :: & (a \to b) \to (c \to d) \to Either\ ()\ (a, c) \to Either\ ()\ (b, d) \\
fmap2_{FList} & = & \lambda p\ r \to fmap2_{Empty}\ p\ r \mathbin{-\!\!+\!\!-} fmap2_{Par*Rec}\ p\ r \\
fmap2_{Empty} & :: & (a \to b) \to (c \to d) \to () \to () \\
fmap2_{Empty} & = & \lambda p\ r \to id \\
fmap2_{Par*Rec} & :: & (a \to b) \to (c \to d) \to (a, c) \to (b, d) \\
fmap2_{Par*Rec} & = & \lambda p\ r \to fmap2_{Par}\ p\ r \mathbin{-\!\!*\!\!-} fmap2_{Rec}\ p\ r
\end{array}
$$

Figure 3.7: Generated code for $pmap_{List}$

3.8 Catamorphisms on specific datatypes

The first argument of function $cata$ is a function of type $\Phi_d\ a\ b \to b$. Polytypic functions of this form, that is functions polymorphic in d, can only be constructed by means of functions *inn*, *out*, and functions defined by means of the **polytypic** construct (like *fsum*). In all these cases the resulting function is also polytypic. If we only want to use *cata* to define a function from one specific datatype $D\ a$, then we do not need a polytypic argument, but can construct an ordinary function of type $\Phi_D\ a\ b \to b$ where Φ_D is the concrete type constructor representing the pattern functor of the datatype $D\ a$.

As an example we define the function *eval* on the datatype *BoolExp a* by means of a *cata*:

$$
\begin{array}{lcl}
\mathbf{data}\ BoolExp\ a & = & Con\ a \\
 & | & Not\ (BoolExp\ a) \\
 & | & And\ (BoolExp\ a)\ (BoolExp\ a) \\
 & | & Or\ (BoolExp\ a)\ (BoolExp\ a) \\
\text{-- } \Phi_{BoolExp} & = & Par + Rec + Rec * Rec + Rec * Rec
\end{array}
$$

$$
\begin{array}{lcl}
eval & :: & BoolExp\ Bool \to Bool \\
eval & = & cata\ feval \\
feval & :: & \Phi_{BoolExp}\ Bool\ Bool \to Bool \\
feval & = & id \\
 & \triangledown & (\neg) \\
 & \triangledown & uncurry\ (\wedge) \\
 & \triangledown & uncurry\ (\vee)
\end{array}
$$

$$\text{-- } eval\ =\ cata\ \{Con \mapsto id, Not \mapsto (\neg), And \mapsto (\wedge), Or \mapsto (\vee)\}$$

This evaluation function is an example of a function that cannot be made polytypic: The pattern functor for *BoolExp* contains two occurrences of the functor $Rec * Rec$ (for *And* and *Or*), and each polytypic function will behave in the same way on these functors. That *eval* cannot be made polytypic should not be all too surprising, it simply means that the there is no general algorithm that given the abstract syntax for an expression language produces the intended semantics for that language!

3.9 Separate: a simple PolyP program

This section presents a simple PolyP program for separating a datatype value into its shape and its content, together with the code PolyP generates for this program. To make the program self-contained, we repeat those definitions from preceding sections that are used in the algorithm. In fact, the remainder of this section is a literal script containing the complete PolyP program Separate and Figure 3.8 contains the code generated from this program by the PolyP compiler.

The default starting point for code generation in a PolyP file is the value *main*. Everything possibly reachable from *main* is instantiated. In this example we choose to test *separate* on a tree.

$$
\begin{array}{lcl}
\mathit{main} & = & \mathit{print}\ \mathit{test} >> \mathit{print}\ \mathit{answer} >> \mathit{print}\ (\mathit{test} == \mathit{answer}) \\
\mathit{test}, \mathit{answer} & :: & (\mathit{Tree}\ (), [\mathit{Int}]) \\
\mathit{test} & = & \mathit{separate}\ (\mathit{Bin}\ (\mathit{Leaf}\ 17)\ (\mathit{Leaf}\ 38)) \\
\mathit{answer} & = & (\mathit{Bin}\ (\mathit{Leaf}\ ())\ (\mathit{Leaf}\ ()), [17, 38]) \\
\mathbf{data}\ \mathit{Tree}\ a & = & \mathit{Leaf}\ a \mid \mathit{Bin}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\ \mathbf{deriving}\ (\mathit{Show}, \mathit{Eq})
\end{array}
$$

Function *separate* takes an element of a regular datatype (of type $d\ a$) and generates a pair. The first component of the pair is just the structure of the datatype without the contents (of type $d\ ()$) and the second component is just the contents without the structure (of type $[a]$). We return to function *separate* in Chapter 6.

$$
\begin{array}{l}
\mathit{separate} :: \mathit{Regular}\ d \Rightarrow d\ a \to (d\ (), [a]) \\
\mathit{separate}\ x\ =\ (\mathit{pmap}\ (\mathit{const}\ ())\ x, \mathit{flatten}\ x)
\end{array}
$$

Mapping

$$
\begin{array}{l}
\mathit{pmap} :: \mathit{Regular}\ d \Rightarrow (a \to b) \to d\ a \to d\ b \\
\mathit{pmap}\ f\ =\ \mathit{inn} \circ \mathit{fmap2}\ f\ (\mathit{pmap}\ f) \circ \mathit{out} \\
\mathbf{polytypic}\ \mathit{fmap2} :: (a \to c) \to (b \to d) \to f\ a\ b \to f\ c\ d \\
\quad =\ \lambda p\ r \to \mathbf{case}\ f\ \mathbf{of}
\end{array}
$$

$$
\begin{array}{lcl}
g + h & \to & (\mathit{fmap2}\ p\ r) -\!\!+ (\mathit{fmap2}\ p\ r) \\
g * h & \to & (\mathit{fmap2}\ p\ r) -\!\!* (\mathit{fmap2}\ p\ r) \\
\mathit{Empty} & \to & \mathit{id} \\
\mathit{Par} & \to & p \\
\mathit{Rec} & \to & r \\
d@g & \to & \mathit{pmap}\ (\mathit{fmap2}\ p\ r) \\
\mathit{Const}\ t & \to & \mathit{id}
\end{array}
$$

Non-polytypic help functions

$$
\begin{array}{lcl}
(\ -\!\!*\ ) & :: & (a \to c) \to (b \to d) \to ((a, b) \to (c, d)) \\
(\ -\!\!+\ ) & :: & (a \to c) \to (b \to d) \to (\mathit{Either}\ a\ b \to \mathit{Either}\ c\ d) \\
f -\!\!* g & = & \lambda(x, y) \to (f\ x, g\ y) \\
f -\!\!+ g & = & \mathit{Left} \circ f \triangledown \mathit{Right} \circ g
\end{array}
$$

Flattening

$$
\begin{array}{l}
\mathit{flatten} :: \mathit{Regular}\ d \Rightarrow\ d\ a \to [\,a\,] \\
\mathit{flatten}\ =\ \mathit{fflatten} \circ \mathit{fmap2}\ \mathit{singleton}\ \mathit{flatten} \circ \mathit{out} \\
\textbf{polytypic}\ \mathit{fflatten}\quad ::\quad f\ [\,a\,]\ [\,a\,] \to [\,a\,] \\
\quad = \textbf{case}\ f\ \textbf{of} \\
\qquad \begin{array}{lcl}
g + h & \to & \mathit{fflatten}\ \triangledown\ \mathit{fflatten} \\
g * h & \to & \lambda(x, y) \to \mathit{fflatten}\ x +\!\!+\ \mathit{fflatten}\ y \\
\mathit{Empty} & \to & \mathit{nil} \\
\mathit{Par} & \to & \mathit{id} \\
\mathit{Rec} & \to & \mathit{id} \\
d@g & \to & \mathit{concat} \circ \mathit{flatten} \circ \mathit{pmap}\ \mathit{fflatten} \\
\mathit{Const}\ t & \to & \mathit{nil}
\end{array}
\end{array}
$$

Help functions for lists

$$
\begin{array}{lcl}
\mathit{singleton} & :: & a \to [\,a\,] \\
\mathit{singleton}\ x & = & [\,x\,] \\
\mathit{nil} & :: & a \to [\,b\,] \\
\mathit{nil}\ x & = & [\,]
\end{array}
$$

3.10 Polytypic laws

Function *cata* satisfies a generalization of the fusion law. The fusion law gives conditions under which intermediate values produced by the catamorphism can be eliminated. The fusion law is polytypic, that is, it holds for every regular datatype.

$$
\begin{array}{ll}
 & h \circ \mathit{cata}\ f\ =\ \mathit{cata}\ g \\
\Leftarrow & \{\,\text{Fusion}\,\} \\
 & h \circ f\ =\ g \circ \mathit{fmap2}\ \mathit{id}\ h
\end{array}
\quad \text{where} \left\{
\begin{array}{lcl}
f & :: & \Phi_d\ a\ b \to b \\
g & :: & \Phi_d\ a\ c \to c \\
h & :: & b \to c \\
\mathit{cata}\ f & :: & d\ a \to b \\
\mathit{cata}\ g & :: & d\ a \to c
\end{array}\right.
$$

The formulation of the fusion is an instance of the free theorem [101] of function *cata*. If we allow partial or infinite objects, then we must add the extra requirement that h be strict.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Tree}\ a\ =\ \mathit{Leaf}\ (a)\mid \mathit{Bin}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\ \textbf{deriving}\ (\mathit{Show},\mathit{Eq})\\
\mathit{main} :: \mathit{IO}\ ()\\
\mathit{main}\ =\ ((\mathit{print}\ \mathit{test}) >\!\!> (\mathit{print}\ \mathit{answer})) >\!\!> (\mathit{print}\ (\mathit{test} == \mathit{answer}))\\
\mathit{test} :: (\mathit{Tree}\ (),[\mathit{Int}])\\
\mathit{test}\ =\ \mathit{separate}_{\mathit{Tree}}\ (\mathit{Bin}\ (\mathit{Leaf}\ 17)\ (\mathit{Leaf}\ 38))\\
\mathit{answer} :: (\mathit{Tree}\ (),[\mathit{Int}])\\
\mathit{answer}\ =\ (\mathit{Bin}\ (\mathit{Leaf}\ ())\ (\mathit{Leaf}\ ()),17:(38:([\,])))\\
\mathit{separate}_{\mathit{Tree}} :: \mathit{Tree}\ a \to (\mathit{Tree}\ (),[a])\\
\mathit{separate}_{\mathit{Tree}}\ x\ =\ (\mathit{pmap}_{\mathit{Tree}}\ (\mathit{const}\ ())\ x,\mathit{flatten}_{\mathit{Tree}}\ x)\\
\mathit{pmap}_{\mathit{Tree}} :: (a \to b) \to \mathit{Tree}\ a \to \mathit{Tree}\ b\\
\mathit{pmap}_{\mathit{Tree}}\ f\ =\ \mathit{inn}_{\mathit{Tree}} \circ ((\mathit{fmap2}_{\mathit{FTree}}\ f\ (\mathit{pmap}_{\mathit{Tree}}\ f)) \circ \mathit{out}_{\mathit{Tree}})\\
\mathit{flatten}_{\mathit{Tree}} :: \mathit{Tree}\ a \to [a]\\
\mathit{flatten}_{\mathit{Tree}}\ =\ \mathit{fflatten}_{\mathit{FTree}} \circ ((\mathit{fmap2}_{\mathit{FTree}}\ \mathit{singleton}\ \mathit{flatten}_{\mathit{Tree}}) \circ \mathit{out}_{\mathit{Tree}})\\
\mathit{inn}_{\mathit{Tree}} :: \mathit{Either}\ a\ (\mathit{Tree}\ a,\mathit{Tree}\ a) \to \mathit{Tree}\ a\\
\mathit{inn}_{\mathit{Tree}}\ =\ \mathit{Leaf}\ \triangledown\ \mathit{uncurry}\ \mathit{Bin}\\
\mathit{fmap2}_{\mathit{FTree}} :: (a \to b) \to (c \to d) \to \mathit{Either}\ a\ (c,c) \to \mathit{Either}\ b\ (d,d)\\
\mathit{fmap2}_{\mathit{FTree}}\ =\ \lambda p\ r \to (\mathit{fmap2}_{\mathit{Par}}\ p\ r) -\!\!+\!\!- (\mathit{fmap2}_{\mathit{Rec*Rec}}\ p\ r)\\
\mathit{out}_{\mathit{Tree}} :: \mathit{Tree}\ a \to \mathit{Either}\ a\ (\mathit{Tree}\ a,\mathit{Tree}\ a)\\
\mathit{out}_{\mathit{Tree}}\ x\ =\ \textbf{case}\ x\ \textbf{of}\\
\qquad\qquad\qquad (\mathit{Leaf}\ a) \to \mathit{Left}\ a\\
\qquad\qquad\qquad (\mathit{Bin}\ a\ b) \to \mathit{Right}\ (a,b)\\
\mathit{fflatten}_{\mathit{FTree}} :: \mathit{Either}\ [a]\ ([a],[a]) \to [a]\\
\mathit{fflatten}_{\mathit{FTree}}\ =\ \mathit{fflatten}_{\mathit{Par}}\ \triangledown\ \mathit{fflatten}_{\mathit{Rec*Rec}}\\
\mathit{singleton} :: a \to [a]\\
\mathit{singleton}\ x\ =\ x:([\,])\\
(\ -\!\!+\!\!-\ ) :: (a \to b) \to (c \to d) \to \mathit{Either}\ a\ c \to \mathit{Either}\ b\ d\\
f -\!\!+\!\!- g\ =\ \mathit{Left} \circ f\ \triangledown\ \mathit{Right} \circ g\\
\mathit{fmap2}_{\mathit{Par}} :: (a \to b) \to (c \to d) \to a \to b\\
\mathit{fmap2}_{\mathit{Par}}\ =\ \lambda p\ r \to p\\
\mathit{fmap2}_{\mathit{Rec*Rec}} :: (a \to b) \to (c \to d) \to (c,c) \to (d,d)\\
\mathit{fmap2}_{\mathit{Rec*Rec}}\ =\ \lambda p\ r \to (\mathit{fmap2}_{\mathit{Rec}}\ p\ r) -\!\!*\!\!- (\mathit{fmap2}_{\mathit{Rec}}\ p\ r)\\
\mathit{fflatten}_{\mathit{Par}} :: [a] \to [a]\\
\mathit{fflatten}_{\mathit{Par}}\ =\ \mathit{id}\\
\mathit{fflatten}_{\mathit{Rec*Rec}} :: ([a],[a]) \to [a]\\
\mathit{fflatten}_{\mathit{Rec*Rec}}\ =\ \lambda(x,y) \to (\mathit{fflatten}_{\mathit{Rec}}\ x) +\!\!+ (\mathit{fflatten}_{\mathit{Rec}}\ y)\\
(\ -\!\!*\!\!-\ ) :: (a \to b) \to (c \to d) \to (a,c) \to (b,d)\\
f -\!\!*\!\!- g\ =\ \lambda(x,y) \to (f\ x,g\ y)\\
\mathit{fmap2}_{\mathit{Rec}} :: (a \to b) \to (c \to d) \to c \to d\\
\mathit{fmap2}_{\mathit{Rec}}\ =\ \lambda p\ r \to r\\
\mathit{fflatten}_{\mathit{Rec}} :: [a] \to [a]\\
\mathit{fflatten}_{\mathit{Rec}}\ =\ \mathit{id}
\end{array}
$$

Figure 3.8: The Haskell code generated from Separate.

Proof: Assume $h \circ f = g \circ \mathit{fmap2}\ \mathit{id}\ h$ is true. Use fixed point induction (Theorem 2.15) with $n = 2$ and improvement functions and inclusive relation given by

$$\begin{array}{lcl} i_1\ x & = & f \circ \mathit{fmap2}\ \mathit{id}\ x \circ \mathit{out} \\ i_2\ y & = & g \circ \mathit{fmap2}\ \mathit{id}\ y \circ \mathit{out} \\ P\ (x, y) & = & h \circ x === y\ . \end{array}$$

Base case: $P\ (\bot, \bot) = h \circ \bot === \bot = \mathit{true}$ if h is strict.

Inductive case: We calculate as follows.

$$\begin{array}{ll} & h \circ i_1\ x === i_2\ y \\ \equiv & \{\text{ Definitions of } i_1 \text{ and } i_2\ \} \\ & h \circ f \circ \mathit{fmap2}\ \mathit{id}\ x \circ \mathit{out} === g \circ \mathit{fmap2}\ \mathit{id}\ y \circ \mathit{out} \\ \Leftarrow & \{\text{ Cancel } (\circ \mathit{out})\ \} \\ & h \circ f \circ \mathit{fmap2}\ \mathit{id}\ x === g \circ \mathit{fmap2}\ \mathit{id}\ y \\ \equiv & \{\text{ Use the assumption on the left }\} \\ & g \circ \mathit{fmap2}\ \mathit{id}\ h \circ \mathit{fmap2}\ \mathit{id}\ x === g \circ \mathit{fmap2}\ \mathit{id}\ y \\ \equiv & \{\text{ Function } \mathit{fmap2} \text{ is a functor (preserves composition) }\} \\ & g \circ \mathit{fmap2}\ \mathit{id}\ (h \circ x) === g \circ \mathit{fmap2}\ \mathit{id}\ y \\ \equiv & \{\text{ Induction hypothesis: } P\ (x, y) = h \circ x === y\ \} \\ & \mathit{True} \end{array}$$

□

As an example of the use of the fusion law we can prove that the function $\mathit{pmap}\ f$ also can be defined as a catamorphism:

$$\begin{array}{l} \mathit{pmap}' :: \mathit{Regular}\ d \Rightarrow (a \to b) \to d\ a \to d\ b \\ \mathit{pmap}'\ f = \mathit{cata}\ (\mathit{inn} \circ \mathit{fmap2}\ f\ \mathit{id}) \end{array}$$

Proof: The proof is by calculation:

$$
\begin{array}{ll}
& \mathit{pmap}\ f === \mathit{pmap}'\ f \\
\equiv & \{\ \text{identity}\ \} \\
& \mathit{pmap}\ f \circ \mathit{id} === \mathit{pmap}'\ f \\
\equiv & \{\ \text{identity is a catamorphism, definition of } \mathit{pmap}'\ \} \\
& \mathit{pmap}\ f \circ \mathit{cata}\ \mathit{inn} === \mathit{cata}\ (\mathit{inn} \circ \mathit{fmap2}\ f\ \ \mathit{id}) \\
\Leftarrow & \{\ \text{fusion } (\mathit{pmap}\ f \text{ is strict})\ \} \\
& \mathit{pmap}\ f \circ \mathit{inn} === (\mathit{inn} \circ \mathit{fmap2}\ f\ \ \mathit{id}) \circ \mathit{fmap2}\ \mathit{id}\ (\mathit{pmap}\ f) \\
\equiv & \{\ \text{definition of } \mathit{pmap}, \mathit{fmap2} \text{ is a bifunctor}\ \} \\
& \mathit{inn} \circ \mathit{fmap2}\ f\ \ (\mathit{pmap}\ f) \circ \mathit{out} \circ \mathit{inn} === \mathit{inn} \circ \mathit{fmap2}\ f\ \ (\mathit{pmap}\ f) \\
\equiv & \{\ \mathit{out} \text{ is the inverse of } \mathit{inn}, \text{ identity}\ \} \\
& \mathit{inn} \circ \mathit{fmap2}\ f\ \ (\mathit{pmap}\ f) === \mathit{inn} \circ \mathit{fmap2}\ f\ \ (\mathit{pmap}\ f) \\
\equiv & \{\ \text{Trivially}\ \} \\
& \mathit{True}
\end{array}
$$

□

As examples of laws for polytypic functions we present the laws expressing that *pmap* and *fmap2* are functors:

$$
\begin{array}{lll}
\mathit{pmap}\ \mathit{id} & === & \mathit{id} \\
\mathit{pmap}\ f \circ \mathit{pmap}\ g & === & \mathit{pmap}\ (f \circ g) \\
\mathit{fmap2}\ \mathit{id}\ \mathit{id} & === & \mathit{id} \\
\mathit{fmap2}\ f\ \ g \circ \mathit{fmap2}\ h\ \ i & === & \mathit{fmap2}\ (f \circ h)\ (g \circ i)
\end{array}
$$

The functor laws for *fmap2* are easily proved from corresponding laws for $(+)$ and $(*)$ by induction over the structure of regular datatypes. The functor laws for *pmap* are proved by fixed point induction using the laws for *fmap2*. These laws, and many others, are presented in PolyLib (Chapter 5).

Chapter 4

PolyP — a polytypic programming language extension[1]

This chapter briefly presents the underlying theory of the functional programming language extension PolyP. In PolyP, definitions of polytypic functions are type checked, and for all other expressions the types are inferred, using an extension of Jones' theories of qualified types and higher-order polymorphism. The semantics of PolyP programs is obtained by first adding functor arguments to polytypic functions in a dictionary passing style and then eliminating these arguments using partial evaluation to obtain a Haskell program. The notation and many definitions in this chapter are based on the work of Jones [63–66]. We are in the process of moving from the PolyP system and the theory presented in this paper, to a system called Generic Haskell [33] and Hinze's [35] theory of type indexed values.

The chapter is organized as follows. Section 4.1 discusses the type inference and checking algorithms used in PolyP. Section 4.2 gives the semantics of PolyP, and Section 4.3 shows how to generate Haskell code from PolyP programs. Section 4.4 presents a short overview of the implementation of PolyP. Section 4.5 concludes the chapter.

4.1 Type inference

Polytypic value definitions can be type checked, and for all other expressions the type can be inferred. This section discusses the type checking and type inference algorithms.

[1] This chapter is a revised version of an article with the same title, presented at the ACM Symposium on Principles of Programming Languages in 1997 [46].

$$
\begin{array}{lcll}
E & ::= & x & \text{variable} \\
 & | & E\ E & \text{application} \\
 & | & \lambda x.E & \text{abstraction} \\
 & | & \mathbf{let}\ Q\ \mathbf{in}\ E & \text{let-expression} \\
 & & & \\
Q & ::= & x = E & \text{variable binding} \\
 & & & \\
C^{\kappa} & ::= & \chi^{\kappa} & \text{constants} \\
 & | & \alpha^{\kappa} & \text{variables} \\
 & | & C^{\kappa' \to \kappa} C^{\kappa'} & \text{applications} \\
 & & & \\
\tau & ::= & C^{*} & \text{types} \\
\rho & ::= & P \Rightarrow \tau & \text{qualified types} \\
\sigma & ::= & \forall t_i^{\kappa}.\rho & \text{type schemes}
\end{array}
$$

Figure 4.1: The core language QML

Section 4.1.1 introduces the core language without the **polytypic** construct, but with qualified and higher-order polymorphic types. Section 4.1.2 extends the core language with the **polytypic** construct and some built-in functions, types and classes. Section 4.1.3 discusses unification in the extended language, and the Section 4.1.4 shows how to type check a polytypic value definition.

4.1.1 The core language

Our core language is an extension of core-ML with qualified types and higher-order polymorphism [66], see Figure 4.1. The non-terminal for types in this grammar is really a kind-indexed family of non-terminals where the superscript denotes its kind. For example, a basic type is in C^* (has kind $*$), and a parametric datatype constructor such as *List* is in $C^{*\to *}$ (has kind $* \to *$). We call the resulting language QML. The set of constructor constants contains:

$$(\to), (,), \mathit{Either} :: * \to * \to *$$

A program consists of a list of datatype declarations and a binding for *main*.

The typing rules and the type inference algorithm are based on the extensions of the standard rules and algorithm [17] that handle qualified and higher-order polymorphic types, see Jones [64,66]. Compared to the traditional Hindley-Milner system the type judgments are extended with a set of predicates P. The rules involving essential changes in the predicate set are shown in Figure 4.2. The other rules and the algorithm are omitted. The entailment relation $\Vdash$ relates sets of predicates and is used to reason about qualified types, see Jones [64].

$$(\Rightarrow E) \quad \frac{P \mid \Gamma \vdash e : \pi \Rightarrow \rho \qquad P \Vdash \pi}{P \mid \Gamma \vdash e : \rho}$$

$$(\Rightarrow I) \quad \frac{P, \pi \mid \Gamma \vdash e : \rho}{P \mid \Gamma \vdash e : \pi \Rightarrow \rho}$$

Figure 4.2: Some of the typing rules for QML

4.1.2 The polytypic language extension

The polytypic extension of QML consists of two parts — an extension of the type system and an extension of the expression language. We call the extended QML language polyQML.

Extending the type system

The type system is extended by generalizing the unification algorithm and by adding new types, kinds and classes to the initial type environment. The initial type environment of the language polyQML consists of four components: a family of functor constructors Φ_d, the types of the functions *inn* and *out*, the type classes *Regular* and *Bifunctor*, and the collection of functor constructors ($+$, $*$, *Empty*, *Par*, *Rec*, @ and *Const* t).

- For every regular datatype $D\ a$ the type constructor Φ_D (written *FunctorOf* D in the actual code) represents its pattern functor. The constructor Φ has kind $1 \rightarrow 2$ where 1 abbreviates the kind of regular type constructors $(* \rightarrow *)$ and 2 abbreviates the kind of pattern functors $(* \rightarrow * \rightarrow *)$.

- The class *Regular* contains all regular datatypes and the class *Bifunctor* contains the functors of all regular datatypes. To reflect this, the entailment relation is extended as follows for polyQML:

  $\Vdash$ *Regular* D, for all regular datatypes $D\ a$
  Regular $d \Vdash$ *Bifunctor* Φ_d

- The functor constructors obtained from Section 3.4 are added to the constructor constants, and have the following kinds:

  $$\begin{array}{lcl} *, + & :: & 2 \rightarrow 2 \rightarrow 2 \\ \mathit{Empty}, \mathit{Par}, \mathit{Rec} & :: & 2 \\ @ & :: & 1 \rightarrow 2 \rightarrow 2 \\ \mathit{Const} & :: & * \rightarrow 2 \end{array}$$

  The corresponding rules in the entailment relation are the following:

$$\begin{array}{l}
\mathit{Bifunctor}\ f, \mathit{Bifunctor}\ g \Vdash \mathit{Bifunctor}\ (f+g), \mathit{Bifunctor}\ (f*g)\\
\Vdash \mathit{Bifunctor}\ \mathit{Empty}, \mathit{Bifunctor}\ \mathit{Par}, \mathit{Bifunctor}\ \mathit{Rec}\\
\mathit{Regular}\ d, \mathit{Bifunctor}\ g \Vdash \mathit{Bifunctor}\ (d@g)\\
\Vdash \mathit{Bifunctor}\ (\mathit{Const}\ t)
\end{array}$$

- Functions *inn* and *out* were introduced in Section 3.5.

$$\begin{array}{lcl}
\mathit{out} & :: & \mathit{Regular}\ d \Rightarrow d\ a \rightarrow \Phi_d\ a\ (d\ a)\\
\mathit{inn} & :: & \mathit{Regular}\ d \Rightarrow \Phi_d\ a\ (d\ a) \rightarrow d\ a
\end{array}$$

  Note that these functions have qualified higher-order polymorphic types.

The resulting type system is quite powerful; it can be used to type check many polytypic programs in a context assigning types to a number of basic polytypic functions. But although we can use and combine polytypic functions, we cannot define new polytypic functions by induction on the structure of datatypes.

At this point we could choose to add some basic polytypic functions that really need an inductive definition to the type environment. This would give us roughly the same expressive power as the language given by Jay [55] extended with qualified types. As a minimal example we could add *fmap2* to the initial environment:

$$\mathit{fmap2} :: \mathit{Bifunctor}\ f \Rightarrow (a \rightarrow b) \rightarrow (c \rightarrow d) \rightarrow f\ a\ c \rightarrow f\ b\ d$$

This would allow us to define and type check polytypic functions like *pmap* and *cata*. The type checking algorithm would for example derive

$$\mathit{pmap}\ (+1)\ (\mathit{Leaf}\ 4) :: \mathit{Regular}\ \mathit{Tree} \Rightarrow \mathit{Tree}\ \mathit{Int}$$

but it would, at best, be hard to write a polytypic version of a function like *zip*. Adding the **polytypic** construct to our language makes writing polytypic programs much simpler.

Adding the polytypic construct

To add the **polytypic** construct, the production for variable bindings in the **let**-expression, Q, is extended with

$$\textbf{polytypic}\ \ x :: \rho = \textbf{case}\ \ f^{*\rightarrow*\rightarrow*}\ \ \textbf{of}\ \ \{f_i \rightarrow e_i\}$$

where f is a functor variable[2], f_i are functor patterns (the grammar for functors was defined in Section 3.4). The functor patterns can be nested and overlapping,

[2]Case analysis over more than one functor can be simulated by handling all but the first functor in the e_i by other polytypic constructs. In the future we might extend the syntax to simplify this.

$$\frac{\Gamma' = (\Gamma, \gamma), \quad \gamma = (x : \sigma), \quad P_i \mid \Gamma' \vdash e_i : \{f \mapsto f_i\}\sigma}{P_1, \ldots, P_n \mid \Gamma \vdash \textbf{polytypic}\ \ x :: \sigma = \textbf{case}\ \ f\ \ \textbf{of}\ \ \{f_i \to e_i\} : \gamma}$$

Figure 4.3: The typing rule for **polytypic**

$$\begin{array}{lcl}
\textbf{type}\ (g + h)\ p\ r & = & \mathit{Either}\ (g\ p\ r)\ (h\ p\ r) \\
\textbf{type}\ (g * h)\ p\ r & = & (g\ p\ r, h\ p\ r) \\
\textbf{type}\ \mathit{Empty}\ p\ r & = & () \\
\textbf{type}\ \mathit{Par}\ p\ r & = & p \\
\textbf{type}\ \mathit{Rec}\ p\ r & = & r \\
\textbf{type}\ (d @ g)\ p\ r & = & d\ (g\ p\ r) \\
\textbf{type}\ \mathit{Const}\ t\ p\ r & = & t
\end{array}$$

Figure 4.4: Interpreting functors as type synonyms

but they must be linear. The resulting language is polyQML. To be able to do the case analysis over a functor, the functor must be constructed from the operators $+$, $*$, @ and the type constants *Empty*, *Par*, *Rec* and *Const t*. This is equivalent to being in the class *Bifunctor* and thus the context *Bifunctor f* must always be included in the type ρ of a function defined by the **polytypic** construct. As *Bifunctor f* must always be in the type, PolyP inserts it automatically if it is not given explicitly.

The typing rules for polyQML are the rules from QML together with the rule for typing the **polytypic** construct given in Figure 4.3. For the notation used, see Jones [64]. Note that the **polytypic** construct is not an expression but a binding, and hence the typing rule returns a binding. The rule is not as simple as it looks — the substitution $\{f \mapsto f_i\}$ replaces a functor variable with a functor interpreted as a partially applied type synonym, see Figure 4.4. For example, interpreting the functors in the pattern functor for *List* as type synonyms, we have:

$$\begin{array}{ll}
 & \Phi_{\mathit{List}}\ p\ r \\
\equiv & \quad \{\ \Phi_{\mathit{List}} = \mathit{Empty} + \mathit{Par} * \mathit{Rec}\ \ \} \\
 & (\mathit{Empty} + \mathit{Par} * \mathit{Rec})\ p\ r \\
\equiv & \quad \{\ \text{Type synonym for } +\ \} \\
 & \mathit{Either}\ (\mathit{Empty}\ p\ r)\ ((\mathit{Par} * \mathit{Rec})\ p\ r) \\
\equiv & \quad \{\ \text{Type synonyms for } \mathit{Empty} \text{ and } *\ \} \\
 & \mathit{Either}\ ()\ (\mathit{Par}\ p\ r, \mathit{Rec}\ p\ r)
\end{array}$$

$\equiv$ { Type synonyms for *Par* and *Rec* }

$Either\ ()\ (p, r)$

4.1.3 Unification

The (standard but omitted) typing rule for application uses a unification algorithm to unify the argument type of a function with the type of its argument.

The unification algorithm we use is the kind-preserving unification algorithm of Jones [66], which is an extension of Robinson's well-known unification algorithm. We write $C \overset{\sigma}{\sim} C'$ if C and C' are unified by substitution σ.

Theorem 4.1 *If there is a unifier for two given types C, C', then $C \overset{\sigma}{\sim} C'$ using Jones [66] algorithm for kind-preserving unification, and σ is a most general unifier for C and C'. Conversely, if no unifier exists, then the unification algorithm fails.*

4.1.4 Type checking the polytypic construct

Instances of polytypic functions generated by means of a function defined with the **polytypic** construct should be type correct. For that purpose we type check polytypic functions.

Type checking a polytypic value definition amounts to checking that the inferred types for the case branches are more general than the corresponding instances of the explicitly given type. So for each polytypic value definition

$$\textbf{polytypic}\ \ x :: \rho = \textbf{case}\ \ f\ \ \textbf{of}\ \ \{f_i \rightarrow e_i\}$$

we have to do the following for each branch of the case:

- Infer the type of $e_i : \tau_i$.
- Calculate the type the branches should have according to the explicit type: $\rho_i = \{f \mapsto f_i\}\rho$.
- Check that ρ_i is an instance of τ_i.

When calculating the types of the alternatives the functor constructors are treated as type synonyms defined in Figure 4.4. The complete type inference/checking algorithm $\mathcal{W}$ is obtained by extending Jones' type inference algorithm [66] with the alternative for the **polytypic** construct. Some of the rules of the algorithm

$$\text{(var)} \quad \frac{(x : \forall t_i.P \Rightarrow \tau) \in \Gamma, \quad s_i \text{ new}, \quad S = \{t_i \mapsto s_i\}}{SP \mid \Gamma \vdash^w x : S\tau}$$

$$\text{(let)} \quad \frac{S(\Gamma) \vdash^w q : \gamma, \quad Q \mid T(S\Gamma, \gamma) \vdash^w e : \tau}{Q \mid TS(\Gamma) \vdash^w \textbf{let } q \textbf{ in } e : \tau}$$

$$\text{(bind)} \quad \frac{P \mid S(\Gamma) \vdash^w e : \tau, \quad \gamma = (x : \forall_{S\Gamma}(P \Rightarrow \tau))}{S(\Gamma) \vdash^w x = e : \gamma}$$

$$\text{(poly)} \quad \frac{\begin{array}{c} \Gamma' = (\Gamma, \gamma), \quad \gamma = (x : \forall_{\{\}}(\rho)) \\ P_i \mid S_i(T_{i-1}\Gamma') \vdash^w e_i : \tau_i \\ \forall_{T_n\Gamma'}(S_n \cdots S_{i+1}(P_i \Rightarrow \tau_i)) \geq \{f \mapsto f_i\}\rho \\ T_0 = \{\}, \quad T_i = S_i T_{i-1} \end{array}}{\Gamma \vdash^w \textbf{polytypic } x :: \rho = \textbf{case } f \textbf{ of } \{f_i \to e_i\} : \gamma}$$

Figure 4.5: Some parts of $\mathcal{W}$

are given in Figure 4.5. As an example we will sketch how the $g * h$ and *Rec* branches in the definition of *fsum* in Figure 1.1 are type checked:

$$\begin{array}{lll} \textbf{polytypic } \mathit{fsum} & :: & f\ \mathit{Int}\ \mathit{Int} \to \mathit{Int} \\ \quad = \textbf{case } f \textbf{ of} & & \\ \qquad \ldots & & \\ \qquad g * h & \to & \lambda(x, y) \to \mathit{fsum}\ x + \mathit{fsum}\ y \\ \qquad \mathit{Rec} & \to & \mathit{id} \\ \qquad \ldots & & \end{array}$$

In the $g * h$ branch of the polytypic case, we first infer the type of the expression $e_* = \lambda(x, y) \to \mathit{fsum}\ x + \mathit{fsum}\ y$. Using fresh instances of the explicit type $\rho = f\ \mathit{Int}\ \mathit{Int} \to \mathit{Int}$ for the two occurrences of *fsum* we get $\tau_* = (x\ \mathit{Int}\ \mathit{Int}, y\ \mathit{Int}\ \mathit{Int}) \to \mathit{Int}$. We then calculate the type ρ_*:

$$\rho_* = \{f \mapsto g * h\}\rho = (g * h)\ \mathit{Int}\ \mathit{Int} \to \mathit{Int} = (g\ \mathit{Int}\ \mathit{Int}, h\ \mathit{Int}\ \mathit{Int}) \to \mathit{Int}$$

Because $\rho_* = \{x \mapsto g, y \mapsto h\}\tau_*$ we see that ρ_* is an instance of τ_*.

In the *Rec* branch of the polytypic case, we first infer the type of the expression $e_{Rec} = id$. The type of this expression is $\tau_{Rec} = a \to a$. We then calculate the type $\rho_{Rec} = \{f \mapsto Rec\}\rho = Rec\ Int\ Int \to Int = Int \to Int$. Because $\rho_{Rec} = \{a \mapsto Int\}\tau_{Rec}$ we see that ρ_{Rec} is an instance of τ_{Rec}. The other branches are handled similarly.

If a **polytypic** binding can be type checked using the typing rules, then algorithm $\mathcal{W}$ also manages to type check the binding. Conversely, if algorithm $\mathcal{W}$ can type check a **polytypic** binding, then the binding can be type checked with the typing rules too. Together with the results from Jones [64] we obtain the following theorem.

Theorem 4.2 *The type inference/checking algorithm is sound and complete.*

Proof sketch. Both the proof of soundness and of completeness are by induction on the structure of the expression. The only part of the inference algorithm that is new is the handling of the **polytypic** construct. Because the **polytypic** construct is explicitly typed, all that soundness and completeness states is that the algorithm succeeds if and only if a type can be inferred for the case branches. Using Jones' lemmas about substitutions and type ordering ($\geq$) together with the induction hypothesis we can show that the algorithm succeeds if and only if there is a derivation using the type rules.

4.2 Semantics

The meaning of a QML expression is obtained by translating the expression into a version of the polymorphic λ-calculus called QP that includes constructs for evidence application and evidence abstraction. Evidence is needed in the code generation process to construct code for functions with contexts. As an example, the evidence for *Regular* D is a dictionary containing *inn* and *out* for $D\ a$, and a symbolic representation of the corresponding functor Φ_D. Again, the results from this section are based on Jones' work on qualified types [64].

The language QP has the same expressions as QML plus three new constructs:

$$
\begin{array}{lcll}
E & ::= & \cdots & \text{same as for QML expressions} \\
& | & E_e & \text{evidence application} \\
& | & \lambda v.E & \text{evidence abstraction} \\
& | & \mathbf{case}\ \ v\ \ \mathbf{of}\ \ \{e_i \to E_i\} & \text{dependent case over evidence} \\
\\
\sigma & ::= & C^* & \text{types} \\
& | & P \Rightarrow \sigma & \text{qualified types} \\
& | & \forall t_i^{\kappa}.\sigma & \text{polymorphic types}
\end{array}
$$

The special **case**-statement is used in the translation of the **polytypic** construct. The typing rules for QP are standard except for the dependent **case** over bifunctors.

$$\text{(var)} \quad \frac{x : \forall t_i.P \Rightarrow \tau \in \Gamma, \quad s_i \text{ and } v \text{ new}, \quad S = \{t_i \mapsto s_i\}}{v : SP \mid \Gamma \vdash^w x \rightsquigarrow x_v : S\tau}$$

$$\text{(let)} \quad \frac{\begin{array}{c} S(\Gamma) \vdash^w q \rightsquigarrow q' : \gamma \\ Q \mid T(S\Gamma, \gamma) \vdash^w e \rightsquigarrow e' : \tau \end{array}}{Q \mid TS(\Gamma) \vdash^w (\mathbf{let}\ q\ \mathbf{in}\ e) \rightsquigarrow (\mathbf{let}\ q'\ \mathbf{in}\ e') : \tau}$$

$$\text{(bind)} \quad \frac{v : P \mid S(\Gamma) \vdash^w e \rightsquigarrow e' : \tau, \quad \gamma = (x : \forall_{S\Gamma}(P \Rightarrow \tau))}{S(\Gamma) \vdash^w (x = e) \rightsquigarrow (x = \lambda v.e') : \gamma}$$

$$\text{(poly)} \quad \frac{\begin{array}{c} \Gamma' = (\Gamma, \gamma), \quad \gamma = (x : \forall_{\{\}}(\rho)) \\ v_i : P_i \mid S_i(T_{i-1}\Gamma') \vdash^w e_i \rightsquigarrow e_i' : \tau_i \\ \forall_{T_n\Gamma'}(S_n \cdots S_{i+1}(P_i \Rightarrow \tau_i)) \geq^{C_i} \forall_{\{\}}(\{f \mapsto f_i\}\rho) \\ T_0 = \{\}, \quad T_i = S_i T_{i-1} \end{array}}{\begin{array}{c} \Gamma \vdash^w \ \mathbf{polytypic}\ \ x :: \rho = \mathbf{case}\ \ f\ \ \mathbf{of}\ \ \{f_i \to e_i\} \ \ \rightsquigarrow \\ x = \lambda v.\mathbf{case}\ \ v\ \ \mathbf{of}\ \ \{f_i \to C_i(\lambda v_i.e_i')v\} : \gamma \end{array}}$$

Figure 4.6: Some translation rules

The translation rules for variables, let expressions, variable bindings and for the **polytypic** construct are given in Figure 4.6. The remaining rules are simple and omitted. A translation rule of the form $P \mid S(\Gamma) \vdash^w e \rightsquigarrow e' : \tau$ can be read as an attribute grammar. The inherited attributes (the input data) consist of a type context Γ and an expression e and the synthesized attributes (the output data) are the evidence context P, the substitution S, the translated QP expression e' and the inferred type τ.

$$\begin{array}{lllll} \mathit{fsum} = & \lambda v.\mathbf{case}\ \ v\ \ \mathbf{of} & & \\ & g + h & \to & \mathit{fsum}_g \ \triangledown \ \mathit{fsum}_h \\ & g * h & \to & \lambda(x, y) \to \ \mathit{fsum}_g\ x + \mathit{fsum}_h\ y \\ & \mathit{Empty} & \to & \lambda x \to 0 \\ & \mathit{Par} & \to & \mathit{id} \\ & \mathit{Rec} & \to & \mathit{id} \\ & d\ @\ g & \to & \mathit{psum}_d \circ \mathit{pmap}_d\ \mathit{fsum}_g \\ & \mathit{Const}\ t & \to & \lambda x \to 0 \end{array}$$

Figure 4.7: The translation of function *fsum* into QP

For example, if we translate function $\mathit{fsum} :: \mathit{Bifunctor}\ f \Rightarrow f\ \mathit{Int}\ \mathit{Int} \to \mathit{Int}$, then, after simplification, we obtain the code in Figure 4.7. Note that the branches of the case expression in the translated code have different (but related) types. This case expression is a restricted version of a dependent case.

In this translation we use a conversion function C, which transforms evidence abstractions applied to evidence parameters into an application of the right type. Function C is obtained from the expression $\sigma \geq^C \sigma'$, which expresses that σ is more general than σ' and that a witness for this statement is the conversion function $C : \sigma \rightarrow \sigma'$. Function C is a non-recursive function distributing (parts of) the evidence parameters to their positions on the right hand side. In a polytypic value definition, such as *fsum*, where the structure of the patterns on the left hand sides corresponds directly to the structure of the expressions on the right hand sides, the conversion function will behave exactly as the matching operation in the case statement. In this case, the conversion function is essentially the identity (just a variable renaming). The conversion function might be more complex in the case where the recursive structure of the polytypic value definition does not correspond directly to the recursive structure of the functor.

The inputs to function $\geq$ are the two type schemes σ and σ', and the output (if it succeeds) is the conversion function C. It succeeds if the unification algorithm succeeds on the types and the substitution is from the left type to the right type only, and if the evidence for the contexts in σ can be constructed from the evidence for the contexts in σ'. The function C is constructed from the entailment relation extended with evidence values.

As evidence for the fact that a functor f is a bifunctor we use the symbolic representation of f as an element of the datatype described by the grammar for pattern functors from Section 3.4:

$$f, g, h \ ::= \ g + h \mid g * h \mid \mathit{Empty} \mid \mathit{Par} \mid \mathit{Rec} \mid d@g \mid \mathit{Const}\ t\ .$$

The evidence for regularity of a datatype $D\ a$ is a dictionary with three components: the definitions of *inn* and *out* on the datatype and evidence that the corresponding functor is indeed a bifunctor.

Theorem 4.3 *The translation from polyQML to QP preserves well-typedness and succeeds for programs with unambiguous type schemes.*

Proof sketch. The proofs are by induction on the structure of the expression. The use of a special syntax for the dependent **case** expression and the fact that this expression only is introduced by the translation of the **polytypic** construct allows us to reuse most of the proofs from Jones' dissertation for the other syntactic constructs.

4.3 Code generation

To generate code for a polyQML program, we generate a QML expression from a polyQML expression in two steps:

- A polyQML expression is translated to a QP expression with explicit evidence parameters (dictionaries).
- The QP expression is partially evaluated with respect to the evidence parameters giving a program in QML.

When the program has been translated to QP all occurrences of the **polytypic** construct and all references to the classes *Regular* and *Bifunctor* have been removed and the program contains evidence parameters instead. We remove all evidence parameters introduced by polytypism by partial evaluation (in the style of Jones [63]). The partial evaluation is started at the *main* expression (which must have an unambiguous type) and is propagated through the program by generating requests from the *main* expression and its subexpressions. A problem with this scheme is that does not support separate compilation: it requires the whole program to be available for translation at once.

The evidence for regularity of a datatype $D\ a$ (the entailment $\Vdash$ *Regular* D) is a dictionary containing the functions *inn*, *out* and the bifunctor Φ_D. PolyP constructs these dictionaries using a few straightforward inductive functions over the abstract syntax of regular datatypes. Functions *inn* and *out* are obtained by selecting the correct component of the dictionary.

In practice, a PolyP program (a program written in a subset of Haskell extended with the polytypic construct) is compiled to Haskell. Section 3.9 contains an example of a simple PolyP program and the code that is generated by PolyP for this program (in Figure 3.8) .

If the size of the original program is n, and the total number of subexpressions of the bifunctors of the regular datatypes occurring in the program is m, then the size of the generated code is at most $n \times m$. Each request for an instance of a function defined by means of the polytypic construct on a datatype $D\ a$ results in as many functions as there are subexpressions in the bifunctor f for datatype $D\ a$ (including the bifunctors of the datatypes used in f). The efficiency of the generated code is only a constant factor worse than hand-written instances of polytypic functions. Most of the overhead is caused by the *inn* and *out* transformations which, as they are isomorphisms, could probably be removed by a more clever implementation.

4.4 Implementation

This section presents a brief overview of the implementation of the PolyP compiler.

The implementation of PolyP is written in Haskell and it is divided into about 30 Haskell modules with a total of about 7000 lines of literate Haskell code. The accumulated time spent on the implementation of PolyP is close to one man-year, but most of that time was spent on non-polytypic parts of the system. As the knowledge base in the field of polytypic programming has grown, and more standard tools for Haskell compiler construction have become available, a new implementation with enhanced functionality could probably be completed in less time. A re-implementation of the type inference algorithm, for example, could be based on "Typing Haskell in Haskell" by Jones [67].

The information flow inside PolyP is as follows:

- The parser takes an input file to a list of equations expressed in the abstract syntax.
- Dependency analysis splits these equations into datatype declarations and mutually recursive groups of function definitions.
- For each regular datatype the corresponding functor is calculated.
- The equation groups are labeled with type information and evidence values using the type inference algorithm from Section 4.1.
- The labeled equations are traversed to collect requests for instances of polytypic functions.
- For every request, code for an instance of a polytypic function is generated and appended to the equation list.
- The final equation list is pretty printed.

More details about the implementation of PolyP are presented in Jansson's licentiate thesis [50].

4.5 Conclusions and future work

We have shown how to extend a functional language with the **polytypic** construct. The **polytypic** construct considerably simplifies writing programs that have the same functionality on a large class of datatypes (polytypic programs).

The extension is a small but powerful extension of a language with qualified types and higher-order polymorphism. We have developed a compiler that compiles Haskell with the **polytypic** construct to plain Haskell.

A lot of work remains to be done. The compiler has to be extended to handle mutual recursive datatypes with an arbitrary number of type arguments and in which function spaces may occur. These extensions are planned for the successor of PolyP: Generic Haskell [33].

Chapter 5

PolyLib — a polytypic function library[1]

During the last few years we have used PolyP to construct a number of polytypic programs, for example for pattern matching, unification, rewriting (Chapter 6), parsing (Chapter 7), etc. These polytypic programs use several basic polytypic functions, such as the relatively well-known *cata* and *pmap*, but also less well-known functions such as *propagate* and *thread*. We have collected these basic polytypic functions in the library of PolyP: PolyLib. This chapter describes the polytypic functions in PolyLib, motivates their presence in the library, and gives a rationale for their design.

Of course, a library is an important part of a programming language. Languages like Java, Delphi, Perl are popular partly because of their useful and extensive libraries. For a polytypic programming language it is even more important to have a clear and well-designed library: writing polytypic programs is difficult, and we do not expect many programmers to *write* polytypic programs. On the other hand, many programmers *use* polytypic programs such as parser generators, equality functions, etc.

We expect that both the form and content of this description will change over time, in fact this is already the second attempt at describing the library of PolyP; the first was presented at the Workshop on Generic Programming, 1998 [47]. One of the goals of that paper was to obtain feedback on the library design from other researchers working within the field. This feedback has led to a few minor corrections and additions, and two bigger changes: we have added laws relating the polytypic functions (mainly free theorems [101]) and included the complete implementation in Appendix A. At the moment the library only contains the basic polytypic functions, but we are actively developing special purpose sub-libraries

[1] This chapter is a revised version of an article with the same title, presented at the Workshop on Generic Programming, 1998 [47].

for polytypic functions with more advanced functionality. Examples are the applications in the later chapters: matching, unification, rewriting (Chapter 6), pretty printing, parsing, packing and unpacking (Chapter 7).

5.1 Describing polytypic functions

This section introduces the format that we use for describing polytypic library functions, and gives an overview of the contents of the library.

The description of a polytypic function consists of (some of) the following components: its name and type; an informal description of the function; properties and laws the function satisfies; other names the function is known by; known uses of the function; and its background and relationship to other polytypic functions.

A few related functions at a time are presented as a manual page enclosed in brackets like those surrounding this sentence.

A problem with describing a library of polytypic functions is that it is not completely clear how to *specify* polytypic functions. The most basic combinators have immediate category theoretic interpretations that can be used as a specification, but for more complicated combinators the matter is not all that obvious. Thus, we will normally not provide formal specifications of the library functions, though we try to give references to more in-depth treatments. We also include examples of laws that relate the different functions.

5.1.1 Notation and naming

For the polytypic functions that have Haskell counterparts we prepend the letter p (for polytypic) to the Haskell name to avoid a name clash. The bifunctor variants instead begin with an f. A polytypic function can be thought of as taking (a representation of) a functor as its first argument. This implicit argument is normally omitted but sometimes written as a subscript for clarity: $pmap_d$. Polytypic functions are only defined for regular datatypes $d\ a$. In the type this is indicated by adding a context $Regular\ d \Rightarrow \ldots$, but we will omit this here for brevity. (The implementation of PolyLib in Appendix A contains the full type declarations.)

5.1.2 Library overview

We have divided the library into six parts, as shown in Figure 5.1. The first

pmap, *fmap2*, *cata*
ana, *hylo*, *para*
crush, *fcrush*
(a) Recursion operators

pzip, *fzip*
punzip, *funzip*
pzipWith, *pzipWith'*
pequal, *fequal*
pcompare, *fcompare*
(b) Zips etc.

pmapM, *fmap2M*, *cataM*
anaM, *hyloM*, *paraM*
thread, *fthread*
propagate, *cross*
(c) Monad operators

flatten, *fflatten*
fl_par, *fl_rec*, *substructures*
(d) Flatten functions

psum, *prod*, *comp*, *conc*, *pand*, *por*
size, *flatten*, *flatten'*, *pall*, *pany*, *pelem*
(e) Miscellaneous

Figure 5.1: Overview of PolyLib

part of the library contains powerful recursion combinators such as *pmap*, *cata* and *ana*. This part is the core of the library in the sense that it is used in the definitions of all the functions in the other parts. The second part deals with zips and some derivatives, such as the equality function. The third part consists of functions that manipulate monads. The fourth and fifth parts consist of simpler (but still very useful) functions, like flattening and summing. The following sections describe each of these parts in more detail.

5.2 Recursion operators

$$\begin{aligned} &\mathit{pmap} :: (a \to b) \to d\ a \to d\ b \\ &\mathit{fmap2} :: (a \to c) \to (b \to d) \to f\ a\ b \to f\ c\ d \end{aligned}$$

Function *pmap* takes a function f and a value x of datatype $d\ a$, and applies f recursively to all occurrences of elements of type a in x.

Properties: With d as a functor acting on types, pmap_d is the corresponding functor action on functions. Function $\mathit{fmap2}_f$ is the corresponding functor action for a pattern functor f.

$$\begin{array}{lll}
\mathit{pmap}\ \mathit{id} & === & \mathit{id} \\
\mathit{pmap}\ g \circ \mathit{pmap}\ h & === & \mathit{pmap}\ (g \circ h) \\
\mathit{fmap2}\ \mathit{id}\ \mathit{id} & === & \mathit{id} \\
\mathit{fmap2}\ g\ h \circ \mathit{fmap2}\ i\ j & === & \mathit{fmap2}\ (g \circ i)\ (h \circ j)
\end{array}$$

Also known as: Function *pmap* is often called *map*. In Jay *et al.* [58], *pmap* is called map_1, *fmap2* is called map_2 and, in general, an n-argument map is called map_n. In *charity* [16], $\mathit{pmap}_d\ f\ x$ is written $d\{f\}(x)$.

Known uses: Everywhere! Function *fmap2* is used in the definition of *pmap*, *cata*, *ana*, *hylo*, *para* and in many other PolyLib functions.

Background: The map function was one of the first combinators distinguished in the work of Bird and Meertens [10, 74]. The traditional $\mathit{map} :: (a \to b) \to [\,a\,] \to [\,b\,]$ in functional languages maps a function over a list of elements. Haskell 98 also contains an overloaded version of *map*:

$$\mathit{fmap} :: \mathit{Functor}\ f \Rightarrow (a \to b) \to f\ a \to f\ b$$

Function *fmap* can be used as the polytypic *pmap* if instance declarations for all regular type constructors are given. Function *pmap* can be used to give default instances for the Haskell *fmap*.

$$\begin{array}{lll}
\mathit{cata} & :: & (\Phi_d\ a\ b \to b) \to (d\ a \to b) \\
\mathit{ana} & :: & (b \to \Phi_d\ a\ b) \to (b \to d\ a) \\
\mathit{hylo} & :: & (f\ a\ b \to b) \to (c \to f\ a\ c) \to (c \to b) \\
\mathit{para} & :: & (d\ a \to \Phi_d\ a\ b \to b) \to (d\ a \to b)
\end{array}$$

Four powerful recursion operators on the type $d\ a$: The catamorphism, $\mathit{cata}\ c$, "evaluates" a value of a regular datatype by recursively replacing the constructors with functions. The anamorphism, $\mathit{ana}\ a$, works in the opposite direction and recursively builds a value of a regular datatype from some other data. The hylomorphism, $\mathit{hylo}\ c\ a$, is the generalization of these two functions that simultaneously builds and evaluates a structure. Finally, the paramorphism, $\mathit{para}\ p$, is a generalized form of *cata* that gives its parameter function access not only to the results of evaluating the substructures, but also the structure itself.

Properties: The catamorphism *cata f* satisfies the fusion law (proved in Section 3.10) for every strict h:

$$\begin{array}{ll} & h \circ cata\ f \;=\; cata\ g \\ \Leftarrow & \quad \{\text{ fusion }\} \\ & h \circ f \;=\; g \circ fmap2\ id\ h\ . \end{array}$$

A dual law holds for the anamorphism, and corresponding laws hold for *hylo* and *para*, see Hoogendijk [37]. The hylomorphism can be specified as:

$$hylo\ i\ o \;=\; cata\ i \circ ana\ o$$

Also known as:

PolyLib	Functorial ML [58]	Squiggol	*charity* [16]
cata i	$fold_1\, i$	$(\!\mid i \mid\!)$	$\{\mid i \mid\}$
ana o	-	$[\!(o)\!]$	$(\mid o \mid)$

Functions *cata* and *para* are closely related to the *Visitor* pattern [27].

Known uses: Many polytypic functions can be defined using *cata*: *pmap*, *crush*, *thread*, *flatten*, *propagate*, and many of our applications use it.

Background: The catamorphism, *cata*, is the generalization of the Haskell function *foldr* and the anamorphism, *ana*, is its (category theoretic) dual. Catamorphisms were introduced by Malcolm [71, 72]. A hylomorphism is the fused composition of a catamorphism and an anamorphism. The paramorphism [75], *para*, is the elimination construct for the type $d\ a$ from Martin-Löf type theory [83]. It captures the recursion pattern of primitive recursive functions on the datatype $d\ a$.

$$\begin{array}{lll} crush & :: & (a \to a \to a) \to a \to d\ a \to a \\ fcrush & :: & (a \to a \to a) \to a \to f\ a\ a \to a \end{array}$$

The function $crush\ (\oplus)\ e$ takes a structure x and inserts the operator $(\oplus)$ from left to right between every pair of values of type a at every level in x. (The value e is used in empty leaves.)

Properties: We can push a function f through a crush

$$f \circ \mathit{crush}\ (\oplus)\ e === \mathit{crush}\ (\otimes)\ (f\ \ e) \circ \mathit{pmap}\ f$$

provided f is strict and the following distributive law holds:

$$\forall\ x, y.\ f\ (x \oplus y)\ =\ f\ x \otimes f\ y$$

This can be used to prove that, for an associative operator $(\oplus)$ with unit e:

$$\mathit{crush}\ (\oplus)\ e === \mathit{foldr}\ (\oplus)\ e \circ \mathit{flatten}$$

Known uses: A number of applications of *cruch* within the library are presented in Section 5.6. Many of the functions in that section are, in turn, used in the different applications.

Background: The *crush* operator was first proposed in "Calculate polytypically" by Meertens [76]. As *crush* has the same arguments as *fold* on lists it can be seen as an alternative to *cata* as the generalization of *fold* to regular datatypes.

5.3 Zips

$$\begin{array}{lll}
\mathit{pzip} & :: & (d\ a, d\ b) \to \mathit{Maybe}\ (d\ (a, b)) \\
\mathit{punzip} & :: & d\ (a, b) \to (d\ a, d\ b) \\
\mathit{fzip} & :: & (f\ a\ b, f\ c\ d) \to \mathit{Maybe}\ (f\ (a, c)\ (b, d)) \\
\mathit{funzip} & :: & f\ (a, c)\ (b, d) \to (f\ a\ b, f\ c\ d)
\end{array}$$

Function *punzip* takes a structure containing pairs and splits it up into a pair of structures containing the first and the second components respectively. Function *pzip* is a partial inverse of *punzip*: it takes a pair of structures and zips them together to *Just* a structure of pairs if the two structures have the same shape, and to *Nothing* otherwise.

Properties: Function *punzip* always produces a pair of structures of the same shape, and for such pairs *pzip* always succeeds. Conversely, if *pzip* succeeds, then *punzip* recovers the original pair.

$$\begin{array}{lll}
\mathit{pzip}\ (x, y) == \mathit{Just}\ z & \equiv & (x, y) == \mathit{punzip}\ z \\
\mathit{fzip}\ (x, y) == \mathit{Just}\ z & \equiv & (x, y) == \mathit{funzip}\ z
\end{array}$$

Naturality laws:

$$\begin{array}{lll}
\mathit{mapM}\ (\mathit{pmap}\ (f \twoheadrightarrow g)) \circ \mathit{pzip} & === & \mathit{pzip} \circ (\mathit{pmap}\ f \twoheadrightarrow \mathit{pmap}\ g) \\
\mathit{mapM}\ (\mathit{fmap2}\ (f \twoheadrightarrow g)\ (h \twoheadrightarrow i)) \circ \mathit{fzip} & === & \mathit{fzip} \circ (\mathit{fmap2}\ f\ h \twoheadrightarrow \mathit{fmap2}\ g\ i)
\end{array}$$

Also known as: The zip functions are called zip_m in Jay *et al.* [58] (with $m = 1$ for *pzip* and $m = 2$ for *fzip*), and *pzip* is called **zip.×.d** in Hoogendijk and Backhouse [38].

Known uses: Function *fzip* is used in the definition of *pzipWith*.

Background: The traditional function *zip*

$$zip :: [\,a\,] \to [\,b\,] \to [(a, b)]$$

combines two lists and does not need the *Maybe* type in the result as the longer list can always be truncated. (In general such truncation is possible for all types that have a nullary constructor, but not for all regular types.) A more general ("doubly polytypic") variant of *pzip*: *transpose* (called **zip.d.e** in by Hoogendijk and Backhouse [38])

$$transpose :: d\ (e\ a) \to e\ (d\ a)$$

was first described by Ruehr [94]. For a formal and relational definition, see Hoogendijk and Backhouse [38].

$$\begin{array}{lll} pzipWith & :: & ((a, b) \to Maybe\ c) \to (d\ a, d\ b) \to Maybe\ (d\ c) \\ pzipWith' & :: & (\Phi_d\ c\ e \to e) \to ((d\ a, d\ b) \to e) \to \\ & & ((a, b) \to c) \to (d\ a, d\ b) \to e \end{array}$$

Function *pzipWith* $(\otimes)$ works like *pzip* but uses the operator $(\otimes)$ to combine the values from the two structures instead of just pairing them. As the zip might fail, we also give the operator a chance to signal failure by giving it a *Maybe*-type as a result. The type constructor *Maybe* can be replaced by any monad with a zero, but we didn't want to clutter up the already complicated type with contexts.

Function *pzipWith'* is a generalization of *pzipWith* that can handle two structures of different shape. In the call *pzipWith' ins fail* $(\otimes)$, the operator $(\otimes)$ is used to combine values of the structures as long as the structures have the same shape, *fail* is used to handle the case when the two structures mismatch, and *ins* combines the results from the substructures. (The type of *ins* is the same as the type of the first argument to *cata*.)

Properties: Function *pzip* is just *pzipWith Just* and *pzipWith* is a special case of *pzipWith'*:

$$\begin{array}{lll} pzip & === & pzipWith\ Just \\ pzipWith & === & pzipWith'\ (mapM\ inn \circ fthread)\ (const\ mzero) \end{array}$$

Also known as: Function *pzipWith* is called $zipop_m$ in Jay *et al.* [58].

Known uses: Function *pzipWith'* is used in the definition of polytypic equality and can be used for matching and even unification.

Background: Function *pzipWith* is the polytypic variant of the Haskell function *zipWith*

$$zipWith :: (a \to b \to c) \to [\,a\,] \to [\,b\,] \to [(a, b)]$$

but *pzipWith'* is new.

$$\begin{aligned} pequal &:: (a \to b \to Bool) \to d\ a \to d\ b \to Bool \\ fequal &:: (a \to b \to Bool) \to (c \to d \to Bool) \to f\ a\ c \to f\ b\ d \to Bool \end{aligned}$$

The expression *pequal eq x y* checks whether or not the structures x and y are equivalent using the equivalence operator *eq* to compare the elements pairwise. Function *fequal* is the corresponding equivalence check for the pattern functor level. Function *fequal eqp eqr* performs a top level equivalence check and *pequal eq* a deep equivalence check.

Properties: A partial equivalence relation (a *per*) is a relation that is symmetric and transitive, but not necessarily reflexive. (In *CPO* no interesting relations are reflexive. In fact, if $eq\ \bot\ \bot\ =\ True$, then, by monotonicity, $eq\ x\ y\ =\ True$ for all x and y!) If *eq* is a *per*, then function *pequal eq* is also a *per*.

Known uses: Function *fequal* is used in matching, unification and rewriting to determine when two terms are top level equal. Function *pequal* is used almost everywhere (indirectly through (==)).

Background: An early version of a polytypic equality function was presented by Sheard in 1991 [98]. Function *pequal* can be instantiated to give a default for the Haskell *Eq*-class for regular datatypes:

$$\begin{aligned} (==) &:: (Regular\ d, Eq\ a) \Rightarrow d\ a \to d\ a \to Bool \\ (==) &= pequal\ (==) \end{aligned}$$

In Haskell the equality function can be automatically derived by the compiler, and our polytypic equality is an attempt at moving that derivation out of the compiler into the prelude.

$$
\begin{array}{lll}
pcompare & :: & (a \to a \to Ordering) \to d\ a \to d\ a \to Ordering \\
fcompare & :: & (a \to a \to Ordering) \to (b \to b \to Ordering) \to \\
 & & f\ a\ b \to f\ a\ b \to Ordering
\end{array}
$$

The comparison operators ($(<)$, $(\leqslant)$, etc.) in Haskell are defined in terms of the method *compare* of the *Ord* class.

$$
\begin{array}{l}
\textbf{data}\ Ordering\ =\ LT \mid EQ \mid GT \\
compare :: Ord\ a \Rightarrow a \to a \to Ordering
\end{array}
$$

Function *pcompare* is the polytypic version of *compare*. The expression *pcompare comp x y* compares the structures x and y with lexicographical ordering, using the function *comp* to compare the elements pairwise.

Also known as: Function *pcompare* is called *cmp* by Hinze [35].

Background: Function *pcompare* can be instantiated to give a default for the Haskell *Ord*-class for regular datatypes:

$$
\begin{array}{l}
compare :: Ord\ a \Rightarrow d\ a \to d\ a \to Bool \\
compare\ =\ pcompare\ compare
\end{array}
$$

5.4 Monad operations

$$
\begin{array}{lll}
pmapM & :: & Monad\ m \Rightarrow (a \to m\ b) \to d\ a \to m\ (d\ b) \\
pmapMl & :: & Monad\ m \Rightarrow (a \to m\ b) \to d\ a \to m\ (d\ b) \\
pmapMr & :: & Monad\ m \Rightarrow (a \to m\ b) \to d\ a \to m\ (d\ b) \\
fmap2M & :: & Monad\ m \Rightarrow (a \to m\ c) \to (b \to m\ d) \to f\ a\ b \to m\ (f\ c\ d) \\
cataM & :: & Monad\ m \Rightarrow (\Phi_d\ a\ b \to m\ b) \to (d\ a \to m\ b) \\
anaM & :: & Monad\ m \Rightarrow (b \to m\ \Phi_d\ a\ b) \to (b \to m\ (d\ a)) \\
hyloM & :: & Monad\ m \Rightarrow (f\ a\ b \to m\ b) \to (c \to m\ (f\ a\ c)) \to c \to m\ b \\
paraM & :: & Monad\ m \Rightarrow (d\ a \to \Phi_d\ a\ b \to m\ b) \to d\ a \to m\ b
\end{array}
$$

Function *pmapM* is a variant of *pmap* that threads a monad m from left to right through a structure after applying its function argument to all elements in the structure. Function *pmapMr* is the same but for threading a monad m from right to left through a structure. For symmetry's sake, the library also contains a function *pmapMl*, which is equal to *pmapM*. Furthermore, the library also contains the left and right variants of functions like *cataM* etc. A monadic map can, for example, use a state monad to record information about the elements in the structure during the traversal. The other recursion operators are generalized in the same way to form even more general combinators.

Properties: The monadic map is closely related to *thread* (presented later):

$$\begin{array}{lll} pmapM\ f & === & thread \circ pmap\ f \\ thread & === & pmapM\ id \end{array}$$

There are many more laws as well, but we only give these examples here.

Also known as: Function *pmapM* (*thread*) is called active (passive) traversal in Jay *et al.* [58].

Known uses: Monadic traversals are very useful for data conversion (Chapter 7).

Background: Monadic maps and catamorphisms are described in Fokkinga [25] and monadic anamorphisms and hylomorphisms are defined in Pardo [87]. A category theoretical description of *pmapM* and *thread* can be found in Moggi *et al.* [81].

$$\begin{array}{l} thread\ ::\ Monad\ m \Rightarrow d\ (m\ a) \to m\ (d\ a) \\ fthread\ ::\ Monad\ m \Rightarrow f\ (m\ a)\ (m\ b) \to m\ (f\ a\ b) \end{array}$$

Function *thread* is used to tie together the monad computations in the elements from left to right.

Properties: Function *thread* can be used to define the monadic map, and vice versa:

$$\begin{array}{lll} pmapM\ f & === & thread \circ pmap\ f \\ thread & === & pmapM\ id \end{array}$$

Also known as: Other names for *thread* are $dist_d$ (used by Fokkinga [25]) and *traverse* (used by Moggi *et al.* [81]).

Known uses: Function *thread* can be instantiated (with $d = [\,]$) to the Haskell prelude function

$$sequence :: Monad\ m \Rightarrow [m\ a] \to m\ [a]\ .$$

It can also be instantiated (with $m = Maybe$) to *propagate* and (with $m = [\,]$) to *cross* defined later.

$$\begin{array}{lll} \textit{propagate} & :: & d\ (\textit{Maybe}\ a) \to \textit{Maybe}\ (d\ a) \\ \textit{cross} & :: & d\ [\,a\,] \to [\,d\ a\,] \end{array}$$

Function *propagate* propagates *Nothing* to the top level. Function *cross* is the cross (or tensor) product that given a structure x containing lists, generates a list of structures of the same shape. This list has one element for every combination of values drawn from the lists in x. These two functions can be generalized to *thread* any monad through a value.

Known uses: *propagate* is used in the definition of *pzip*.

Background: Function *propagate* is an instance of *transpose* [94], and both *propagate* and *cross* are instances of *thread*.

5.5 Flatten functions

$$\begin{array}{lll} \textit{flatten} & :: & d\ a \to [\,a\,] \\ \textit{fflatten} & :: & f\ [\,a\,]\ [\,a\,] \to [\,a\,] \\ \textit{fl\_par} & :: & f\ a\ b \to [\,a\,] \\ \textit{fl\_rec} & :: & f\ a\ b \to [\,b\,] \\ \textit{substructures} & :: & d\ a \to [\,d\ a\,] \end{array}$$

Function *flatten* x traverses the structure x and collects all elements from left to right in a list. Functions *fflatten*, *fl_par* and *fl_rec* are variants of this for a pattern functor f. The list *substructures* x contains all substructures of x.

Properties: The free theorem for *flatten*:

$$\textit{flatten} \circ \textit{pmap}\ f === \textit{map}\ f \circ \textit{flatten}$$

Flatten can be defined in terms of $\textit{conc} = \textit{crush}\ (+\!\!+)\ [\,]$ and *pmap*:

$$\textit{flatten} = \textit{conc} \circ \textit{pmap}\ (:[\,])$$

With normal Haskell lists and concatenation this has quadratic asymptotic complexity, but we can apply the standard accumulating parameter trick to obtain a linear implementation: $(\textit{comp} = \textit{crush}\ (\circ)\ \textit{id})$

$$\begin{array}{lll} \textit{flatten}' & = & \textit{comp} \circ \textit{pmap}\ (:) \\ \textit{flatten}\ l & = & \textit{flatten}'\ l\ [\,] \end{array}$$

Functions *conc*, *comp* and *flatten'* are also defined in Section 5.6.

Function *substructures* can be defined in terms of *fl_rec* and *para*:

$$\begin{array}{lcl} \mathit{substructures} & :: & \mathit{Regular}\ d \Rightarrow d\ a \rightarrow [\,d\ a\,] \\ \mathit{substructures} & = & \mathit{para}\ (\lambda x\ y \rightarrow x : \mathit{concat}\ (\mathit{fl\_rec}\ y)) \end{array}$$

Also known as: The integer-indexed family $\mathit{extract}_{m,i}$ defined in Jay *et al.* [58], contains *flatten* when $(m, i) = (1, 0)$, *fl_par* when $(m, i) = (2, 0)$ and *fl_rec* when $(m, i) = (2, 1)$. Another name for *flatten* is **listify** (used in Hoogendijk and Backhouse [38]).

Known uses: Function *fl_rec* is used in the unification algorithm to find the list of immediate subterms of a term.

Background: In the relational theory of polytypism [38] there is a membership relation **mem.d** for every relator (type constructor) **d**. Function *flatten* can be seen as a functional implementation of this relation:

$$a\ \mathbf{mem.d}\ x \equiv a \in\ \mathit{flatten}_d\ x$$

5.6 Miscellaneous

A number of simple polytypic functions can be defined in terms of *crush* and *pmap*. For brevity we present this part of PolyLib by providing only the name, the type and the definition of each function.

$$\begin{array}{lcl} \mathit{psum} & :: & d\ \mathit{Int} \rightarrow \mathit{Int} \\ \mathit{prod} & :: & d\ \mathit{Int} \rightarrow \mathit{Int} \\ \mathit{comp} & :: & d\ (a \rightarrow a) \rightarrow (a \rightarrow a) \\ \mathit{conc} & :: & d\ [\,a\,] \rightarrow [\,a\,] \\ \mathit{pand} & :: & d\ \mathit{Bool} \rightarrow \mathit{Bool} \\ \mathit{por} & :: & d\ \mathit{Bool} \rightarrow \mathit{Bool} \end{array}$$

$$\begin{array}{lcl} \mathit{psum} & = & \mathit{crush}\ (+)\ 0 \\ \mathit{prod} & = & \mathit{crush}\ (*)\ 1 \\ \mathit{comp} & = & \mathit{crush}\ (\circ)\ \mathit{id} \\ \mathit{conc} & = & \mathit{crush}\ (+\!\!+)\ [\,] \\ \mathit{pand} & = & \mathit{crush}\ (\,\wedge\,)\ \mathit{True} \\ \mathit{por} & = & \mathit{crush}\ (\,\vee\,)\ \mathit{False} \end{array}$$

All these are defined using *crush* only, and by combining *crush* and *pmap* we immediately get a few more useful functions.

$$
\begin{array}{lll}
\mathit{size} & :: & d\ a \to \mathit{Int} \\
\mathit{flatten} & :: & d\ a \to [\,a\,] \\
\mathit{flatten}' & :: & d\ a \to [\,a\,] \to [\,a\,] \\
\mathit{pall} & :: & (a \to \mathit{Bool}) \to d\ a \to \mathit{Bool} \\
\mathit{pany} & :: & (a \to \mathit{Bool}) \to d\ a \to \mathit{Bool} \\
\mathit{pelem} & :: & \mathit{Eq}\ a \Rightarrow\ a \to d\ a \to \mathit{Bool}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{size} & = & \mathit{psum} \circ \mathit{pmap}\ (\lambda\_ \to 1) \\
\mathit{flatten} & = & \mathit{conc} \circ \mathit{pmap}\ (\lambda x \to [\,x\,]) \\
\mathit{flatten}' & = & \mathit{comp} \circ \mathit{pmap}\ (:) \\
\mathit{pall}\ p & = & \mathit{pand} \circ \mathit{pmap}\ p \\
\mathit{pany}\ p & = & \mathit{por} \circ \mathit{pmap}\ p \\
\mathit{pelem}\ x & = & \mathit{pany}\ (\lambda y \to x == y)
\end{array}
$$

5.7 Conclusions

We have given a description of PolyLib: the library of PolyP. This library has grown out of our experience with implementing polytypic functions. PolyLib is very likely incomplete, but we think we have included most basic polytypic combinators. We have used PolyLib in the construction of special purpose sub-libraries for matching, unification and rewriting (Chapter 6) and data conversion (Chapter 7), and some of these applications will be included in future versions of PolyLib. Both PolyP and PolyLib are available from the author's homepage.

Chapter 6

Rewriting[1]

Abstract

Given any value of a datatype (an algebra of terms), and rules to rewrite values of that datatype, we want a function that rewrites the value to normal form if the value is normalizable. This chapter develops a polytypic rewriting function that uses the parallel innermost rewriting strategy. It improves upon our earlier work on polytypic rewriting in two fundamental ways. Firstly, the rewriting function uses a term interface that hides the polytypic part from the rest of the program. The term interface is a framework for polytypic programming on terms. This implies that the rewriting function is independent of the particular implementation of polytypism. We give several functions and laws on terms, which simplify calculating with programs. Secondly, the rewriting function is developed together with a correctness proof.

6.1 Introduction

A term rewriting system is an algebra (a datatype of terms) together with a set of rewrite rules. The rewrite rules describe how to rewrite the terms of the algebra. A rewrite rule is a pair (lhs, rhs) of terms containing variables with the interpretation that any term that matches the left hand side (lhs) may be rewritten to the right hand side (rhs) with the variables replaced by the matches from the left hand side.

[1]This chapter is a revised and extended version of the article "A framework for polytypic programming on terms, with an application to rewriting", Workshop on Generic Programming, 2000 [52].

6.1.1 An example rewriting system

An example of a term datatype is the type *Expr*:

$$\begin{array}{lcl}
\textbf{data}\ \mathit{Expr} & = & \mathit{EVar}\ \mathit{Int} \mid Z \mid S\ \mathit{Expr} \mid \mathit{Expr}\ {:}{+}{:}\ \mathit{Expr} \mid \mathit{Expr}\ {:}{*}{:}\ \mathit{Expr} \\
\textbf{type}\ \mathit{Rule}\ t & = & (t, t) \\
\mathit{plusZero} & :: & \mathit{Rule}\ \mathit{Expr} \\
\mathit{plusZero} & = & (x\ {:}{+}{:}\ Z, x) \\
\quad \textbf{where}\ x & = & \mathit{EVar}\ 0
\end{array}$$

For example, with the rule *plusZero* the left hand side x :+: Z matches the expression $S\ Z$:+: Z with the substitution $\{x \mapsto S\ Z\}$. Thus the rewritten term is the right hand side x after the substitution is applied: $S\ Z$. To introduce the notation we can express this in Haskell syntax: the following expression evaluates to *True*.

$$\begin{array}{lcl}
\textbf{let}\ (\mathit{lhs}, \mathit{rhs}) & = & \mathit{plusZero} \\
\quad\ \mathit{Just}\ s & = & \mathit{match}\ \mathit{lhs}\ (S\ Z\ {:}{+}{:}\ Z) \\
\multicolumn{3}{l}{\textbf{in}\ \mathit{appSubst}\ s\ \mathit{rhs} == S\ Z}
\end{array}$$

The functions involved are the following:

$$\begin{array}{lcl}
\mathit{appSubst} & :: & \mathit{Term}\ t \Rightarrow \mathit{Sub}\ t \to t \to t \\
\mathit{match} & :: & \mathit{Term}\ t \Rightarrow t \to t \to \mathit{Maybe}\ (\mathit{Sub}\ t) \\
(==) & :: & \mathit{Term}\ t \Rightarrow t \to t \to \mathit{Bool}
\end{array}$$

Function *appSubst* takes a substitution and a term, and applies the substitution to the term. The type *Expr* is an instance of a type class for *Term*s defined in Section 6.2.1. The definitions of *appSubst* and the type constructor for *Sub*stitutions are given in Section 6.3.1. Function *match* (defined in Section 6.3.2) takes a term containing variables, and a term without variables, and returns *Just* a substitution s if the terms can be matched by means of s, and *Nothing* otherwise. The operator (==) is the Haskell equality operator, defined for terms in Section 6.2.3.

A rule set is a collection of rules, and a rule set matches a term if at least one of the rules matches that term. To keep the system deterministic, even when more than one rule matches, we order the rules and always use the first match. In practice this means that our rule set is a rule list.

$$\begin{array}{lcl}
\textbf{type}\ \mathit{Rules}\ t & = & [\mathit{Rule}\ t] \\
\mathit{exprrules} & :: & \mathit{Rules}\ \mathit{Expr} \\
\mathit{exprrules} & = & [\mathit{plusZero}, \mathit{plusSucc}, \mathit{timesZero}, \mathit{timesSucc}] \\
\quad \textbf{where}\ \mathit{plusZero} & = & (x\ {:}{+}{:}\ Z, x) \\
\qquad\qquad \mathit{plusSucc} & = & (x\ {:}{+}{:}\ S\ y, S\ (x\ {:}{+}{:}\ y)) \\
\qquad\qquad \mathit{timesZero} & = & (x\ {:}{*}{:}\ Z, Z) \\
\qquad\qquad \mathit{timesSucc} & = & (x\ {:}{*}{:}\ S\ y, (x\ {:}{*}{:}\ y)\ {:}{+}{:}\ x) \\
\qquad\qquad (x, y) & = & (\mathit{EVar}\ 0, \mathit{EVar}\ 1)
\end{array}$$

Function *rewrite* (defined in Section 6.4.2) rewrites a term to normal form by repeatedly applying rules from a rule list:

$$\mathit{rewrite} :: \mathit{Term}\ t\ \Rightarrow\ \mathit{Rules}\ t \to t \to t$$

Because the rule list *exprrules* is normalizing, function *rewrite* will rewrite any expression of type *Expr* to normal form. In general, *rewrite rs t* terminates if and only if the term t is normalizing with respect to the rule list *rs*.

6.1.2 Polytypic rewriting

For other kinds of terms, rewriting behaves exactly as on expressions. We would like to have a *polytypic* rewriting function: a rewriting function that can be applied to any kind of terms.

This chapter develops a polytypic rewriting function that uses the parallel innermost rewriting strategy. We have chosen the parallel innermost rewriting strategy because this lets us transform the rewriting function into an asymptotically optimal solution. The results in this chapter improve upon our earlier work on polytypic rewriting [62] in two fundamental ways.

Firstly, the program uses an interface that hides the polytypic part from the rest of the program. The term interface is a framework for polytypic programming on terms. We assume that we have a type of terms, on which several functions, such as a function that determines whether or not a term is a variable and a function that returns the children of a term, are defined. The rewriting function (including functions for matching and for applying a substitution) uses just these functions on terms. This idea was also present in our previous work [2, 51], but it was only applied to unification. It turns out that the same interface for terms can be used for matching and term rewriting. We also introduce some combinators on terms such as *mapTerm*, which maps a function over all variables in a term, and *bup* which applies a term transformer bottom up to all levels of a term. Furthermore, to facilitate calculating with programs, we give a number of laws for these functions. Programming against an interface for terms implies that our rewriting functions are independent of the particular implementation of polytypism, so that we can use our rewriting functions in future polytypic programming languages such as Generic Haskell [33] too.

Secondly, the program is developed together with a correctness proof, which says that our rewriting function rewrites any normalizable term to normal form. A specification of rewriting is transformed in a few steps into an efficient rewriting function. We prove that the transformation steps are semantics preserving.

This chapter is organized as follows. Section 6.2 introduces terms, combinators on terms, and laws for these combinators. Section 6.3 gives three applications of terms: substitutions, matching and unification. Section 6.4 specifies and implements polytypic functions for rewriting and states the theorems they satisfy. Section 6.5 contains detailed proofs of some of the theorems. Section 6.6 concludes.

6.2 A term interface

This section introduces an interface for terms and shows that every regular datatype supports this interface. Furthermore, it defines a few combinators that work on terms, and states some laws that relate these combinators. The proofs of these laws are presented in Section 6.5.

6.2.1 Terms

This subsection defines a Haskell class for types that can be used as terms for matching, unification and in a term rewriting system. A careful analysis of the properties we need from terms reveals that

- a term has (updatable) children,
- two terms can be tested for top level equality,
- and a term can be a variable.

Each of these requirements is captured in a class and the class of terms is the intersection of these requirements.

$$\textbf{class}\ (\mathit{Children}\ t, \mathit{TopEq}\ t, \mathit{VarCheck}\ t) \Rightarrow\ \mathit{Term}\ t$$

In the following subsections we will define the three classes *Children*, *TopEq* and *VarCheck* together with the laws we require from the instances to make the rewriting proofs go through later.

Children

The children (immediate subterms) of a term can be extracted or mapped over.

$$\begin{array}{llll}\textbf{class}\ \mathit{Children}\ t\ \textbf{where} & \mathit{children} & :: & t \rightarrow [\,t\,] \\ & \mathit{mapC} & :: & (t \rightarrow t) \rightarrow t \rightarrow t\end{array}$$

The functions *children* and *mapC* should be related by the following law:

$$children \circ mapC\ f === map\ f \circ children$$

Function *mapC* should preserve identities and composition:

$$\begin{array}{lcl} mapC\ id & = & id \\ mapC\ (f \circ g) & = & mapC\ f \circ mapC\ g \end{array}$$

Top level equality

Function *topEq* is a shallow equality test. A typical *topEq* checks if two terms have the same outermost constructor.

$$\textbf{class}\ TopEq\ t\ \textbf{where}\ topEq :: t \to t \to Bool$$

We require *topEq* to be almost an equivalence relation:

$$\begin{array}{l} \neg\,(x\ =\ \bot) \Rightarrow \lfloor topEq\ x\ x \rfloor \\ \lfloor topEq\ x\ y \rfloor \Rightarrow \lfloor topEq\ y\ x \rfloor \\ \lfloor topEq\ x\ y \rfloor \Rightarrow (\lfloor topEq\ y\ z \rfloor \Rightarrow \lfloor topEq\ x\ z \rfloor) \end{array}$$

It should not depend on the children:

$$\lfloor topEq\ x\ y \rfloor\ \equiv\ \lfloor topEq\ x\ (mapC\ f\ y) \rfloor$$

And the number of children should be part of the top level:

$$\lfloor topEq\ x\ y \rfloor \Rightarrow \lfloor length\ (children\ x) == length\ (children\ y) \rfloor$$

Checking for variables

We model variables with the type *Var* (any type with equality would do), and it should be possible to check whether or not a term is a variable, and if it is, which variable.

$$\begin{array}{l} \textbf{newtype}\ Var\ =\ MkVar\ Int\ \textbf{deriving}\ Eq \\ \textbf{class}\ VarCheck\ t\ \textbf{where}\ varCheck :: t \to Maybe\ Var \end{array}$$

If a term is a variable, then it cannot have children.

$$\lfloor varCheck\ t == Just\ v \rfloor \Rightarrow \lfloor children\ t == [\,] \rfloor$$

6.2.2 Polytypic *Term* instances

In this subsection we show that all *Regular* datatypes are, in fact, *Term*s. We do this by defining polytypic instances for *children*, *mapC*, *topEq* and *varCheck*.

Functions *children* and *mapC*

Function $children :: Children\ t \Rightarrow t \rightarrow [t]$ returns the immediate subterms of a term. We find these subterms by unfolding the term one level, using *out*, mapping the parameters to empty lists and the subterms to singletons using *fmap2* and flattening the result to a list using *fflatten*:

$$\begin{array}{l} \textbf{instance}\ Regular\ d \Rightarrow Children\ (d\ a)\ \textbf{where} \\ \quad children \ = \ fflatten \circ fmap2\ (const\ [\,])\ (:[\,]) \circ out \\ \quad mapC\ f \ = \ inn \circ fmap2\ id\ f \circ out \end{array}$$

Function $fflatten :: f\ [a]\ [a] \rightarrow [a]$ takes a value v of type $f\ [a]\ [a]$, and returns the concatenation of all the lists (of type $[a]$) occurring in v. The polytypic definition of *fflatten* was given in Section 3.9.

Function *topEq*

Function $topEq :: TopEq\ t \Rightarrow t \rightarrow t \rightarrow Bool$ compares the top level of two terms for equality. It is defined in terms of the polytypic equality function *fequal* described in PolyLib. The first argument to *fequal* compares parameters for equality, the second argument (which compares the subterms) is constantly true (to get top level equality) and the third and fourth arguments are the two (unfolded) terms to be compared:

$$\begin{array}{l} \textbf{instance}\ (Regular\ d, Eq\ a) \Rightarrow TopEq\ (d\ a)\ \textbf{where} \\ \quad topEq\ t\ t' \ = \ fequal\ (==)\ (\lambda\_\ \_ \rightarrow True)\ (out\ t)\ (out\ t') \end{array}$$

Function *varCheck*

Function $varCheck :: VarCheck\ t \Rightarrow t \rightarrow Maybe\ Var$ checks whether or not a term is a variable. A polytypic *varCheck* must recognize the datatype constructor that represents variables, using only information about the structure of the datatype. We have for simplicity chosen to represent variables by the first constructor in the datatype, which should have one parameter of type *Var*.

$$\begin{array}{l} \textbf{instance}\ Regular\ d \Rightarrow VarCheck\ (d\ a)\ \textbf{where} \\ \quad varCheck \ = \ fvarCheck \circ out \end{array}$$

$$
\begin{array}{lll}
\textbf{polytypic}\ \mathit{fvarCheck} & :: & f\ a\ b \rightarrow \mathit{Maybe}\ \mathit{Var} \\
\quad = \textbf{case}\ f\ \textbf{of} & & \\
\qquad g + h & \rightarrow & \mathit{fvarCheck} \triangledown \mathit{const}\ \mathit{Nothing} \\
\qquad \mathit{Const}\ \mathit{Var} & \rightarrow & \mathit{Just} \\
\qquad g & \rightarrow & \mathit{const}\ \mathit{Nothing}
\end{array}
$$

Summary

We have made all regular datatypes instances of the class *Term*. Thus all applications written using only *Term* operations are automatically polytypic in the sense that they can be used with PolyP. Of course any such application could have used PolyP directly, but restricting the use of polytypic functions to a minimal interface (the class *Term*), makes the code more reusable and opens it up for experimentation with alternative implementations of polytypism.

6.2.3 Combinators on terms

In this section we define a few general purpose functions on terms. A first example is the function *size* that calculates the number of nodes in a term.

$$
\begin{array}{l}
\mathit{size} :: \mathit{Children}\ t \Rightarrow t \rightarrow \mathit{Int} \\
\mathit{size}\ t \ = \ 1 + \mathit{sum}\ (\mathit{map}\ \mathit{size}\ (\mathit{children}\ t))
\end{array}
$$

Using *children* we can easily extend the top level equality to deep equality:

$$
\begin{array}{l}
(==) :: (\mathit{TopEq}\ t, \mathit{Children}\ t) \Rightarrow t \rightarrow t \rightarrow \mathit{Bool} \\
x == y \ = \ \mathit{topEq}\ x\ y \wedge \mathit{and}\ (\mathit{zipWith}\ (==)\ (\mathit{children}\ x)\ (\mathit{children}\ y))
\end{array}
$$

If *topEq* is almost an equivalence relation (as defined in section 6.2.1), then $(==)$ is an equivalence relation for all finite terms.

A simple application of the equality check is to define a predicate *fixedBy* f that is true for the set of fixed points of f:

$$
\begin{array}{l}
\mathit{fixedBy} :: (\mathit{TopEq}\ t, \mathit{Children}\ t) \Rightarrow (t \rightarrow t) \rightarrow t \rightarrow \mathit{Bool} \\
\mathit{fixedBy}\ f\ x \ = \ x == f\ x
\end{array}
$$

Function *bup* f applies a term transformer at all levels of a term bottom up. It is as close we can get to a generic catamorphism for types in the *Children* class. Function *bup* is more restricted than a normal catamorphism as the output is always of the same type as the input, but it is sufficient to specify and implement rewriting.

$$\begin{array}{l} \mathit{bup} :: \mathit{Children}\ t \Rightarrow (t \to t) \to t \to t \\ \mathit{bup}\ f\ =\ f \circ \mathit{mapC}\ (\mathit{bup}\ f) \end{array}$$

Function *mapTerm* is one possible generic map function for *Term*s with variables. The application *mapTerm s* maps s over all variables in a term, leaving the rest of the structure unchanged. It is implemented in terms of the more general function *foldTerm p s* that also applies the function p to post-process the results from the children. Function *foldTerm* can be seen as the combination of a *bup* (a catamorphism) and a map.

$$\begin{array}{lcl} \mathit{mapTerm} & :: & \mathit{Term}\ t \Rightarrow (\mathit{Var} \to \mathit{Maybe}\ t) \to t \to t \\ \mathit{mapTerm}\ s & = & \mathit{foldTerm}\ \mathit{id}\ s \\ \mathit{foldTerm} & :: & \mathit{Term}\ t \Rightarrow (t \to t) \to (\mathit{Var} \to \mathit{Maybe}\ t) \to t \to t \\ \mathit{foldTerm}\ p\ s\ t & = & \mathit{maybe}\ (p\ (\mathit{mapC}\ (\mathit{foldTerm}\ p\ s)\ t)) \\ & & \qquad\qquad (\mathit{maybe}\ (p\ t)\ \mathit{id} \circ s) \\ & & \qquad\qquad (\mathit{varCheck}\ t) \end{array}$$

Function *foldTerm* traverses all nodes in a term containing variables bottom up. If a node is a variable, then it is replaced by the term to which that variable is bound in the finite map s, or transformed by p if it is not bound by s. If a node is not a variable, then *foldTerm* is applied recursively to the children (if any) and the result is transformed by p.

6.2.4 Laws for term combinators

Using the properties required for the functions from the *Term* class we can derive a number of laws for the term combinators. The proofs of these laws are given in Section 6.5. The theorems for *bup* are restricted to finite terms as captured by the predicate *fin*:

Definition 6.1 *Finite terms:*

$$\begin{array}{lcl} \mathit{fin} & :: & \mathit{Children}\ t \Rightarrow t \to \mathit{Bool} \\ \mathit{fin} & = & \mathit{fix}\ \mathit{deeper} \\ \mathit{deeper} & :: & \mathit{Children}\ t \Rightarrow (t \to \mathit{Bool}) \to (t \to \mathit{Bool}) \\ \mathit{deeper}\ p & = & \mathit{all}\ p \circ \mathit{children} \end{array}$$

The predicate *fin* is defined as the fixed point of *deeper* or equivalently as the limit of a chain of approximations $deeper^n \perp$. The predicate $deeper^n \perp$ is true for all terms of depth less than n. Proofs by fixed point induction using *fin* and *deeper* are closely related to proofs using the generic approximation lemma by Hutton and Gibbons [44].

Using fixed point induction we can prove a characterization of *bup*:

Theorem 6.2 *bup-characterization:*

$$(f \overset{fin}{===} g \circ mapC\ f) \;\equiv\; (f \overset{fin}{===} bup\ g)$$

If we let $f === g === id$ in the *bup*-characterization theorem, then the premise $id \overset{fin}{===} id \circ mapC\ id$ follows trivially from the requirement $mapC\ id === id$ of the class *Children*. Thus we get the following corollary to *bup*-characterization:

Corollary 6.3 *bup-identity:*

$$id \overset{fin}{===} bup\ id$$

A law similar to the *fix*-equality law but for proving equality of functions defined using *bup* is an easy consequence of *bup*-characterization:

Theorem 6.4 *bup-equality:*

$$\begin{aligned}&(g \circ mapC\ f \overset{fin}{===} h \circ mapC\ f) \;\equiv\; (bup\ g \overset{fin}{===} bup\ h)\\ &\quad \textbf{where}\ f \;=\; bup\ g\end{aligned}$$

Function *bup* is closely related to *foldTerm*; both traverse the term bottom up, but *bup* does not distinguish variables from other (sub)terms. The behavior of *bup* can be simulated by *foldTerm* if the substitution argument does *Nothing* for all variables:

Theorem 6.5 *bup is a foldTerm:*

$$foldTerm\ f\ (const\ Nothing) \overset{fin}{===} bup\ f$$

The final theorem of this section says that we can fuse the composition of a bottom-up traversal with a *mapTerm s*, where s is a function that maps variables to *Maybe* some value, into a *foldTerm*, provided that the bottom-up traversal is the identity on the result of s.

Theorem 6.6 *bup-mapTerm-fusion:*

$$\lfloor mapM\ (bup\ f) \circ s === s \rfloor \Rightarrow (foldTerm\ f\ s \overset{fin}{===} bup\ f \circ mapTerm\ s)$$

6.3 Substitutions, matching and unification

This section presents three applications expressed in terms of the methods of the *Term* class: substitutions, matching and unification. Substitutions and matching are used in the following section on rewriting.

6.3.1 Substitutions

A substitution is a mapping from variables to terms that changes only a finite number of variables. As the concrete representation of substitutions is irrelevant for the definition of rewriting, we use an abstract datatype *Sub t* for finite maps from variables to terms.

$$\begin{array}{lll} \mathit{idSubst} & :: & \mathit{Sub}\ t \\ \mathit{modBind} & :: & (\mathit{Var}, t) \rightarrow \mathit{Sub}\ t \rightarrow \mathit{Sub}\ t \\ \mathit{lookupIn} & :: & \mathit{Sub}\ t \rightarrow \mathit{Var} \rightarrow \mathit{Maybe}\ t \end{array}$$

This could be implemented as a constructor class in Haskell, but we avoid that because we don't want to clutter up the types with an extra type context. The value *idSubst* represents the identity substitution, the call *modBind* (v, t) s modifies the substitution s to bind v to t (leaving the bindings for other variables unchanged) and *lookupIn* s v looks up the variable v in the substitution s, giving *Nothing* if the variable is not bound in s.

Using *lookupIn* a substitution can be viewed as a function from *variables* to terms. To use substitutions as functions from *terms* to terms we define *appSubst*:

$$\begin{array}{l} \mathit{appSubst} :: \mathit{Term}\ t \Rightarrow \mathit{Sub}\ t \rightarrow t \rightarrow t \\ \mathit{appSubst}\ s\ =\ \mathit{mapTerm}\ (\mathit{lookupIn}\ s) \end{array}$$

We can also define a variant of *appSubst* that does the equivalent of a bottom-up traversal with f after the substitution has been applied. A straightforward implementation would be the following:

$$\begin{array}{l} \mathit{fromVarsUpAfterSubst} :: \mathit{Term}\ t \Rightarrow (t \rightarrow t) \rightarrow (\mathit{Sub}\ t, t) \rightarrow t \\ \mathit{fromVarsUpAfterSubst}\ f\ (s, t)\ =\ \mathit{bup}\ f\ (\mathit{appSubst}\ s\ t) \end{array}$$

Instead, we use a simple corollary of *bup*-*mapTerm*-fusion (Theorem 6.6) to obtain a more efficient definition (for some combinations of f and s).

$$\begin{array}{l} \mathit{fromVarsUpAfterSubst} :: \mathit{Term}\ t \Rightarrow (t \rightarrow t) \rightarrow (\mathit{Sub}\ t, t) \rightarrow t \\ \mathit{fromVarsUpAfterSubst}\ f\ (s, t)\ =\ \mathit{foldTerm}\ f\ (\mathit{lookupIn}\ s)\ t \end{array}$$

Corollary 6.7 *bup-appSubst-fusion:*

$$\lfloor mapM\ (bup\ f) \circ lookupIn\ s === lookupIn\ s \rfloor$$
$$\Rightarrow \lfloor fromVarsUpAfterSubst\ f\ (s, t) == bup\ f\ (appSubst\ s\ t) \rfloor$$

For example, if $bup\ f$ is an implementation of rewriting to normal form and the substitution binds all variables to terms in normal form, then the condition is satisfied.

6.3.2 Matching

Matching a pattern p with a term t yields *Just* a substitution s such that $appSubst\ s\ p == t$ or, if no such substitution exists, then the matching fails with *Nothing*. Both the pattern and the term may contain variables, but the matching only allows variables in the pattern to be instantiated — any variable in the term is treated as a term constant. Function *match* is defined in terms of $match'$ that carries around a current substitution, starting with the identity substitution.

```
match :: Term t => t -> t -> Maybe (Sub t)
match' :: Term t => t -> t -> Sub t -> Maybe (Sub t)

match p t = match' p t idSubst
match' p t s = maybe no yes (varCheck p)
  where no = if topEq p t then
                 threadList (zipWith match' (children p) (children t)) s
             else
                 Nothing
        yes v = Just (modBind (v, t) s)
```

We assume that the patterns are linear - that is, no variable occurs twice in the same pattern. It is easy to extend this definition to work in the presence of nonlinear patterns; we do not, however, include the details here.

The utility functions *threadList* and (@@) compose monadic functions in sequence.

```
threadList :: Monad m => [a -> m a] -> (a -> m a)
threadList =  foldr (@@) return
(@@)       :: Monad m => (a -> m b) -> (c -> m a) -> (c -> m b)
f @@ g     =  \x -> g x >>= f
```

6.3.3 Unification

The unification algorithm described in this section is included for completeness only and it is not used in the rewriting algorithm. The reader may skip this section without loss of continuity.

If we change the pattern matching function from Section 6.3.2 to allow also variables in the second term to match expressions in the first term, thus making the matching symmetric, then we obtain unification. A unification algorithm tries to find a *most general unifier* (*mgu*) of two terms. A most general unifier of two terms is a smallest substitution of terms for variables such that the substituted terms become equal. (If two first order terms are unifiable, then their *mgu* is unique up to renaming [93].) Use of unification is widespread; it is used in type inference algorithms, rewriting systems, compilers, etc. (see the survey by Knight [68]).

Descriptions of unification algorithms normally deal with a general datatype of terms, containing variables and applications of constructors to terms, but each real implementation uses one specific instance of terms and a specialized version of the algorithm for this term type. This section describes a functional unification program that works for all regular term types. However, we do not prove that it is a correct implementation of unification.

Substitutions and unifiers

A unifier of two terms is a substitution that makes the terms equal. We start with an example. Consider the unification of the two terms $F(x, F(A, B))$ and $F(G(y, A), y)$, where x and y are variables and F, G, A and B are term constructors. Because both terms have an F on the outermost level, these expressions can be unified if x can be unified with $G(y, A)$, and $F(A, B)$ can be unified with y. As these two pairs of terms are unified by the unifier $\sigma = \{x \mapsto G(y, A), y \mapsto F(A, B)\}$, the original pair of terms is also unified by applying the unifier σ, yielding the unified term $F(G(F(A, B), A), F(A, B))$.

As the example shows we use a slightly different variant of *appSubst* for unification than the one used for matching:

$$\begin{array}{lcl} \mathit{appSubst} & :: & \mathit{Term}\ t \Rightarrow \mathit{Sub}\ t \to t \to t \\ \mathit{appSubst}\ s & = & \mathit{mapTerm}\ (\mathit{mapM}\ (\mathit{appSubst}\ s) \circ \mathit{lookupIn}\ s) \end{array}$$

When calling *appSubst* s t, the substitution s is applied to all variables in the term t as in the version for matching, but here s is also applied recursively to all variables in the substituted terms. A substitution σ is at least as general as a substitution σ' if and only if σ' can be factored by σ, that is, if there exists a substitution ρ such that $\mathit{appSubst}\ \sigma' = \mathit{appSubst}\ \rho \circ \mathit{appSubst}\ \sigma$.

We want to define a function that given two terms finds a most general unifier that unifies the terms or, if the terms are not unifiable, reports this.

The unification algorithm

Function *unify* takes two terms, and returns their most general unifier. It is implemented in terms of *unify'*, which updates a current substitution that is passed around as an extra argument. The unification algorithm starts with the

$$
\begin{array}{ll}
\mathit{unify} :: \mathit{Term}\ t \Rightarrow\ t \to t & \to\ \ \mathit{Maybe}\ (\mathit{Sub}\ t) \\
\mathit{unify}' :: \mathit{Term}\ t \Rightarrow\ t \to t \to \mathit{Sub}\ t & \to\ \ \mathit{Maybe}\ (\mathit{Sub}\ t)
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{unify}\ tx\ ty\ =\ \mathit{unify}'\ tx\ ty\ \mathit{idSubst}} \\
\multicolumn{3}{l}{\mathit{unify}'\ tx\ ty\ s\ =\ \mathit{uni}\ (\mathit{varCheck}\ tx, \mathit{varCheck}\ ty)\ \mathbf{where}} \\
\quad \mathit{uni}\ (\mathit{Nothing}, \mathit{Nothing})\ \mid \mathit{topEq}\ tx\ ty & = & \mathit{uniTerms}\ tx\ ty\ s \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} & = & \mathit{Nothing} \\
\quad \mathit{uni}\ (\mathit{Just}\ i, \mathit{Just}\ j)\ \mid i == j & = & \mathit{Just}\ s \\
\quad \mathit{uni}\ (\mathit{Just}\ i, \_) & = & (i \mapsto ty)\ s \\
\quad \mathit{uni}\ (\_, \mathit{Just}\ j) & = & (j \mapsto tx)\ s
\end{array}
$$

$$
\begin{array}{l}
\mathit{uniTerms} :: \mathit{Term}\ t \Rightarrow\ t \to t \to \mathit{Sub}\ t \to \mathit{Maybe}\ (\mathit{Sub}\ t) \\
\mathit{uniTerms}\ x\ y\ =\ \mathit{threadList}\ (\mathit{zipWith}\ \mathit{unify}'\ (\mathit{children}\ x)\ (\mathit{children}\ y))
\end{array}
$$

$$
\begin{array}{l}
(\mapsto) :: \mathit{Term}\ t \Rightarrow\ \mathit{Var} \to t \to \mathit{Sub}\ t\quad \to\quad \mathit{Maybe}\ (\mathit{Sub}\ t) \\
(i \mapsto t)\ s\ =\ \mathbf{if}\ \mathit{occursCheck}\ i\ s\ t\ \mathbf{then}\ \mathit{Nothing} \\
\qquad\qquad\quad \mathbf{else\ case}\ \mathit{lookupIn}\ s\ i\ \mathbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Nothing} \qquad \to\quad \mathit{Just}\ (\mathit{modBind}\ (i, t)\ s) \\
\qquad\qquad\qquad\qquad \mathit{Just}\ t' \qquad\ \ \to\quad \mathit{unify}'\ t\ t'\ s
\end{array}
$$

Figure 6.1: The core of the unification algorithm

identity substitution, traverses the terms and tries to update the substitution (as little as possible) while solving the constraints found. If this succeeds, then the resulting substitution is a most general unifier of the terms. The algorithm distinguishes three cases depending on whether or not the terms are variables.

- If none of the terms is a variable, then we have two sub-cases; either the constructors of the terms are different (that is, the terms are not top level equal) and unification fails, or the constructors are equal and we unify all the children pairwise.

$$
\begin{array}{l}
\mathit{vars} :: (\mathit{Children}\ t, \mathit{VarCheck}\ t) \Rightarrow t \to [\,\mathit{Var}\,] \\
\mathit{vars}\ t \;=\; [v \mid \mathit{Just}\ v \leftarrow \mathit{map}\ \mathit{varCheck}\ (\mathit{subTerms}\ t)] \\
\\
\mathit{subTerms} :: \mathit{Children}\ t \Rightarrow t \to [\,t\,] \\
\mathit{subTerms}\ t \;=\; t : \mathit{concat}\ (\mathit{map}\ \mathit{subTerms}\ (\mathit{children}\ t)) \\
\\
\mathit{occursCheck} :: \mathit{Term}\ t \Rightarrow \mathit{Var} \to \mathit{Sub}\ t \to t \to \mathit{Bool} \\
\mathit{occursCheck}\ i\ s\ t \;=\; i \in \mathit{reachlist}\ (\mathit{vars}\ t) \\
\quad \mathbf{where} \\
\qquad \mathit{reachlist}\ l \;=\; l \mathbin{+\!\!\!+} \mathit{concat}\ (\mathit{map}\ \mathit{reachable}\ l) \\
\qquad \mathit{reachable}\ v \;=\; \mathit{reachlist}\ (\mathit{maybe}\ [\,]\ \mathit{vars}\ (\mathit{lookupIn}\ s\ v))
\end{array}
$$

Figure 6.2: Auxiliary functions in the unification algorithm

- If both terms are variables and the variables are equal, then we succeed without changing the substitution. (If the variables are not equal, then the following case matches.)
- If one of the terms is a variable, then we try to add the binding of this variable to the other term, to the substitution. This succeeds if the variable does not occur in the term and if the new binding of the variable can be unified with the old binding (in the current substitution).

A straightforward implementation of this description gives the code in Figure 6.1 using the auxiliary functions in Figure 6.2. Part of a correctness proof of this implementation can be found in the introduction to generic programming from the summer school on Advanced Functional Programming 1998 [2].

6.4 Rewriting

This section specifies polytypic rewriting by means of a clearly correct, but inefficient function. The specification can be transformed, using the proof tools for fixed points and terms presented in the previous sections, in a number of steps into an efficient rewriting function.

We start in Section 6.4.1 with defining a function *rewrite_step*, which performs a single rewrite step on a term. Function *rewrite_step* is then used in a specification (a clearly correct, but very inefficient version) of function *rewrite* in Section 6.4.2. Using laws about least fixed points and the definitions of concrete fixed points in Section 6.4.3, function *rewrite* is transformed into an efficient rewriting function in a sequence of four steps in Section 6.4.4.

Many of the functions defined in sequel are parametrized on the rule list (representing the rewriting system). As the rule list argument is fixed during the rewriting calculations, we write this argument as a subscript to improve readability. For example, we write $\mathit{rewrite}_{rs}\ t$ for the application of function *rewrite* to the rule list *rs* and the term t.

6.4.1 One step rewriting

Given a rule list *rs* and a term t to match we can select the first matching rule with $\mathit{firstmatch}_{rs}\ t$:

$$
\begin{array}{l}
\mathit{firstmatch} :: \mathit{Term}\ t \Rightarrow \mathit{Rules}\ t \to t \to \mathit{Maybe}\ (\mathit{Sub}\ t, t) \\
\mathit{firstmatch}_{rs}\ t \ = \ \mathit{firstJust}\ (\mathit{map}\ (\mathit{try}\ t)\ \mathit{rs}) \\
\quad \textbf{where}\ \mathit{try}\ t\ (\mathit{lhs}, \mathit{rhs}) \ = \ \mathit{mapM}\ (\lambda s \to (s, \mathit{rhs}))\ (\mathit{match}\ \mathit{lhs}\ t)
\end{array}
$$

$$
\begin{array}{l}
\mathit{firstJust} :: [\mathit{Maybe}\ a] \to \mathit{Maybe}\ a \\
\mathit{firstJust} \ = \ \mathit{foldr}\ \mathit{mplus}\ \mathit{Nothing}
\end{array}
$$

$$
\begin{array}{l}
\mathit{mplus} :: \mathit{Maybe}\ a \to \mathit{Maybe}\ a \to \mathit{Maybe}\ a \\
\mathit{mplus}\ (\mathit{Just}\ x)\ m \ = \ \mathit{Just}\ x \\
\mathit{mplus}\ \mathit{Nothing}\ m \ = \ m
\end{array}
$$

If a rule matches, then $\mathit{firstmatch}_{rs}\ t$ returns *Just* a pair (s, rhs) of the substitution and the right hand side of the matching rule. A note on notation: we use subscripts for the rule list parameter to various rewriting functions, as in $\mathit{firstmatch}_{rs}$. The subscript is used as a convenient syntax for normal function application.

Using *firstmatch* and *appSubst* we can transform a rule list to a top level reduction function *reduceM* that gives *Just* the rewritten term or *Nothing*. An immediate variant is *reduce* that returns the term unchanged if no rule matches.

$$
\begin{array}{lcl}
\mathit{reduceM} & :: & \mathit{Term}\ t \Rightarrow \mathit{Rules}\ t \to t \to \mathit{Maybe}\ t \\
\mathit{reduceM}_{rs} & = & \mathit{mapM}\ (\mathit{uncurry}\ \mathit{appSubst}) \circ \mathit{firstmatch}_{rs} \\
\mathit{reduce} & :: & \mathit{Term}\ t \Rightarrow \mathit{Rules}\ t \to t \to t \\
\mathit{reduce}_{rs}\ t & = & \mathit{maybe}\ t\ \mathit{id}\ (\mathit{reduceM}_{rs}\ t)
\end{array}
$$

The reduce functions only apply the rewrite rules on the top level of the term, but we want to apply the rules at any level. In a relational treatment of rewriting this corresponds to extending the top level reduction relation to a congruence. To retain the deterministic functional view we have to choose a rewriting strategy. We have chosen the parallel innermost rewriting strategy as this lets us transform the rewriting function into an asymptotically optimal solution. Innermost means that we order the subterms by their depth and apply the reduction function bottom up until the first match, and parallel means that all subterms at the same depth are reduced at the same time. Function *parallelInnermost* takes any top level term transformer to a global one step transformer, using the parallel innermost rewriting strategy. (The corresponding function for the parallel outermost rewriting strategy, *parallelOutermost*, is included here for comparison, but is not used in the sequel.)

$$\begin{array}{l} \mathit{parallelInnermost} :: (\mathit{Children}\ t, \mathit{TopEq}\ t) \Rightarrow (t \to t) \to t \to t \\ \mathit{parallelInnermost}\ f \ = \ \mathit{contIfFixedBy}\ f\ (\mathit{mapC}\ (\mathit{parallelInnermost}\ f)) \end{array}$$

$$\begin{array}{l} \mathit{parallelOutermost} :: (\mathit{Children}\ t, \mathit{TopEq}\ t) \Rightarrow (t \to t) \to t \to t \\ \mathit{parallelOutermost}\ f \ = \ \mathit{contIfFixedBy}\ (\mathit{mapC}\ (\mathit{parallelOutermost}\ f))\ f \end{array}$$

$$\begin{array}{l} \mathit{contIfFixedBy} :: (\mathit{Children}\ t, \mathit{TopEq}\ t) \Rightarrow (t \to t) \to (t \to t) \to t \to t \\ \mathit{contIfFixedBy}\ r\ f \ = \ \textbf{iff}\ \mathit{fixedBy}\ f\ \textbf{then}\ r\ \textbf{else}\ f \end{array}$$

Combining *parallelInnermost* with *reduce* we arrive at one-step rewriting:

$$\begin{array}{l} \mathit{rewrite\_step} :: \mathit{Term}\ t \Rightarrow \mathit{Rules}\ t \to t \to t \\ \mathit{rewrite\_step}_{rs} \ = \ \mathit{parallelInnermost}\ \mathit{reduce}_{rs} \end{array}$$

6.4.2 Rewriting to normal form

The final step needed to obtain rewriting to normal form is, in relational terminology, the transitive closure. As a functional counterpart we use a fixed point operator *fp* that takes a one step reduction function r to a normalizer by applying r until the input term doesn't change:

$$\begin{array}{l} \mathit{fp} :: \mathit{Term}\ t \Rightarrow (t \to t) \to t \to t \\ \mathit{fp}\ f \ = \ \textbf{iff}\ \mathit{fixedBy}\ f\ \textbf{then}\ \mathit{id}\ \textbf{else}\ \mathit{fp}\ f \circ f \end{array}$$

The result $\mathit{res} == \mathit{fp}\ f\ x$, when *fp* terminates, is a fixed point in the sense that $\mathit{res} == f\ \mathit{res}$, that is, $\mathit{fixedBy}\ f\ \mathit{res}$ holds. Now we are ready to define rewriting to normal form:

$$\begin{array}{l} \mathit{rewrite} :: \mathit{Term}\ t \Rightarrow \mathit{Rules}\ t \to t \to t \\ \mathit{rewrite}_{rs} \ = \ \mathit{fp}\ \mathit{rewrite\_step}_{rs} \end{array}$$

Function $\mathit{rewrite}_{rs}$ rewrites a term until no rule applies anymore, that is, it rewrites a term to normal form. A term is in normal form for a rule list rs if it is unchanged by $\mathit{rewrite\_step}_{rs}$:

$$\begin{array}{l} \mathit{normal} :: \mathit{Term}\ t \Rightarrow \mathit{Rules}\ t \to t \to \mathit{Bool} \\ \mathit{normal}_{rs} \ = \ \mathit{fixedBy}\ \mathit{rewrite\_step}_{rs} \end{array}$$

For rule lists rs corresponding to strongly normalizing rewrite systems, $rewrite_{rs}$ will take any term to its normal form, but $rewrite_{rs}$ also works for the subset of normalizing terms of any other rewriting system. If a term has multiple normal forms, then $rewrite_{rs}$ calculates only the one (if any) reachable by the parallel innermost rewriting strategy. If this strategy does not terminate for a certain term, then neither does $rewrite_{rs}$. More formally, we define normalizing terms and the first theorem for *rewrite*:

Definition 6.8 *Normalizing terms:*

$$\begin{array}{lcl}
normalizing & :: & Term\ t \Rightarrow Rules\ t \rightarrow t \rightarrow Bool \\
normalizing_{rs} & = & fix\ moreNormal_{rs} \\
moreNormal & :: & Term\ t \Rightarrow Rules\ t \rightarrow (t \rightarrow Bool) \rightarrow (t \rightarrow Bool) \\
moreNormal_{rs}\ p & = & normal_{rs}\ \vee\ p \circ rewrite_step_{rs}
\end{array}$$

Theorem 6.9 *Rewriting gives a normal form:*

$$\lfloor normalizing_{rs} \rfloor \Rightarrow \lfloor normal_{rs} \circ rewrite_{rs} \rfloor$$

The proof of this theorem by fixed point induction is in the next section.

Function $rewrite_{rs}$ can be seen as an executable specification of rewriting to normal form for a given rule list and a given term. It can be useful for experimenting with different rule lists but for larger terms it is unacceptably inefficient. We define the norm of a term (with respect to a specific rule list) to be the number of (parallel innermost) reduction steps that it takes to reach normal form:

$$\begin{array}{lcl}
norm & :: & Term\ t \Rightarrow Rules\ t \rightarrow t \rightarrow Int \\
norm_{rs}\ t & = & \mathbf{if}\ normal_{rs}\ t\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + norm_{rs}\ (rewrite_step_{rs}\ t)
\end{array}$$

The time it takes to execute $rewrite_{rs}$ is linear in the norm, n, of the input term but quadratic in the (average) size, s, of the term being rewritten. Clearly it should be possible to do better than that - optimally we hope to obtain a running time of $O(n + s)$. Using the laws for fixed points and terms given in the previous sections, we can transform the specification of rewriting, $rewrite_{rs}$, into an optimal function. The result of this transformation, which is linear in the norm of the input term, is presented in Section 6.4.4.

6.4.3 Concrete fixed points

Using *fix* we can give the following equivalent definition of function *fp* presented in Section 6.4.2:

$$
\begin{array}{lcl}
\mathit{fp} & :: & \mathit{Term}\ t \Rightarrow (t \to t) \to t \to t \\
\mathit{fp}\ f & = & \mathit{fix}\ (\mathit{ffp}\ f) \\
\mathit{ffp} & :: & \mathit{Term}\ t \Rightarrow (t \to t) \to (t \to t) \to (t \to t) \\
\mathit{ffp}\ f\ r & = & \mathbf{iff}\ \mathit{fixedBy}\ f\ \mathbf{then}\ \mathit{id}\ \mathbf{else}\ r \circ f
\end{array}
$$

The two parameters of $\mathit{ffp}\ f\ r$ are both functions. The first parameter is used to test if we have reached a concrete fixed point. It does not change during the calculation of $\mathit{fix}\ (\mathit{ffp}\ f)$. The second parameter, on the other hand, is an approximation of $\mathit{fp}\ f$. It starts out as $\perp$ and improves in each iteration of *fix*. Thus $\mathit{ffp}\ f$ is an example of an improvement function.

Calculating fixed points using *ffp* is often very inefficient because of the expensive equality test in *fixedBy*. For some functions f, the efficiency can be improved if the equality test is fused with f, so that $f\ t$ is *Nothing* if the term is left unchanged and *Just* the changed term otherwise. The corresponding change to *ffp* results in *ffpM*:

$$
\begin{array}{l}
\mathit{ffpM} :: (a \to \mathit{Maybe}\ b) \to (b \to a) \to a \to a \\
\mathit{ffpM}\ \mathit{fM}\ r\ =\ \lambda x \to \mathit{maybe}\ x\ r\ (\mathit{fM}\ x)
\end{array}
$$

A typical example of a function f with the desired property is, as we will see later, reduce_{rs}.

A fusion lemma for *ffpM* is an easy consequence of the *maybe* - *mapM* - fusion law from the prelude (Chapter 2):

Lemma 6.10 *ffpM - mapM - fusion:*

$$\mathit{ffpM}\ (\mathit{mapM}\ f \circ g)\ r == \mathit{ffpM}\ g\ (r \circ f)$$

6.4.4 Improving rewriting

In this section we transform the definition of rewrite step by step until we reach a linear algorithm. Each transformation step is reasonably small and the correctness of the whole sequence is proved by a chain of equalities of the intermediate versions. As each function is a version of *rewrite* we will use names such as $\mathit{rewrite}^B$, $\mathit{rewrite}^C$ etc. Proofs of some of the theorems of this section are given in Section 6.5.

Children first

As a first step, we transform the specification into a definition that actually has a slightly worse running time than the original, but which simplifies the coming transitions. We are aiming at using the bottom up nature of the parallel innermost rewriting strategy to obtain an efficient rewriting algorithm that is a bottom up traversal on the outermost level. With rewriting defined using *bup* we can make optimizations based on the invariant that all children already are in normal form when a certain level is to be rewritten. For the first "improved" variant we thus choose:

$$rewrite^{B}_{rs} \;=\; bup\ rewrite_{rs}$$

The correctness of this first step follows from the fact that we can always rewrite the children to normal form first and only then start working on the top level.

Theorem 6.11 *Children first:*

$$rewrite_{rs} \overset{fin}{=\!=\!=} rewrite_{rs} \circ mapC\ rewrite_{rs}$$

Intuitively this theorem follows immediately from the use of an innermost rewriting strategy, but the proof by fixed point induction is rather long and omitted from this presentation.

Corollary 6.12 *Version* B *equals the specification:*

$$rewrite_{rs} \overset{fin}{=\!=\!=} rewrite^{B}_{rs}$$

The corollary follows from theorem 6.11 by *bup*-characterization.

Using the normal children invariant

For the next transformation we need to look more closely at the definition of $rewrite^{B}$. The first thing to note is that the outermost *bup* means that we know that all children are in normal form when the argument to *bup* is applied. Then it is clearly overkill to use the full fledged *rewrite* function at that stage, when a simpler variant would suffice, but it is important that the simpler variant is guaranteed to produce only normal forms so that the "normal children" invariant is preserved. To get an idea of where to go next, we expand the definitions of *rewrite* and *fp* to arrive at this equivalent definition of $rewrite^{B}$:

$$rewrite^{B}_{rs} \;=\; bup\ (fix\ (ffp\ rewrite\_step_{rs}))$$

If we unfold *fix* one level, then we see that the argument to *bup* is of the form $r = \mathit{ffp}\ \mathit{rewrite\_step}_{rs}\ r'$. The "normal children" invariant means that the function r will receive a term whose children are in normal form. If we expand the definition of *ffp*, then we get

$$r \;=\; \textbf{iff}\ \mathit{fixedBy}\ \mathit{rewrite\_step}_{rs}\ \textbf{then}\ \mathit{id}\ \textbf{else}\ r' \circ \mathit{rewrite\_step}_{rs}$$

As the children of the input term are in normal form, the bottom up rewriting strategy implemented by $\mathit{rewrite\_step}_{rs}$ will not find a reducible term until possibly at the top level. More formally, this is captured by the following lemma.

Lemma 6.13 *Rewrite with normal children is reduce:*

$$\mathit{rewrite\_step}_{rs} \overset{\mathit{deeper\ normal}_{rs}}{===} \mathit{reduce}_{rs}$$

Proof: Expand the definition of *rewrite_step* one level and use Lemma 6.18. □

Thus we can replace *rewrite_step* by *reduce* in r to obtain $r == \mathit{ffp}\ \mathit{reduce}_{rs}\ r'$. Unfortunately we cannot immediately make the same transformation also for r' as the term argument of r' is $\mathit{reduce}_{rs}\ t$ whose children need not be in normal form. But if we replace r' with $\mathit{bup}\ r$, then the function r will only be applied to normal terms. (That this is really the case is not easy to see, but it is confirmed by a proof using fixed point induction.)

To summarize, we can replace $\mathit{fix}\ (\mathit{ffp}\ \mathit{rewrite\_step}_{rs})$ by $\mathit{fix}\ (\mathit{ffp}\ \mathit{reduce}_{rs} \circ \mathit{bup})$ to arrive at the definition:

$$\mathit{rewrite}^{C}_{rs} \;=\; \mathit{bup}\ (\mathit{fix}\ (\mathit{ffp}\ \mathit{reduce}_{rs} \circ \mathit{bup}))$$

This mind boggling creature can be simplified somewhat by the rolling rule: $f\ (\mathit{fix}\ (g \circ f)) \;=\; \mathit{fix}\ (f \circ g)$

$$\mathit{rewrite}^{C}_{rs} \;=\; \mathit{fix}\ (\mathit{bup} \circ \mathit{ffp}\ \mathit{reduce}_{rs})$$

Theorem 6.14 *Version* B *equals version* C*:*

$$\mathit{rewrite}^{B}_{rs} \overset{\mathit{normalizing}_{rs}}{===} \mathit{rewrite}^{C}_{rs}$$

The asymptotic complexity is not changed by this transformation.

Removing equality checks

Function *reduce* can be expressed in terms of *ffpM* and *reduceM*.

$$reduce_{rs} \;=\; ffpM\ reduceM_{rs}\ id$$

If we expand this definition in version C, then we get:

$$rewrite^{C}_{rs} \;=\; fix\ (bup \circ ffp\ (ffpM\ reduceM_{rs}\ id))$$

We can simplify the expression *ffp* (*ffpM* $reduceM_{rs}$ *id*) to *ffpM* $reduceM_{rs}$ and thus remove all equality checks, if we make an assumption about the rule list:

$$\lfloor reduceM_{rs}\ t == Just\ x \rfloor \Rightarrow \lfloor \neg\,(t == x) \rfloor$$

By examining the implementation of $reduceM_{rs}$, we can express the requirement as follows: in the rule list *rs*, a rule should only match a term, if applying that rule changes the term. This is a reasonable requirement. Otherwise, if a rule can match but leave the term unchanged, then $rewrite^{D}_{rs}$ will loop even though $rewrite^{C}_{rs}$ would have terminated (with the unchanged term).

The new version of the rewriting function is the following:

$$rewrite^{D}_{rs} \;=\; fix\ (bup \circ ffpM\ reduceM_{rs})$$

Theorem 6.15 *Version C equals version D:*

$$rewrite^{C}_{rs} \overset{normalizing_{rs}}{===} rewrite^{D}_{rs}$$

The simple proof is presented in Section 6.5.

This transformation reduces the asymptotic complexity of rewriting: version A-C are quadratic in the size (as calculated by function *size* in Section 6.2.3) but version D is linear in the size. They are all linear in the norm (that is, the number of rewrite steps as calculated by function *norm* is Section 6.2.3). Moreover, removing the equality tests is essential to obtain a version with a better complexity than linear in the size.

Avoiding unnecessary traversals of normal children

An analysis of $\mathit{rewrite}^D$ shows that quite some time is spent on trying to rewrite terms that are already normal. After each successful rewrite step with a rule $(\mathit{lhs}, \mathit{rhs})$, the resulting term is traversed bottom up in search of redeces. But as the incoming term has normal children, the resulting term is *rhs* with normal terms substituted for its variables. Clearly any redeces in this result must be in the topmost part coming from *rhs*. The following transformation changes $\mathit{rewrite}^D$ to limit the search for redeces to this topmost part.

We start by using the rolling rule on version D.

$$
\begin{array}{ll}
 & \mathit{rewrite}^D_{rs} \\
\equiv & \quad \{\, \text{Definition} \,\} \\
 & \mathit{fix}\ (\mathit{bup} \circ \mathit{ffpM}\ \mathit{reduceM}_{rs}) \\
\equiv & \quad \{\, \text{Rolling rule (Lemma 2.12)} \,\} \\
 & \mathit{bup}\ (\mathit{fix}\ (\mathit{ffpM}\ \mathit{reduceM}_{rs} \circ \mathit{bup})) \\
\equiv & \quad \{\, \text{Introduce } \mathit{reduce}^E_{rs} \ = \ \mathit{ffpM}\ \mathit{reduceM}_{rs} \circ \mathit{bup} \,\} \\
 & \mathit{bup}\ (\mathit{fix}\ \mathit{reduce}^E_{rs})
\end{array}
$$

Now we transform reduce^E.

$$
\begin{array}{ll}
 & \mathit{reduce}^E_{rs}\ r \\
\equiv & \quad \{\, \text{Definition of } \mathit{reduce}^E_{rs} \,\} \\
 & \mathit{ffpM}\ \mathit{reduceM}_{rs}\ (\mathit{bup}\ r) \\
\equiv & \quad \{\, \text{Def.: } \mathit{reduceM}_{rs} \ = \ \mathit{mapM}\ (\mathit{uncurry}\ \mathit{appSubst}) \circ \mathit{firstmatch}_{rs} \,\} \\
 & \mathit{ffpM}\ (\mathit{mapM}\ (\mathit{uncurry}\ \mathit{appSubst}) \circ \mathit{firstmatch}_{rs})\ (\mathit{bup}\ r) \\
\equiv & \quad \{\, \mathit{ffpM}\text{-}\mathit{mapM}\text{-fusion (Lemma 6.10)} \,\} \\
 & \mathit{ffpM}\ \mathit{firstmatch}_{rs}\ (\mathit{bup}\ r \circ \mathit{uncurry}\ \mathit{appSubst}) \\
\equiv & \quad \{\, \mathit{bup}\text{-}\mathit{appSubst}\text{-fusion (Corollary 6.7)} \,\} \\
 & \mathit{ffpM}\ \mathit{firstmatch}_{rs}\ (\mathit{fromVarsUpAfterSubst}\ r)
\end{array}
$$

The real improvement comes in the last step. In the next to last expression, remember that $\mathit{firstmatch}_{rs}\ t$ gives *Just* a pair (s, rhs) of a substitution and the right hand side of the matching rewrite rule, or *Nothing* if no rule applies. Function *appSubst* applies s to rhs, and $\mathit{bup}\ r$ applies r at all levels in the resulting term. We know that all children of the incoming term are normal, and this means that all variables in the substitution s will be bound to normal terms. But when *appSubst* is applied the information about which terms are normal is lost, and the bottom up traversal is used to restore the invariant.

The improved version is obtained by fusing $\mathit{bup}\ r$ with *appSubst* to a function $\mathit{fromVarsUpAfterSubst}\ r$. The improved version only applies r from the variables in the right hand side of the matching rule and upwards upwards, and leaves the normal children alone. The improved version is thus:

$$\mathit{rewrite}^E_{rs} \;=\; \mathit{bup}\ (\mathit{fix}\ (\mathit{ffpM}\ \mathit{firstmatch}_{rs} \circ \mathit{fromVarsUpAfterSubst}))$$

Theorem 6.16 *Version* D *equals version* E*:*

$$\mathit{rewrite}^D_{rs} \overset{\mathit{normalizing}_{rs}}{===} \mathit{rewrite}^E_{rs}$$

This version of *rewrite* is linear in the number of steps needed to rewrite a term, and independent of the size of the intermediate terms. This big improvement is obtained by avoiding repeated traversals of already normal children. The improved version instead only traverses the right hand sides from the matching rules.

6.4.5 Efficiency comparison

A very simple measure of the running time for the different rewriting functions is the number of Hugs-reduction steps (not to be confused with rewriting steps in the rewriting system) required to run the functions on some examples. The following table shows some measurements of the number of Hugs reductions required by the different versions to rewrite the expression 2^n for $n = 6, 7, 8$. The number 2 abbreviates $S\ (S\ Z) :: \mathit{Expr}$ and the exponentiation notation (2^n), is a shorthand for repeated multiplications (uses of (:∗:)). The expression is normalized using the rewrite rules *exprrules* defined in Section 6.1.

		Hugs-reductions for version				
expression	rewrite steps	A	B	C	D	E
2^6	107	$7.4M$	$7.4M$	$2.7M$	$478k$	$72k$
2^7	179	$47M$	$47M$	$20M$	$1.6M$	$122k$
2^8	323	$344M$	$344M$	$156M$	$5.7M$	$218k$
e	n	$O(e^2n)$	$O(e^2n)$	$O(e^2n)$	$O(en)$	$O(n)$

The last line in the table gives the asymptotic complexity for the different versions in terms of the size of the answer e and the number of parallel innermost rewrite steps n. The number of rewrite steps n increases more slowly than the size of the answer e as the parallel rewriting strategy performs more and more inner reductions in parallel as the terms grow. Hence, n is not quite proportional to e, and we can analyze the complexity in terms of both variables. As we can see from the table all versions are linear in n but the dependence on e differs. As n is the number of rewrite steps, we can think of the dependence on e as the complexity per rewrite step.

For versions A-C the complexity can be explained by the test for equality at every node in the term. As the equality check and the number of nodes are both linear in e, we get a quadratic dependency in total. An equality test that reports *False* is often quick, but determining that two terms are completely equal is of course linear. As the equality checks are performed to see if a term is in normal form, we can confirm the suspicion that these versions do a lot of work on already normal (sub)terms.

Version D reduces the e complexity to linear, by removing the equality checks, but in every rewrite step it still traverses the normal subterms and applies *match* in search for reducible terms. Version E completely removes the unnecessary traversals of normal subterms, and thus reduces the cost of each rewrite step to a constant (determined by the rule list).

6.5 Proofs

In this section we prove the term combinator laws from Section 6.2.4 and a selection of the rewriting function laws from Section 6.4. We start with a few lemmata used in the proofs.

As variables have no children, they cannot be changed by *mapC*:

Lemma 6.17 *mapC does not change variables:*

$$\lfloor isJust \circ varCheck \rfloor \Rightarrow \lfloor mapC\ f === id \rfloor$$

Lemma 6.18 *Fixed by mapC:*

$$fixedBy\ (mapC\ f) === deeper\ (fixedBy\ f)$$

The following lemma (used in the proof of *bup*-characterization) is an easy consequence of the laws required for *Term* instances and the definition of (==).

Lemma 6.19 *Deep equality and mapC:*

$$mapC\ f === mapC\ g \quad = \quad deeper\ (f === g)$$

The improvement function *deeper* used in the definition of $fin \ = \ fix\ deeper$ is monotone in the following sense:

Lemma 6.20 *deeper is monotone:*

For all p and q:

$$(\lfloor p \rfloor \Rightarrow \lfloor q \rfloor) \Rightarrow (\lfloor deeper\ p \rfloor \Rightarrow \lfloor deeper\ q \rfloor)$$

6.5.1 Proofs of term combinator laws

In this subsection we restate and prove the laws from Section 6.2.4.

Theorem 6.2 (with proof) *bup*-characterization:

$$(f \overset{fin}{===} g \circ mapC\ f) \ \equiv \ (f \overset{fin}{===} bup\ g)$$

Proof: The $\Leftarrow$ implication follows immediately from the definition of *bup*. For the other implication we first assume the left hand side is true and expand the definition of the right hand side to expose the fixed points:

$$f \overset{fix\ deeper}{===} fix\ ((g\circ) \circ mapC)$$

We use fixed point induction with $n \ = \ 2$, improvement functions $i_1 \ = \ (g\circ) \circ mapC$ and $i_2 \ = \ deeper$ and relation P:

$$P\ (h, p) \ = \ f \overset{p}{===} h \ \wedge \ InLim_{i_2}\ p$$

The side condition $InLim_{i_2}\ p \ = \ \lfloor p \rfloor \Rightarrow \lfloor fin \rfloor$ follows from Lemma 6.20 (monotonicity of *deeper*) and Lemma 2.17 (*InLim*), thus we only need to prove the equality here.

Base case: $P\ (\bot, \bot)$ is trivially true as $\lfloor \bot \rfloor \ = \ false$ and $(false \Rightarrow q) \ = \ true$.

Inductive case: We prove $P\ (g \circ mapC\ h, deeper\ p) \Leftarrow P\ (h, p)$ by calculation:

$$
\begin{array}{ll}
 & P\ (g \circ mapC\ h, deeper\ p) \\
\equiv & \quad \{\text{Definitions}\} \\
 & f \overset{deeper\ p}{===} g \circ mapC\ h \\
\Leftarrow & \quad \{\text{Transitivity: } a \overset{q}{===} c \Leftarrow (a \overset{q}{===} b) \wedge (b \overset{q}{===} c)\ \} \\
 & (f \overset{deeper\ p}{===} g \circ mapC\ f) \wedge (g \circ mapC\ f \overset{deeper\ p}{===} g \circ mapC\ h) \\
\Leftarrow & \quad \{\text{Use the assumption } f \overset{fin}{===} g \circ mapC\ f \text{ on the left}\} \\
 & g \circ mapC\ f \overset{deeper\ p}{===} g \circ mapC\ h \\
\Leftarrow & \quad \{\text{Cancel } g\text{, definition of } (\overset{q}{===})\} \\
 & \lfloor deeper\ p \rfloor \Rightarrow \lfloor mapC\ f === mapC\ h \rfloor \\
\equiv & \quad \{\text{Lemma 6.19}\} \\
 & \lfloor deeper\ p \rfloor \Rightarrow \lfloor deeper\ (f === h) \rfloor \\
\Leftarrow & \quad \{\text{Lemma 6.20: } deeper \text{ is monotone}\} \\
 & \lfloor p \rfloor \Rightarrow \lfloor f === h \rfloor \\
\equiv & \quad \{\text{Definition of } (\overset{q}{===}) \text{ and } P\ (h, p)\} \\
 & P\ (h, p)
\end{array}
$$

□

Theorem 6.4 (with proof) bup-equality:

$$
\begin{array}{l}
(g \circ mapC\ f \overset{fin}{===} h \circ mapC\ f) \equiv (bup\ g \overset{fin}{===} bup\ h) \\
\quad \textbf{where}\ f = bup\ g
\end{array}
$$

Proof: We calculate as follows:

$$
\begin{array}{ll}
 & g \circ mapC\ f \overset{fin}{===} h \circ mapC\ f \\
\equiv & \quad \{\text{By definition: } f = bup\ g\} \\
 & g \circ mapC\ (bup\ g) \overset{fin}{===} h \circ mapC\ (bup\ g) \\
\equiv & \quad \{\text{By definition: } g \circ mapC\ (bup\ g) === bup\ g\} \\
 & bup\ g \overset{fin}{===} h \circ mapC\ (bup\ g) \\
\equiv & \quad \{bup\text{-characterization}\} \\
 & bup\ g \overset{fin}{===} bup\ h
\end{array}
$$

□

Theorem 6.5 (with proof) *bup* is a *foldTerm*:

$$foldTerm\ f\ (const\ Nothing) \overset{fin}{===} bup\ f$$

Proof: By *bup*-characterization the theorem follows from:

$$foldTerm\ f\ (const\ Nothing) \overset{fin}{===} f \circ mapC\ (foldTerm\ f\ (const\ Nothing))$$

We let $s\ =\ const\ Nothing$ and calculate:

$$\begin{aligned}
& foldTerm\ f\ s \\
= \quad & \{\ \text{Definition of } foldTerm\ \} \\
& \lambda t \to maybe\ (f\ (mapC\ (foldTerm\ f\ s)\ t)) \\
& \qquad\qquad (maybe\ (f\ t)\ id \circ const\ Nothing) \\
& \qquad\qquad (varCheck\ t) \\
= \quad & \{\ \text{Simplify } maybe\text{: } maybe\ n\ j \circ const\ Nothing === const\ n\ \} \\
& \lambda t \to maybe\ (f\ (mapC\ (foldTerm\ f\ s)\ t)) \\
& \qquad\qquad (const\ (f\ t)) \\
& \qquad\qquad (varCheck\ t) \\
= \quad & \{\ \text{Lemma 6.17: } mapC \text{ does not change variables}\ \} \\
& \lambda t \to maybe\ (f\ (mapC\ (foldTerm\ f\ s)\ t)) \\
& \qquad\qquad (const\ (f\ (mapC\ (foldTerm\ f\ s)\ t))) \\
& \qquad\qquad (varCheck\ t) \\
= \quad & \{\ \text{Simplification: } maybe\ n\ (const\ n)\ m == n \text{ if } m \text{ is not } \perp\ \} \\
& \lambda t \to f\ (mapC\ (foldTerm\ f\ s)\ t) \\
= \quad & \{\ \text{Definition of } (\circ)\ \} \\
& f \circ mapC\ (foldTerm\ f\ s)
\end{aligned}$$

The value m in the second last step is not $\perp$ because the input term is finite and *varCheck* terminates for finite terms. $\Box$

Theorem 6.6 (with proof) *bup*-*mapTerm*-fusion:

$$\lfloor mapM\ (bup\ f) \circ s === s \rfloor\ \equiv\ foldTerm\ f\ s \overset{fin}{===} bup\ f \circ mapTerm\ s$$

Proof: We give a calculational proof by induction over the depth of the incoming term. Thus the induction hypothesis is that the equality holds for terms of lower depth.

$$
\begin{array}{cl}
& \mathit{bup}\ f \circ \mathit{mapTerm}\ s \\
= & \quad \{\text{definition of } \mathit{mapTerm}, \mathit{foldTerm}\} \\
& \mathit{bup}\ f \circ \lambda t \to \mathit{maybe}\ (\mathit{mapC}\ (\mathit{mapTerm}\ s)\ t) \\
& \qquad\qquad\qquad\quad (\mathit{maybe}\ t\ \mathit{id} \circ s) \\
& \qquad\qquad\qquad\quad (\mathit{varCheck}\ t) \\
= & \quad \{\mathit{maybe}\text{-fusion: } g \circ \mathit{maybe}\ n\ j\ =\ \mathit{maybe}\ (g\ n)\ (g \circ j)\} \\
& \lambda t \to \mathit{maybe}\ (\mathit{bup}\ f\ (\mathit{mapC}\ (\mathit{mapTerm}\ s)\ t)) \\
& \qquad\qquad (\mathit{maybe}\ (\mathit{bup}\ f\ t)\ (\mathit{bup}\ f) \circ s) \\
& \qquad\qquad (\mathit{varCheck}\ t) \\
= & \quad \{\text{Subcalculations below of the first two arguments of } \mathit{maybe}\} \\
& \lambda t \to \mathit{maybe}\ (f\ (\mathit{mapC}\ (\mathit{foldTerm}\ f\ s)\ t)) \\
& \qquad\qquad (\mathit{maybe}\ (f\ t)\ \mathit{id} \circ s) \\
& \qquad\qquad (\mathit{varCheck}\ t) \\
= & \quad \{\text{definition of } \mathit{foldTerm}\} \\
& \mathit{foldTerm}\ f\ s
\end{array}
$$

The second but last step, where we simplify the first two arguments to *maybe*, is motivated by the following calculations. For the first argument we have

$$
\begin{array}{cl}
& \mathit{bup}\ f \circ \mathit{mapC}\ (\mathit{mapTerm}\ s) \\
= & \quad \{\text{Definition } \mathit{bup}\} \\
& f \circ \mathit{mapC}\ (\mathit{bup}\ f) \circ \mathit{mapC}\ (\mathit{mapTerm}\ s) \\
= & \quad \{\mathit{mapC} \text{ preserves composition}\} \\
& f \circ \mathit{mapC}\ (\mathit{bup}\ f \circ \mathit{mapTerm}\ s) \\
= & \quad \{\text{Induction hypothesis: } \lfloor \mathit{bup}\ f \circ \mathit{mapTerm}\ s === \mathit{foldTerm}\ f\ s \rfloor\} \\
& f \circ \mathit{mapC}\ (\mathit{foldTerm}\ f\ s)
\end{array}
$$

In the simplification of the second argument of *maybe*, we know that the incoming term is a variable. Thus we can calculate as follows:

$$
\begin{array}{ll}
& maybe\ (bup\ f\ t)\ (bup\ f) \circ s \\
= & \{\ \text{Unfold the first } bup \text{ one level}\ \} \\
& maybe\ (f\ (mapC\ (bup\ f)\ t))\ (bup\ f) \circ s \\
= & \{\ \text{Lemma 6.17: } mapC \text{ does not change variables}\ \} \\
& maybe\ (f\ t)\ (bup\ f) \circ s \\
= & \{\ maybe\text{-law: } maybe\ n\ (j \circ g) === (maybe\ n\ j) \circ mapM\ g\ \} \\
& maybe\ (f\ t)\ id \circ mapM\ (bup\ f) \circ s \\
= & \{\ \text{Assumption } mapM\ (bup\ f) \circ s === s\ \} \\
& maybe\ (f\ t)\ id \circ s
\end{array}
$$

□

6.5.2 Proofs of rewriting transformations

In this subsection we restate and prove a selection of the rewriting function laws from Section 6.4.

Lemma 6.21 *moreNormal is monotone:*

For all p and q:

$$(\lfloor p \rfloor \Rightarrow \lfloor q \rfloor) \Rightarrow (\lfloor moreNormal_p \rfloor \Rightarrow \lfloor moreNormal_q \rfloor)$$

Theorem 6.9 (with proof) Rewriting gives a normal form:

$$\lfloor normalizing_{rs} \rfloor \Rightarrow \lfloor normal_{rs} \circ rewrite_{rs} \rfloor$$

Proof: Expand the definitions of *normalizing* and *rewrite* to expose the fixed points:

$$\lfloor \mathit{fix}\ \mathit{moreNormal}_{rs} \rfloor \Rightarrow \lfloor \mathit{normal}_{rs} \circ \mathit{fix}\ (\mathit{ffp}\ \mathit{rewrite\_step}_{rs}) \rfloor$$

Use fixed point induction with $n = 2$, predicate

$$P\ (f, p) \ = \ \lfloor p \rfloor \Rightarrow \lfloor \mathit{normal}_{rs} \circ f \rfloor$$

and the improvement functions $i_1 = \mathit{ffp}\ \mathit{rewrite\_step}_{rs}$ and $i_2 = \mathit{moreNormal}_{rs}$. The base case $P\ (\bot, \bot)$ is trivial as the left hand side of the implication is false. For the inductive step start by transforming $\mathit{normal}_{rs} \circ \mathit{ffp}\ \mathit{rewrite\_step}_{rs}\ f$.

$$\begin{array}{ll}
 & \lfloor \mathit{normal}_{rs} \circ \mathit{ffp}\ \mathit{rewrite_step}_{rs}\ f \rfloor \\
\equiv & \quad \{\, \text{Definition of } \mathit{ffp} \,\} \\
 & \lfloor \mathit{normal}_{rs} \circ \mathbf{iff}\ \mathit{normal}_{rs}\ \mathbf{then}\ \mathit{id}\ \mathbf{else}\ f \circ \mathit{rewrite_step}_{rs} \rfloor \\
\equiv & \quad \{\, \text{Lemma 2.8: } \mathbf{iff}\ \ \mathbf{then}\ \ \mathbf{else}\text{-fusion} \,\} \\
 & \mathbf{iff}\ \mathit{normal}_{rs}\ \mathbf{then}\ \lfloor \mathit{normal}_{rs} \rfloor\ \mathbf{else}\ \lfloor \mathit{normal}_{rs} \circ f \circ \mathit{rewrite_step}_{rs} \rfloor \\
\equiv & \quad \{\, \text{Lemma 2.9} \,\} \\
 & \mathbf{iff}\ \mathit{normal}_{rs}\ \mathbf{then}\ \mathit{true}\ \mathbf{else}\ \lfloor \mathit{normal}_{rs} \circ f \circ \mathit{rewrite_step}_{rs} \rfloor \\
\equiv & \quad \{\, \text{Lemma 2.10: } \vee \text{ expressed with } \mathbf{iff}\ \ \mathbf{then}\ \ \mathbf{else}\ . \,\} \\
 & \lfloor \mathit{normal}_{rs}\ \vee\ (\mathit{normal}_{rs} \circ f \circ \mathit{rewrite_step}_{rs}) \rfloor \\
\equiv & \quad \{\, \text{Definition of } \mathit{moreNormal} \,\} \\
 & \lfloor \mathit{moreNormal}_{rs}\ (\mathit{normal}_{rs} \circ f) \rfloor
\end{array}$$

Thus prepared the calculation is simple:

$$\begin{array}{ll}
 & P\ (f, p) \\
\equiv & \quad \{\, \text{Definition of } P \,\} \\
 & \lfloor p \rfloor \Rightarrow \lfloor \mathit{normal}_{rs} \circ f \rfloor \\
\Rightarrow & \quad \{\, \text{Monotonicity of } \mathit{moreNormal} \,\} \\
 & \lfloor \mathit{moreNormal}_{rs}\ p \rfloor \Rightarrow \lfloor \mathit{moreNormal}_{rs}\ (\mathit{normal}_{rs} \circ f) \rfloor \\
\equiv & \quad \{\, \text{Preceding calculation} \,\} \\
 & \lfloor \mathit{moreNormal}_{rs}\ p \rfloor \Rightarrow \lfloor \mathit{normal}_{rs} \circ \mathit{ffp}\ \mathit{rewrite_step}_{rs}\ f \rfloor \\
\equiv & \quad \{\, \text{Definition of } P \,\} \\
 & P\ (\mathit{ffp}\ \mathit{rewrite_step}_{rs}, \mathit{moreNormal}_{rs}\ f)
\end{array}$$

□

Theorem 6.15 (with proof) Version $\textsc{c}$ equals version $\textsc{d}$:

$$rewrite^{C}_{rs} \stackrel{normalizing_{rs}}{===} rewrite^{D}_{rs}$$

Proof: The definitions of $rewrite^{C}_{rs}$ and $rewrite^{D}_{rs}$ are very similar:

$$\begin{array}{lcl} rewrite^{C}_{rs} & = & \mathit{fix}\ (bup \circ \mathit{ffp}\ reduce_{rs}) \\ rewrite^{D}_{rs} & = & \mathit{fix}\ (bup \circ \mathit{ffpM}\ \ reduceM_{rs}) \end{array}$$

Thus it is enough to show that $\mathit{ffp}\ \ reduce_{rs} === \mathit{ffpM}\ \ reduceM_{rs}$. Remember that $reduce_{rs}\ t\ = \ maybe\ t\ id\ (reduceM_{rs}\ t)$ and calculate as follows:

$$\begin{array}{ll}
 & \mathit{ffp}\ reduce_{rs}\ r\ t == \mathit{ffpM}\ \ reduceM_{rs}\ r\ t \\
\equiv & \{\, \text{Definition of } \mathit{ffp} \text{ and } \mathit{ffpM}\ \} \\
 & \textbf{if}\ t == reduce_{rs}\ t\ \textbf{then}\ t\ \textbf{else}\ r\ (reduce_{rs}\ t) \\
 & == maybe\ t\ r\ (reduceM_{rs}\ t) \\
\equiv & \{\, \text{Case analysis on } reduceM_{rs}\ t\ \} \\
 & \textbf{case}\ reduceM_{rs}\ t\ \textbf{of} \\
 & \quad Nothing \ \ \rightarrow \ \ (\textbf{if}\ t == t\ \textbf{then}\ t\ \textbf{else}\ r\ t) == t \\
 & \quad Just\ x \ \ \ \ \rightarrow \ \ (\textbf{if}\ t == x\ \textbf{then}\ t\ \textbf{else}\ r\ x) == r\ x \\
\equiv & \{\, \text{Simplify} \,\} \\
 & \textbf{case}\ reduceM_{rs}\ t\ \textbf{of} \\
 & \quad Nothing \ \ \rightarrow \ \ t == t \\
 & \quad Just\ x \ \ \ \ \rightarrow \ \ \textbf{if}\ t == x\ \textbf{then}\ t == r\ x\ \textbf{else}\ r\ x == r\ x \\
\equiv & \{\, \text{Reflexivity and assume } \neg\ (t == x) \text{ is true} \,\} \\
 & True
\end{array}$$

Thus $\mathit{ffp}\ reduce_{rs} === \mathit{ffpM}\ \ reduceM_{rs}$ follows from the assumption

$$\lfloor reduceM_{rs}\ t == Just\ x \rfloor \Rightarrow \lfloor \neg\ (t == x) \rfloor\ .$$

□

6.6 Conclusions

We have presented a framework for polytypic programming on terms, with which polytypic programs for matching, unification, rewriting, etc. can be constructed. The framework is an interface consisting of four functions. Using these four basic functions we have defined a set of combinators on terms, and we have proved several laws for these combinators. The framework has been used to calculate an efficient rewriting program from an inefficient, clearly correct specification.

Because the only polytypic components of the functions for rewriting, matching and unification are the functions in the term interface, our functions are independent of the particular implementation of polytypism. This is an important advantage. Other, less domain specific, frameworks for polytypic programming are the monadic traversal library of Moggi, Bellè and Jay [81] and the basic combinator library PolyLib (Chapter 5). Very likely there are other domain specific polytypic libraries, but they can only be determined by developing many example polytypic programs.

Chapter 7

Polytypic Data Conversion Programs[1]

Abstract

Several generic programs for converting values from regular datatypes to some other format, together with their corresponding inverses, are constructed. The formats considered are shape plus contents, compact bit streams and pretty printed strings. The different data conversion programs are constructed using John Hughes' arrow combinators along with a proof that printing (from a regular datatype to another format) followed by parsing (from that format back to the regular datatype) is the identity. The printers and parsers are described in PolyP, a polytypic extension of the functional language Haskell.

7.1 Introduction

Many programs convert data from one format to another, for example, parsers, pretty printers, data compressors, encryptors and functions that communicate with a database. Some of these programs, such as parsers and pretty printers, critically depend on the structure of the input data. Other programs, such as most data compressors and encryptors, more or less ignore the structure of the data. Using the structure of the input data in a program for a data conversion problem almost always gives a more efficient program with better results. For example,

[1] An article version of this chapter has been submitted to Science of Computer Programming in 2000 [53]. A shorter version, "Polytypic compact printing and parsing", appeared in the proceedings of the European Symposium on Programming in 1999 [48].

a data compressor that uses the structure of the input data runs faster and compresses better than a conventional data compressor. This chapter constructs several polytypic data conversion programs that make use of the structure of the input data. We construct programs for determining the shape of data, packing and pretty printing data.

7.1.1 Data conversion programs

Shape.

A value of a container type $d\ a$ can be uniquely represented by its shape (of type $d\ ()$) and a list of its contents (of type $[a]$). As an example, consider the datatype of binary trees with leaves containing values of type a.

$$\textbf{data}\ \mathit{Tree}\ a\ =\ \mathit{Leaf}\ a \mid \mathit{Bin}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

The following example binary tree

$$\begin{aligned}&\mathit{tree} :: \mathit{Tree}\ \mathit{Int}\\ &\mathit{tree}\ =\ \mathit{Bin}\ (\mathit{Bin}\ (\mathit{Leaf}\ 1)\ (\mathit{Bin}\ (\mathit{Leaf}\ 7)\ (\mathit{Leaf}\ 3)))\ (\mathit{Leaf}\ 8)\end{aligned}$$

can be represented by a pair of its shape

$$\begin{aligned}&\mathit{treeShape} :: \mathit{Tree}\ ()\\ &\mathit{treeShape}\ =\ \mathit{Bin}\ (\mathit{Bin}\ (\mathit{Leaf}\ ())\ (\mathit{Bin}\ (\mathit{Leaf}\ ())\ (\mathit{Leaf}\ ())))\ (\mathit{Leaf}\ ())\end{aligned}$$

and its contents $[1, 7, 3, 8]$.

Our first data conversion program is a program for separating a value into its shape and its contents, together with its inverse: a program that combines a shape and some contents into a datatype value. The construction proves that the two functions are each others' inverses. Note that shapes are at the heart of Jay's [56] theory of polytypism, but here we only use *separate* and *combine* as examples of simple data conversion programs.

We start with this almost trivial data conversion problem because these conversion functions serve as nice examples of simple polytypic programs, but also because much of the essential structure of the packing and pretty printing programs is present already at this stage.

Packing.

Many files that are distributed around the world, either over the Internet or on CD-rom, possess structure — examples are databases, HTML files, and JavaScript programs — and it pays to compress these structured files to obtain faster transmission or fewer CDs. Structure-specific compression methods give much better compression results than conventional compression methods such as the Unix compress utility [6, 15]. Structured compression is also used in heap compression and binary I/O [104].

The idea of designing structure-specific compression programs has been around since the beginning of the 1980s, but, as far as we are aware, there is no generic description of the program, only example instantiations appear in the literature. This chapter describes the compression program generically by combining a polytypic parser with a polytypic packing program. The uncompression program is similarly composed of a polytypic unpacker and a polytypic pretty printer.

Our packing algorithm compresses data by compactly representing the structure of the data using only static information — the type of the data. Traditional (bit stream) compressors that use dynamic (statistical) properties of the data are largely orthogonal to our approach and thus the best compression results are obtained by composing the packer with a bit stream compressor.

Pretty printing.

Modern programming languages allow the user to define new kinds of data. When testing or debugging a program, the user often wants to see values of these new datatypes. Many languages support the automatic derivation of printing functions for user-defined datatypes. For example, by writing **deriving** *Show* after a Haskell datatype definition, the function *show* for this datatype is obtained for free. Thus in Haskell one can use a built-in polytypic function *show*, but *show* can not be expressed in the language, and one can not define alternative polytypic pretty printing functions.

This chapter shows how one can define polytypic versions of the functions *show* and its inverse *read* that work for values of arbitrary regular datatypes. Again, the functions *show* and *read* are each others inverses by construction. Thus we externalize the definitions of these functions (in Haskell they are part of the compiler and cannot be inspected), and we show that our definitions are correct.

7.1.2 Constructing data conversion programs

The fundamental property of the three printing functions *print* just described is that each of them has a right inverse with respect to forward composition: the parsing function *parse*. That is, *print* ; *parse* = *id*, but *parse* ; *print* need not be *id*.[2] In the rest of the chapter we will write just inverse, when we really mean right inverse. This is a very common specification pattern: all data conversion problems are specified as pairs of inverse functions with some additional properties. In this chapter, the driving force behind the definitions of the functions *print* and *parse* is inverse function construction. Thus correctness of *print* and *parse* is guaranteed by construction. Interestingly, when we forced ourselves to only construct pairs of inverse functions, we managed to reduce the size and complexity of the resulting programs considerably compared with our previous attempts.

The conversion programs are expressed using arrows — John Hughes' suggestion for generalizing monads [42]. The arrow combinators can be seen as defining a small (impure) functional language embedded in Haskell. We use constructor classes to allow for varying interpretations of this embedded language. Thus the conversion programs are implicitly parametrized with respect to the choice of implementation and semantics for this embedded language, and the laws needed to prove the correctness of the conversion programs are expressed as restrictions on the possible implementations.

This chapter has the following goals:

- construct a number of polytypic programs for data conversion problems, together with their inverses;
- show how to construct and calculate with polytypic functions.

The implementation of the data conversion programs as PolyP code can be obtained from the polytypic programming WWW page [49].

The rest of this chapter is organized as follows. Section 7.2 constructs polytypic programs for separating a datatype value into its shape and its contents, and for combining shape and contents back to the original value. Section 7.3 introduces an abstract function concept called arrows, which is used a lot in the following sections. Section 7.4 defines two kinds of arrow maps and proves that they are inverses. Section 7.5 sketches the construction and correctness proof of the packing program. Section 7.6 constructs polytypic programs for showing and reading values of datatypes. Section 7.7 defines instances of the various arrow classes. Section 7.8 concludes with an overview of the results, a discussion and some suggestions for future work.

[2] The composition *parse* ; *print* is automatically almost *id*: if $s = print\ x$ then $(parse\ ;\ print)\ s = (print\ ;\ parse\ ;\ print)\ x = (id\ ;\ print)\ x = print\ x = s$. Thus it is *id* on the image of the *print* function (a subset of the set of values that can be parsed) — but the behavior for other values is not specified.

7.2 Shape

The shape of a value is its structure without its contents. This section defines functions for separating a datatype value into its shape and its contents, and for combining shape and contents to a datatype value. Furthermore, it proves that the composition of these functions is the identity.

7.2.1 Function *separate*

A first definition of function *separate*, using the functions *flatten* and *pmap*, was presented already in Section 3.9.

$$\begin{array}{lcl} separate & :: & Regular\ d \Rightarrow\ d\ a \rightarrow (d\ (), [\,a\,]) \\ separate\ x & = & (pmap\ (const\ ())\ x, flatten\ x) \end{array}$$

It is more difficult to define the function *combine*, the inverse of function *separate*. A standard implementation of function *combine* traverses the shape, carrying around the content list, and inserts one element from the list at each of the parameter positions in the shape. Because it is not easy to prove that such a function is the inverse of function *separate*, we redefine function *separate* to make the inverse construction straightforward.

The preceding definition of function *separate* traverses its input datatype value twice: once with *pmap* $(const\ ())$, and once with *flatten*. We can fuse these two traversals into a single traversal that carries around an accumulating state parameter. This traversal is carried out by a function similar to *pmap* which we call an arrow map. The arrow map takes as argument a function, in this case the function *put*, which at each parameter position prepends the element to the accumulating list, and replaces the element by the empty tuple. To avoid 'pollution' of the types with state information, we introduce a new type constructor *SA* for functions that side-effect on a state.

$$\textbf{newtype}\ SA\ s\ a\ b = SA\ ((a, s) \rightarrow (b, s))$$

We use the notation $a \rightsquigarrow_s b$ for $SA\ s\ a\ b$. Using this type and an arrow map called *pmapAr*, we obtain the following definition for function *separate*.

$$\begin{array}{lcl} separate & :: & d\ a \rightsquigarrow_{[a]} d\ () \\ separate & = & pmapAr\ put \end{array}$$

$$\begin{array}{lcl} put & :: & a \rightsquigarrow_{[a]} () \\ put & = & SA\ (\lambda(a, xs) \rightarrow ((), a : xs)) \end{array}$$

$$pmapAr\ ::\ (a \rightsquigarrow_s b) \rightarrow (d\ \ a \rightsquigarrow_s d\ \ b)$$

where *pmapAr* is defined in Section 7.4. The r in *pmapAr* denotes the direction of the traversal: *pmapAr* is a right to left traversal. This means that given a tree node with two subtrees, function *pmapAr* first traverses the right subtree, and then the left subtree. Direction doesn't matter for normal maps, but for maps that carry around and update a state direction is important. For *separate* we could have used $put' = SA\ (\lambda(a, xs) \rightarrow ((), xs +\!\!+ [a]))$ and the left to right traversal, *pmapAl*, but it turns out that the (somewhat counterintuitive) right to left traversal with *put* is lazier, more efficient and easier to prove correct.

7.2.2 Function *combine*

Using the left to right traversing variant of the arrow map, *pmapAl*, we can write the inverse of *separate*, called *combine*, as follows.

$$\begin{array}{lcl} combine & :: & d\ () \rightsquigarrow_{[a]} d\ a \\ combine & = & pmapAl\ get \end{array}$$

$$\begin{array}{lcl} get & :: & () \rightsquigarrow_{[a]} a \\ get & = & SA\ (\lambda((), a : as) \rightarrow (a, as)) \end{array}$$

$$pmapAl :: (a \rightsquigarrow_s b) \rightarrow (d\ a \rightsquigarrow_s d\ b)$$

It remains to define the arrow maps, and to prove that *combine* is the inverse of *separate*, that is, *separate* followed by *combine* is the identity. Note that, due to the constructor *SA*, we cannot use normal function composition for values of type $a \rightsquigarrow_s b$. Instead we define a new composition operator $(\ggg)$:

$$\begin{array}{l} (\ggg) :: (a \rightsquigarrow_s b) \rightarrow (b \rightsquigarrow_s c) \rightarrow (a \rightsquigarrow_s c) \\ SA\ f \ggg SA\ g = SA\ (f\ ;\ g) \end{array}$$

It is easy to see that *get* is the inverse of *put*, but we include the proof as a reminder of the notation we use for calculational equality proofs.

$$\begin{array}{ll} & put \ggg get \\ = & \quad \text{Definitions of } get \text{ and } put \\ & SA\ (\lambda(a, xs) \rightarrow ((), a : xs)) \ggg SA\ (\lambda((), a : xs) \rightarrow (a, xs)) \\ = & \quad \text{Definition of } (\ggg) \\ & SA\ ((\lambda(a, xs) \rightarrow ((), a : xs))\ ;\ (\lambda((), a : xs) \rightarrow (a, xs))) \\ = & \quad \text{Simplification} \\ & SA\ id \end{array}$$

Here $SA\ id\ ::\ a \rightsquigarrow_{[a]} a$ is the identity on *SA* and the operator $(\ =\ )$ is equality on *SA*.

7.2.3 Function *combine* is the inverse of *separate*

The main ingredient of the proof that *combine* is the inverse of *separate* is a law about inverting arrow maps. More specifically, we have that *pmapAl* is the inverse of *pmapAr* provided the arguments of the maps are inverses:

$$\begin{array}{lll} pmapAr & :: & (a \leadsto_s b) \to (d\ a \leadsto_s d\ b) \\ pmapAl & :: & (b \leadsto_s a) \to (d\ b \leadsto_s d\ a) \end{array}$$

$$f \ggg g = SA\ id \quad\Rightarrow\quad pmapAr\, f \ggg pmapAl\ g = SA\ id \tag{7.1}$$

Using this law (which is proved in Section 7.4) we have:

$$\begin{array}{ll} & separate \ggg combine \\ = & \qquad \text{Definitions of } separate \text{ and } combine \\ & pmapAr\ put \ggg pmapAl\ get \\ = & \qquad \text{Law (7.1); } put \ggg get = SA\ id \\ & SA\ id \end{array}$$

This proves the correctness of functions *separate* and *combine*.

7.3 Arrows and laws

This section generalizes the type constructor $SA\ s$ to Hughes' abstract class for *arrows* [42]. The arrow class can be seen as a minimal signature of an embedded domain specific language as described by Paterson [89]. For additional motivation and background for using arrows, see the papers by Hughes and Paterson [42, 89]. We use a hierarchy of arrow classes as embedded domain specific languages for expressing data conversion programs. We introduce the arrow combinators together with example implementations for the $SA\ s$ arrow. In definitions and laws that hold for arbitrary arrows we write $a \leadsto b$ instead of $a \leadsto_s b$.

7.3.1 Basic definitions and laws for arrows

To define the arrow maps and to prove (a generalization of) Law (7.1), we need a few combinators to construct and combine arrows (that is, values of type $a \leadsto b$), together with some laws that relate these combinators. The implementations are given for the type $a \leadsto_s b$ (that is, $SA\ s\ a\ b$) to exemplify a typical arrow type but as we will see later, the types and the laws for the combinators form the signature of a more general class of arrows. Thus, any program written using these combinators will automatically be parametrized over the instances of this class. The arrow class and the arrow combinators come from Hughes' arrow paper [42].

Lifting.

The function that lifts normal functions to functions that also take and return a state value is called *arr*.

$$\begin{array}{l} arr \;::\; (a \to b) \to (a \rightsquigarrow_s b) \\ arr\ f = SA\ (f \twoheadleftarrow id) \end{array}$$

We will often write $\overrightarrow{f}$ instead of $arr\ f$. Function arr is a functor from the category of types and functions to the category of types and arrows: it distributes over composition (and trivially preserves the identity).

$$\overrightarrow{f} \ggg \overrightarrow{g} = \overrightarrow{f\ ;\ g}$$

Arrow composition.

Composition of arrows (defined already in Section 7.2) satisfies the usual laws: it is associative, and $\overrightarrow{id}$ is its unit.

$$\begin{array}{l} \overrightarrow{id} \ggg f = f = f \ggg \overrightarrow{id} \\ (f \ggg g) \ggg h = f \ggg (g \ggg h) \end{array}$$

We denote reverse composition with $(\lll)$, where $f \lll g = g \ggg f$.

Arrows between pairs.

Function *first* applies an arrow to the first component of a pair, leaving the second component unchanged.

$$\begin{array}{l} \mathit{first} \;::\; (a \rightsquigarrow_s b) \to ((a, c) \rightsquigarrow_s (b, c)) \\ \mathit{first}\ (SA\ f) = SA\ (\lambda((a, c), s) \to \mathbf{let}\ \ (b, s') = f\ (a, s) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ \ ((b, c), s')) \end{array}$$

Function *first* is a functor, that is, it preserves (arrow) identities and distributes over (arrow) composition.

$$\begin{array}{l} \mathit{first}\ \overrightarrow{f} = \overrightarrow{f \twoheadleftarrow id} \\ \mathit{first}\ (f \ggg g) = \mathit{first}\ f \ggg \mathit{first}\ g \end{array}$$

The dual function *second* that applies an arrow to the second component of a pair, can be defined in terms of *first* :

$$
\begin{aligned}
&\mathit{second} \; :: \; (a \rightsquigarrow b) \rightarrow ((c,a) \rightsquigarrow (c,b)) \\
&\mathit{second}\ f = \overrightarrow{\mathit{swap}} \ggg \mathit{first}\ f \ggg \overrightarrow{\mathit{swap}}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{swap} \; :: \; (a,b) \rightarrow (b,a) \\
&\mathit{swap}\ (a,b) = (b,a)
\end{aligned}
$$

Using *first* and *second* we can define two candidates for product functors, but when the arrows have side-effects, neither of these are functors because they fail to preserve composition.

$$
\begin{aligned}
(\ast\!\!\!\gt), (\lt\!\!\!\ast) \;&::\; (a \rightsquigarrow c) \rightarrow (b \rightsquigarrow d) \rightarrow ((a,b) \rightsquigarrow (c,d)) \\
f \ast\!\!\!\gt g \;&=\; \mathit{first}\ f \ggg \mathit{second}\ g \\
f \lt\!\!\!\ast g \;&=\; \mathit{second}\ g \ggg \mathit{first}\ f
\end{aligned}
$$

If one of the two arguments of *first* and *second* is side effect free (doesn't change the state), then *first* commutes with *second* . The canonical form of a side effect free arrow is $\overrightarrow{j}$ for some function j.

$$
\mathit{first}\ \overrightarrow{f} \ggg \mathit{second}\ g = \mathit{second}\ g \ggg \mathit{first}\ \overrightarrow{f}
$$

Arrows with a choice.

We can view the arrow combinators as a very small embedded language. With the combinators defined thus far we can embed functions as arrows using $\overrightarrow{\cdot}$, we can plug arrows together using $(\ggg)$ and we can simulate a value environment by using *first* , *second* etc. However, we cannot write conditionals — there is no way to choose between different branches depending on the input.

We lift the operator $(\triangledown) \; :: \; (a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow (\mathit{Either}\ a\ b \rightarrow c)$ to the arrow level to model a choice between different arrow branches. For state arrows the implementation is straightforward:

$$
\begin{aligned}
&(|||) \; :: \; (a \rightsquigarrow_s c) \rightarrow (b \rightsquigarrow_s c) \rightarrow (\mathit{Either}\ a\ b \rightsquigarrow_s c) \\
&\mathit{SA}\ f\ |||\ \mathit{SA}\ g = \mathit{SA}\ (\lambda(x,s) \rightarrow ((\lambda a \rightarrow f\ (a,s)) \triangledown (\lambda b \rightarrow g\ (b,s)))\ x)
\end{aligned}
$$

As a simple exercise in arrow plumbing we define **if**-expressions:

$$
\begin{aligned}
&\mathit{ifA} \; :: \; (a \rightsquigarrow \mathit{Bool}) \rightarrow (a \rightsquigarrow b) \rightarrow (a \rightsquigarrow b) \rightarrow (a \rightsquigarrow b) \\
&\mathit{ifA}\ p\ t\ e = \overrightarrow{\mathit{dup}} \ggg \mathit{first}\ p \ggg \overrightarrow{\mathit{bool2Either}} \ggg (t\ |||\ e) \\
&\quad \textbf{where}\ \mathit{dup}\ a = (a,a) \\
&\qquad\qquad \mathit{bool2Either}\ (b,x) = \textbf{if}\ b\ \textbf{then}\ \mathit{Left}\ x\ \textbf{else}\ \mathit{Right}\ x
\end{aligned}
$$

The lifted variant of operator $(+)$ for arrows is defined by:

$$(+\!\!+\!\!+) \;::\; (a \leadsto c) \rightarrow (b \leadsto d) \rightarrow (\mathit{Either}\ a\ b \leadsto \mathit{Either}\ c\ d)$$
$$f +\!\!+\!\!+ g = (f \ggg \overrightarrow{\mathit{Left}}) \mid\mid\mid (g \ggg \overrightarrow{\mathit{Right}})$$

Operator $(+\!\!+\!\!+)$ is a bifunctor on arrows — it preserves identities and distributes over composition.

$$\overrightarrow{f} +\!\!+\!\!+ \overrightarrow{g} = \overrightarrow{f + g}$$
$$(f +\!\!+\!\!+ g) \ggg (f' +\!\!+\!\!+ g') = (f \ggg f') +\!\!+\!\!+ (g \ggg g')$$

7.3.2 A class for arrows

The type $SA\ s\ a\ b$ encapsulates functions from a to b that manipulate a state of type s. However, most of the programs and laws we want to express don't refer to the state. Therefore, we go one step further in the abstraction by introducing the Haskell constructor class *Arrow* [42, 89]. An arrow type constructor $(\leadsto)$ is any two-parameter type constructor that supports the operations of the class *Arrow*. We require a number of laws to hold for the instances of the arrow class and for documentation purposes, we include these laws in the class definition although they can't be directly expressed in Haskell.

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Arrow}\ (\leadsto)\ \mathbf{where} \\
\quad \begin{array}{lll}
\mathit{arr} & :: & (a \rightarrow b) \rightarrow (a \leadsto b) \\
(\ggg) & :: & (a \leadsto b) \rightarrow (b \leadsto c) \rightarrow (a \leadsto c) \\
\mathit{first} & :: & (a \leadsto b) \rightarrow ((a, c) \leadsto (b, c))
\end{array} \\
\\
\quad \texttt{--}\ \mathit{Laws}: \\
\quad \overrightarrow{f} \ggg \overrightarrow{g} = \overrightarrow{f\ ;\ g} \\
\quad \overrightarrow{id} \ggg f = f = f \ggg \overrightarrow{id} \\
\quad (f \ggg g) \ggg h = f \ggg (g \ggg h) \\
\quad \mathit{first}\ \overrightarrow{f} = \overrightarrow{f * id} \\
\quad \mathit{first}\ (f \ggg g) = \mathit{first}\ f \ggg \mathit{first}\ g \\
\quad \mathit{first}\ \overrightarrow{f} \ggg \mathit{second}\ g = \mathit{second}\ g \ggg \mathit{first}\ \overrightarrow{f}
\end{array}
$$

For arrows with a choice operator, $(\mid\mid\mid)$, we define the subclass *ArrowChoice*. We include both the operator $(\mid\mid\mid)$ and $(+\!\!+\!\!+)$, but it is sufficient to define either of them in every instance because of the defaults. The default declarations are part of the Haskell class definition and can be seen as laws with immediate implementations.

$$
\begin{array}{l}
\textbf{class } \mathit{Arrow}\ (\rightsquigarrow)\ \Rightarrow\ \mathit{ArrowChoice}\ (\rightsquigarrow)\ \textbf{where} \\
\quad \begin{array}{lll}
(+\!\!+\!\!+) & :: & (a \rightsquigarrow c) \to (b \rightsquigarrow d) \to (\mathit{Either}\ a\ b \rightsquigarrow \mathit{Either}\ c\ d) \\
(|||) & :: & (a \rightsquigarrow c) \to (b \rightsquigarrow c) \to (\mathit{Either}\ a\ b \rightsquigarrow c)
\end{array} \\
\quad \texttt{--}\ \mathit{Defaults}: \\
\quad \begin{array}{lll}
f +\!\!+\!\!+ g & = & (f \ggg \overrightarrow{\mathit{Left}})\ |||\ (g \ggg \overrightarrow{\mathit{Right}}) \\
f\ |||\ g & = & (f +\!\!+\!\!+ g) \ggg \overrightarrow{\mathit{id} \triangledown \mathit{id}}
\end{array} \\
\quad \texttt{--}\ \mathit{Laws}: \\
\quad \overrightarrow{f} +\!\!+\!\!+ \overrightarrow{g} = \overrightarrow{f + g} \\
\quad (f +\!\!+\!\!+ g) \ggg (f' +\!\!+\!\!+ g') = (f \ggg f') +\!\!+\!\!+ (g \ggg g') \\
\quad (f\ |||\ g) \ggg h = (f \ggg h)\ |||\ (g \ggg h)
\end{array}
$$

The type constructor $\mathit{SA}\ s$ is made an instance of *Arrow* and *ArrowChoice* by taking the definitions of arr, $(\ggg)$, first, $(|||)$ and $(+\!\!+\!\!+)$ from Section 7.3.1. Normal functions are trivially *Arrow*s and they support choice:

$$
\begin{array}{l}
\textbf{instance } \mathit{Arrow}\ (\to)\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{arr}\ f & = & f \\
f \ggg g & = & f\ ;\ g \\
\mathit{first}\ f & = & f \times \mathit{id}
\end{array} \\
\textbf{instance } \mathit{ArrowChoice}\ (\to)\ \textbf{where} \\
\quad \begin{array}{lll}
f +\!\!+\!\!+ g & = & f + g \\
f\ |||\ g & = & f \triangledown g
\end{array}
\end{array}
$$

With the definitions from these instances, three of the laws from the *Arrow* and the *ArrowChoice* classes can be rewritten to a form which more clearly indicates that $\overrightarrow{\cdot}$ lifts composition, first and choice from normal functions to arrows:

$$
\begin{array}{rcl}
\overrightarrow{f} \ggg \overrightarrow{g} & = & \overrightarrow{f \ggg g} \\
\mathit{first}\ \overrightarrow{f} & = & \overrightarrow{\mathit{first}\ f} \\
\overrightarrow{f} +\!\!+\!\!+ \overrightarrow{g} & = & \overrightarrow{f +\!\!+\!\!+ g}
\end{array}
$$

Many side effecting computations can be captured by the *Arrow* signature, including all functions returning monadic results: we can define a *Kleisli* arrow for every Haskell *Monad* [102]:

$$
\begin{array}{l}
\textbf{newtype } \mathit{Kleisli}\ m\ a\ b = \mathit{Kleisli}\ (a \to m\ b) \\
\textbf{instance } \mathit{Monad}\ m \Rightarrow \mathit{Arrow}\ (\mathit{Kleisli}\ m)\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{arr}\ f & = & \mathit{Kleisli}\ (\lambda a \to \mathit{return}\ (f\ a)) \\
\mathit{Kleisli}\ f \ggg \mathit{Kleisli}\ g & = & \mathit{Kleisli}\ (\lambda a \to f\ a \gg\!= g) \\
\mathit{first}\ (\mathit{Kleisli}\ f) & = & \mathit{Kleisli}\ (\lambda(a, c) \to f\ a \gg\!= \lambda b \to \mathit{return}\ (b, c))
\end{array} \\
\textbf{instance } \mathit{Monad}\ m \Rightarrow \mathit{ArrowChoice}\ (\mathit{Kleisli}\ m)\ \textbf{where} \\
\quad \mathit{Kleisli}\ f\ |||\ \mathit{Kleisli}\ g\ =\ \mathit{Kleisli}\ (f \triangledown g)
\end{array}
$$

7.3.3 An inverse law for arrow products

The two product operators ($\ast\!\!\gg$) and ($\ll\!\!\ast$) are inverses in a certain sense:

$$(f \ast\!\!\gg g)^{-1} \;=\; f^{-1} \ll\!\!\ast\, g^{-1}$$

We will prove this equality in a slightly more general form, which will turn out to be useful in the following sections. We generalize the inverse law by weakening the inverse requirement to require only the side-effects to be inverses. If $f \ggg f' = \overrightarrow{i}$, then the arrow f' un-does the side-effects of the arrow f, leaving just a side-effect free computation $\overrightarrow{i}$. If i is chosen to be id, then we regain the usual (left-) inverse concept. The more general inverse concept will be used in the rest of the chapter. The generalized inverse law for the product operators is:

$$\begin{array}{lll} (\ll\!\!\ast) & :: & (a \leadsto c) \to (b \leadsto d) \to ((a,b) \leadsto (c,d)) \\ (\ast\!\!\gg) & :: & (c \leadsto a) \to (d \leadsto b) \to ((c,d) \leadsto (a,b)) \end{array}$$

$$f \ggg f' = \overrightarrow{i} \;\Rightarrow\; g \ggg g' = \overrightarrow{j} \;\Rightarrow\; (f \ll\!\!\ast\, g) \ggg (f' \ast\!\!\gg g') = \overrightarrow{i \ast j} \tag{7.2}$$

Perhaps a word on notation is appropriate here. We present the types of the product operators together with the inverse law, to stress that we are not dealing with just a pair of inverse functions, but rather with a triple containing two functions and a proof that they are inverses. We take a curried view of functions with two arguments, that is, they have type $a \to b \to c$ rather than $(a, b) \to c$. Similarly, we prefer to write a proof term with two premises as $P \Rightarrow Q \Rightarrow R$, instead of the more traditional $P \wedge Q \;\Rightarrow\; R$. Thus we stress that the components of the triple share the same structure: they take two arrows (two proofs) and return an arrow (a proof).

Proof: We assume $f \ggg f' = \overrightarrow{i}$ and $g \ggg g' = \overrightarrow{j}$ and calculate as follows:

$$\begin{array}{ll} & (f \ll\!\!\ast\, g) \ggg (f' \ast\!\!\gg g') \\ = & \quad \text{Definitions of } \ll\!\!\ast \text{ and } \ast\!\!\gg \\ & \mathit{second}\ g \ggg \mathit{first}\ f \ggg \mathit{first}\ f' \ggg \mathit{second}\ g' \\ = & \quad \mathit{first} \text{ is a functor} \\ & \mathit{second}\ g \ggg \mathit{first}\ (f \ggg f') \ggg \mathit{second}\ g' \\ = & \quad \text{Assumption 1} \\ & \mathit{second}\ g \ggg \mathit{first}\ \overrightarrow{i} \ggg \mathit{second}\ g' \\ = & \quad \overrightarrow{i} \text{ is side-effect free} \\ & \mathit{first}\ \overrightarrow{i} \ggg \mathit{second}\ g \ggg \mathit{second}\ g' \\ = & \quad \mathit{second} \text{ is a functor} \end{array}$$

$$
\begin{array}{ll}
& \mathit{first}\ \overrightarrow{i} \ggg \mathit{second}\ (g \ggg g') \\
= & \quad \text{Assumption 2} \\
& \mathit{first}\ \overrightarrow{i} \ggg \mathit{second}\ \overrightarrow{j} \\
= & \quad \mathit{first}\ ,\ \mathit{second}\ \text{ and } (\ggg) \text{ preserve } \overrightarrow{\cdot} \\
& \overrightarrow{(i \twoheadrightarrow id)\ ;\ (id \twoheadrightarrow j)} \\
= & \quad (\twoheadrightarrow) \text{ is a bifunctor} \\
& \overrightarrow{i \twoheadrightarrow j}
\end{array}
$$

□

7.3.4 Fixed point induction and arrows

Section 7.4 proves an inverse law (7.4) for arrow maps. A similar law for normal maps can be proved with the fusion law for catamorphisms. This fusion law is derived from the fact that datatypes are defined as initial functor-algebras. A catamorphism on arrows is defined in terms of the function TR, but because TR is not a functor, we cannot prove, let alone apply, a fusion law. In the proof of the law for arrow maps we will use instead the fixed point induction theorem from Section 2.11.1. We can instantiate Theorem 2.15 to a form that is more suitable for our purposes by letting $n = 3$ and the (inclusive) relation $P(x, y, z) = x \ggg y = \overrightarrow{z}$. The instance takes the following form:

$$
\begin{array}{ll}
& (p' \ggg u' = \overrightarrow{i'} \Rightarrow f\ p' \ggg g\ u' = \overrightarrow{h\ i'}) \\
\Rightarrow & \mathit{fix}\ f \ggg \mathit{fix}\ g = \overrightarrow{\mathit{fix}\ h}
\end{array}
\tag{7.3}
$$

where we have left out the proposition $\bot \ggg \bot = \overrightarrow{\bot}$ which is true for the arrows considered in this chapter.

7.4 Arrow maps

In Section 7.2, *separate* and *combine* were defined using the arrow maps *pmapAr* and *pmapAl*. The arrow maps can be seen as simple data conversion programs, which change the contents but leave the shape of the data unchanged. Using the arrow combinators from Section 7.3 we can now define the arrow maps, and prove a generalization of (7.1): if u is the inverse of p, then a left traversal with u is the inverse of a right traversal with p.

$$
\begin{array}{lcl}
\mathit{pmapAr} & :: & \mathit{ArrowChoice}\ (\leadsto)\ \Rightarrow\ (a \leadsto b) \to (d\ a \leadsto d\ b) \\
\mathit{pmapAl} & :: & \mathit{ArrowChoice}\ (\leadsto)\ \Rightarrow\ (a \leadsto b) \to (d\ a \leadsto d\ b)
\end{array}
$$

$$
p \ggg u = \overrightarrow{i} \quad \Rightarrow \quad \mathit{pmapAr}_d\ p \ggg \mathit{pmapAl}_d\ u = \overrightarrow{\mathit{pmap}_d\ i} \tag{7.4}
$$

$$
\begin{array}{l}
\textbf{polytypic}\ \textsc{Tr}_f :: (a \leadsto c) \to (b \leadsto d) \to (f\ a\ b \leadsto f\ c\ d) \\
\quad = \lambda p\ r \to\ \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad \begin{array}{lll}
g + h & \longrightarrow & \textsc{Tr}_g\ p\ r \ggg\!\!+ \textsc{Tr}_h\ p\ r \\
g * h & \longrightarrow & \textsc{Tr}_g\ p\ r \lessdot\!\!* \textsc{Tr}_h\ p\ r \\
\mathit{Empty} & \longrightarrow & \overrightarrow{id} \\
\mathit{Par} & \longrightarrow & p \\
\mathit{Rec} & \longrightarrow & r \\
d\ @\ g & \longrightarrow & \mathit{pmapAr}_d\ (\textsc{Tr}_g\ p\ r) \\
\mathit{Const}\ t & \longrightarrow & \overrightarrow{id}
\end{array}
\end{array}
$$

Figure 7.1: The definition of $\textsc{Tr}$

The definitions of the arrow maps are obtained by a straightforward generalization of *pmap* to arrows.

$$
\begin{array}{lll}
\mathit{pmapAr}_d\ p & = & \overrightarrow{\mathit{out}_d} \ggg \textsc{Tr}_{\Phi_d}\ p\ (\mathit{pmapAr}_d\ p) \ggg \overrightarrow{\mathit{inn}_d} \\
\mathit{pmapAl}_d\ p & = & \overrightarrow{\mathit{out}_d} \ggg \textsc{Tl}_{\Phi_d}\ p\ (\mathit{pmapAl}_d\ p) \ggg \overrightarrow{\mathit{inn}_d}
\end{array}
$$

Functions $\textsc{Tr}$ and $\textsc{Tl}$ are the corresponding generalizations of *fmap2*. All functions used in the definition of *fmap2* are lifted to the arrow level. For all cases except the product functor case there is only one choice for a reasonable lifting, but when we lift the operator $(-\!\!*\!\!-)$ we have two possible choices: $(\lessdot\!\!*)$ and $(*\!\!\gtrdot)$. This is the only difference between two two traversal functions: the right to left traversal, $\textsc{Tr}$, uses $(\lessdot\!\!*)$ and the left to right traversal, $\textsc{Tl}$, uses $(*\!\!\gtrdot)$. Function $\textsc{Tr}$ is defined in Figure 7.1 and function $\textsc{Tl}$ is completely analogous and therefore omitted. Functions $\textsc{Tr}$ and $\textsc{Tl}$ satisfy the following inverse law:

$$
\begin{array}{l}
\textsc{Tr}_f :: (a \leadsto c) \to (b \leadsto d) \to (f\ a\ b \leadsto f\ c\ d) \\
\textsc{Tl}_f :: (c \leadsto a) \to (d \leadsto b) \to (f\ c\ d \leadsto f\ a\ b)
\end{array}
$$

$$
p \ggg u = \overrightarrow{i} \ \Rightarrow\ p' \ggg u' = \overrightarrow{i'} \ \Rightarrow\ \textsc{Tr}_f\ p\ p' \ggg \textsc{Tl}_f\ u\ u' = \overrightarrow{\mathit{fmap2}_f\ i\ i'} \tag{7.5}
$$

Note the close correspondence between this law and the inverse law for the product operators (7.2).

7.4.1 The arrow maps are inverses

The proof of Equation (7.4) can be interpreted either as fusing *pmapAr* p with *pmapAl* u to get a pure arrow $\overrightarrow{\mathit{pmap}_d\ i}$ or, equivalently, as splitting the function $\mathit{pmap}_d\ i$ into a composition of two arrow maps. We use induction over the structure of a regular datatype $d\ a$. As the grammars for datatypes and pattern functors are mutually recursive we get two induction hypotheses. The datatype level hypothesis is, that Equation (7.4) holds for datatypes defined earlier, and the pattern functor level hypothesis is, that Equation (7.5) holds for the subfunctors. We rewrite the definitions of *pmapAr*, *pmapAl* and *pmap* to expose the top level fixed points:

$$
\begin{array}{rcl}
\mathit{pmapAr}\ p & = & \mathit{fix}\ (\lambda p' \to \overrightarrow{\mathit{out}} \ggg \textsc{Tr}\ p\ p' \ggg \overrightarrow{\mathit{inn}}) \\
\mathit{pmapAl}\ u & = & \mathit{fix}\ (\lambda u' \to \overrightarrow{\mathit{out}} \ggg \textsc{Tl}\ u\ u' \ggg \overrightarrow{\mathit{inn}}) \\
\mathit{pmap}\ i & = & \mathit{fix}\ (\lambda i' \to \mathit{out}\ ;\ \mathit{fmap2}\ i\ i'\ ;\ \mathit{inn})
\end{array}
$$

We assume $p \ggg u = \overrightarrow{i}$ and calculate as follows:

$$
\begin{array}{ll}
 & \mathit{pmapAr}\ p \ggg \mathit{pmapAl}\ u = \overrightarrow{\mathit{pmap}\ i} \\
\Leftarrow & \qquad \text{Definitions of } \mathit{pmapAr},\ \mathit{pmapAl}, \text{ fixed point law (7.3)} \\
 & p' \ggg u' = \overrightarrow{i'} \Rightarrow \\
 & \overrightarrow{\mathit{out}} \ggg \textsc{Tr}\ p\ p' \ggg \overrightarrow{\mathit{inn}} \ggg \overrightarrow{\mathit{out}} \ggg \textsc{Tl}\ u\ u' \ggg \overrightarrow{\mathit{inn}} = \\
 & \overrightarrow{\mathit{out}\ ;\ \mathit{fmap2}\ i\ i'\ ;\ \mathit{inn}} \\
\equiv & \qquad \overrightarrow{f} \ggg \overrightarrow{g} = \overrightarrow{f ; g},\ \mathit{inn}\ ;\ \mathit{out} = \mathit{id},\ f \ggg \overrightarrow{\mathit{id}} = f \\
 & p' \ggg u' = \overrightarrow{i'} \Rightarrow \\
 & \overrightarrow{\mathit{out}} \ggg \textsc{Tr}\ p\ p' \ggg \textsc{Tl}\ u\ u' \ggg \overrightarrow{\mathit{inn}} = \overrightarrow{\mathit{out}} \ggg \overrightarrow{\mathit{fmap2}\ i\ i'} \ggg \overrightarrow{\mathit{inn}} \\
\Leftarrow & \qquad \text{Law (7.5) and the assumption: } p \ggg u = \overrightarrow{i} \\
 & \mathit{True}
\end{array}
$$

We prove Law (7.5) by induction over the structure of the pattern functor f. Because there are seven constructors for functors, we have to verify seven cases. Although this is laborious, we want to show at least one complete proof of a statement about polytypic functions.

The sum case, $g + h$**:**

$$
\begin{array}{ll}
 & \textsc{Tr}_{g+h}\ p\ p' \ggg \textsc{Tl}_{g+h}\ u\ u' \\
= & \qquad \text{Definitions} \\
 & (\textsc{Tr}_g\ p\ p' \mathbin{+\!\!\!+\!\!\!+} \textsc{Tr}_h\ p\ p') \ggg (\textsc{Tl}_g\ u\ u' \mathbin{+\!\!\!+\!\!\!+} \textsc{Tl}_h\ u\ u') \\
= & \qquad (\mathbin{+\!\!\!+\!\!\!+}) \text{ is a bifunctor} \\
 & (\textsc{Tr}_g\ p\ p' \ggg \textsc{Tl}_g\ u\ u') \mathbin{+\!\!\!+\!\!\!+} (\textsc{Tr}_h\ p\ p' \ggg \textsc{Tl}_h\ u\ u') \\
= & \qquad \text{Induction hypothesis 7.5 (twice)} \\
 & \overrightarrow{\mathit{fmap2}_g\ i\ i'} \mathbin{+\!\!\!+\!\!\!+} \overrightarrow{\mathit{fmap2}_h\ i\ i'} \\
= & \qquad \overrightarrow{f} \mathbin{+\!\!\!+\!\!\!+} \overrightarrow{g} = \overrightarrow{f + g}, \text{ definition of } \mathit{fmap2}_{g+h} \\
 & \overrightarrow{\mathit{fmap2}_{g+h}\ i\ i'}
\end{array}
$$

The product case, $g * h$:

$$
\begin{array}{ll}
& \text{TR}_{g*h}\ p\ p' \ggg \text{TL}_{g*h}\ u\ u' \\
= & \quad \text{Definitions} \\
& (\text{TR}_g\ p\ p' \lessdot\!\!* \ \text{TR}_h\ p\ p') \ggg (\text{TL}_g\ u\ u' \ *\!\!\gtrdot \ \text{TL}_h\ u\ u') \\
= & \quad \text{Inverse law for products (7.2), induction hypothesis (7.5) (twice)} \\
& \overrightarrow{\mathit{fmap2}_g\ i\ i' \ast \mathit{fmap2}_h\ i\ i'} \\
= & \quad \text{definition of } \mathit{fmap2}_{g*h} \\
& \overrightarrow{\mathit{fmap2}_{g*h}\ i\ i'}
\end{array}
$$

The empty case, *Empty*:

$$
\begin{array}{ll}
& \text{TR}_{\mathit{Empty}}\ p\ p' \ggg \text{TL}_{\mathit{Empty}}\ u\ u' \\
= & \quad \text{Definitions} \\
& \overrightarrow{id} \ggg \overrightarrow{id} \\
= & \quad \overrightarrow{id} \text{ is the unit of } \ggg \\
& \overrightarrow{id} \\
= & \quad \text{Definition of } \mathit{fmap2}_{\mathit{Empty}} \\
& \overrightarrow{\mathit{fmap2}_{\mathit{Empty}}\ i\ i'}
\end{array}
$$

The constant case *Const* t is proved in exactly the same way as the empty case; the calculation is omitted.

The parameter case, *Par*:

$$
\begin{array}{ll}
& \text{TR}_{\mathit{Par}}\ p\ p' \ggg \text{TL}_{\mathit{Par}}\ u\ u' \\
= & \quad \text{Definitions} \\
& p \ggg u \\
= & \quad \text{Assumption} \\
& \overrightarrow{i} \\
= & \quad \text{Definition of } \mathit{fmap2}_{\mathit{Par}} \\
& \overrightarrow{\mathit{fmap2}_{\mathit{Par}}\ i\ i'}
\end{array}
$$

The recursive case, *Rec*, is proved in exactly the same way as the parameter case; the calculation is omitted.

The composition case, d @ g:

$$
\begin{array}{ll}
& \text{TR}_{d@g}\ p\ p' \ggg \text{TL}_{d@g}\ u\ u' \\
= & \quad \text{Definitions}
\end{array}
$$

$$
\begin{array}{ll}
 & pmapAr_d\ (\mathrm{T}\textsc{r}_g\ p\ p') \ggg pmapAl_d\ (\mathrm{T}\textsc{l}_g\ u\ u') \\
= & \left\{\begin{array}{l}\text{The top level induction hypothesis (7.4) is} \\ f \ggg g = \overrightarrow{h} \quad \Rightarrow \quad pmapAr\ f \ggg pmapAl\ g = \overrightarrow{pmap\ h} \\ \text{where we take } f = \mathrm{T}\textsc{r}_g\ p\ p',\ g = \mathrm{T}\textsc{l}_g\ u\ u' \text{ and } h = fmap2\ i\ i' \\ \text{and induction hypothesis (7.5) is precisely } f \ggg g = \overrightarrow{h}.\end{array}\right. \\
 & \overrightarrow{pmap_d\ (fmap2_g\ i\ i')} \\
= & \quad \text{Definition of } fmap2_{d@g} \\
 & \overrightarrow{fmap2_{d@g}\ i\ i'}
\end{array}
$$

This concludes the proof.

In the conclusions we will spend some words on (how to simplify) proving statements about polytypic functions.

7.5 Packing

This section sketches the construction and correctness proof of a polytypic packing program. The basic idea of the packing program is simple: given a datatype value (an abstract syntax tree), construct a compact (bit stream) representation of the abstract syntax tree. For example, the following rather artificial binary tree, called *treeShape* in the introductory section,

$$
\begin{array}{lll}
treeShape & :: & Tree\ () \\
treeShape & = & Bin\ (Bin\ (Leaf\ ())\ (Bin\ (Leaf\ ())\ (Leaf\ ())))\ (Leaf\ ())
\end{array}
$$

can be pretty-printed to a text representation of *treeShape* requiring 55 bytes. However, because the datatype *Tree a* has only two constructors, each constructor can be represented by a single bit. Furthermore, the datatype () has only one constructor, so the single element (also written ()) can be represented by 0 bits. Thus we get the following representations:

```
Bin (Bin (Leaf ()) (Bin (Leaf ()) (Leaf ()))) (Leaf ())
 1    1    0         1    0          0           0
```

The compact representation consists of 7 bits, so only 1 byte is needed to store this tree. In fact, the pretty-printed text of a value of type *Tree* () is asymptotically 64 times bigger than the compact representation.[3] Of course, this is an unusually simple datatype, but the average case is still very compact.

[3] A value of type *Tree* () with n leaves has $n-1$ internal nodes. A leaf is printed as the seven character string `"Leaf ()"` and a node as `"Bin ("`, left subtree, `") ("`, right subtree, `")"` — a total of nine characters per node. Thus the pretty printed string representation of a tree contains exactly $7n+9(n-1) = 16n-9$ bytes while the compact representation with one bit per constructor contains $2n-1$ bits. The ratio is then $8(16n-9)/(2n-1) \approx 8(16n-8)/(2n-1) = 64$.

Given a datatype value, the polytypic packing function prepends the compact representation of the value to a state, on which it side effects. Let *Text* be the type of packed values, for example *String* or [*Bit*]. Then the packing function can be implemented using the state arrow type constructor *SA Text*, but we will keep the arrow type abstract and only require that it supports packing of constructors.

To pack a value of type $d\ a$ we need a function that can pack values of type a. We could use *separate* and *combine* to reduce the packing problem to packing the structure and the contents separately, but instead we parametrize on the element level (un)packing function. With Hinze style polytypism [35], this parametrization comes for free.

Our goal is to construct two functions and a proof:

- A function *ppack* ('polytypic packing') that takes an element level packer to a datatype level packer.

  $$ppack \ :: \ (a \leadsto ()) \to (d\ a \leadsto ())$$

  For example, the function that packs the tree *treeShape* :: *Tree* () is obtained by instantiating the polytypic function *ppack* on *Tree* and applying the instance to a (trivial) packing program for the type ().

- A function *punpack* ('polytypic unpacking') that takes an unpacker on the element level a to an unpacker on the datatype level $d\ a$:

  $$punpack \ :: \ (() \leadsto a) \to (() \leadsto d\ a)$$

  For the *Tree* example the element level parsing program is a function that parses nothing, and returns (), the value of type ().

- A proof that if p and u are inverses on the element level a, then *ppack* p and *punpack* u are inverses on the datatype level $d\ a$.

Representing constructors.

To construct the printer and the parser we need a little more structure than provided by the *Arrow* class – we need a way of handling constructors. Because a constructor can be coded by a single natural number, we can use a class *ArrowPack* to characterize arrows that have operations for printing and parsing constructor numbers:

$$
\begin{array}{l}
\textbf{class}\ ArrowChoice\ (\leadsto)\ \Rightarrow\ ArrowPack\ (\leadsto)\ \textbf{where}\\
\quad packCon \quad :: \quad Nat \leadsto ()\\
\quad unpackCon \ :: \quad () \leadsto Nat\\
\\
\quad \texttt{--}\ Laws:\\
\quad packCon \ggg unpackCon = \overrightarrow{id_{Nat}}
\end{array}
$$

With $Text = [Nat]$, the instances for $SA\ Text$ are just put and get from Section 7.2, and the printing algorithm constructed in the following section will in its simplest form just output a list of numbers given an argument tree of any type. A better solution is to code these numbers as bits and here we have some choices on how to proceed. We could decide on a fixed maximal size for numbers and store them using their binary representation but, as most datatypes have few constructors, this would waste space. We will instead statically determine the number of constructors in the datatype and code every single number in only as many bits as needed. For an n-constructor datatype we use just $\lceil \log_2 n \rceil$ bits to code a constructor. An interesting effect of this coding is that the constructor of any single constructor datatype will be coded using 0 bits! We obtain better results if we use Huffman coding with equal probabilities for the constructors, resulting in a variable number of bits per constructor. Even better results are obtained if we analyze the datatype, and give different probabilities to the different constructors. However, our goal is not to squeeze the last bit out of our data, but rather to show how to construct the polytypic program. Because the number of bits used per constructor depends on the type of the value that is compressed, *packCon* and *unpackCon* need in general be polytypic functions. Their definitions are omitted, but can be found in the code on the web page for this dissertation.

In the rest of this section $(\leadsto)$ will always stand for an arrow type constructor in the class *ArrowPack* but, as with *Regular*, we often omit the type context for brevity.

7.5.1 The construction of the packing function

We construct a printing function *ppack*, which promotes an element level packer to a datatype level packer, together with a parsing function *punpack*, which similarly promotes an unpacker to the datatype level. If the element level arguments are inverses, then we want *punpack* to be the inverse of *ppack*:

$$\begin{aligned} ppack &:: (a \leadsto ()) \to (d\ a \leadsto ()) \\ punpack &:: (() \leadsto a) \to (() \leadsto d\ a) \end{aligned}$$

$$p \ggg u = \overrightarrow{i} \;\Rightarrow\; ppack\ p \ggg punpack\ u = \overrightarrow{pmap\ i} \qquad (7.6)$$

In the following proofs we will assume that the argument packer p and the unpacker u satisfy $p \ggg u = \overrightarrow{i}$.

Overview of the construction.

Again, the construction can be interpreted as fusing the 'printer' *ppack* with the 'parser' *punpack* to get a pure arrow $\overrightarrow{pmap\ i}$. As we are defining polytypic functions the construction follows the structure of regular datatypes: a regular datatype is a fixed point of a pattern functor, the pattern functor is a sum of products of type terms, and the terms can involve type parameters, other types, etc.

The arrow *ppack* p prints a compact representation of a value of type $d\ a$. It does this by recursing over the value, printing each constructor by computing its constructor number, and each element by using the argument printer p. The constructor number is computed by means of function Ps ('Pack Sum'), which also takes care of passing on the recursion to the children. An arrow *packCon* prints the constructor number with the correct number of bits. Finally, function Pp ('Pack Product') makes sure the information is correctly threaded through the children.

Top level recursion.

We want function *ppack* to be 'on-line' or lazy: it should output compactly printed data immediately, and given part of the compactly printed data, *punpack* should reconstruct part of the input value. Thus functions *ppack* and *punpack* can also be used to pack infinite streams, for example. Function *ppack* cannot be defined with a standard recursion operator such as the catamorphism because the side effecting arrows would be threaded in the wrong order. Instead of a recursion operator we use explicit recursion on the top level, guided by Pt ('Pack Top-level') and Ut ('Unpack Top-level').

As *ppack* decomposes its input value, and compactly prints the constructor and the children by means of a function Pt (defined later), *punpack* must do the opposite: first parse the components using Ut and then construct the top level value:

$$\begin{aligned}
\mathit{ppack}\ p &= \mathrm{P{\scriptstyle T}}\ p\ (\mathit{ppack}\ p) \lll \overrightarrow{\mathit{out}} \\
\mathit{punpack}\ u &= \mathrm{U{\scriptstyle T}}\ u\ (\mathit{punpack}\ u) \ggg \overrightarrow{\mathit{inn}}
\end{aligned}$$

Here $(\lll)$ is used to reveal the symmetry of the definitions. Thus we need two new functions, Pt and Ut, and we can already guess that we will need a corresponding fusion law:

$$\begin{aligned}
\mathrm{P{\scriptstyle T}} &:: (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow ()) \\
\mathrm{U{\scriptstyle T}} &:: (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (() \rightsquigarrow f\ a\ b)
\end{aligned}$$

$$p \ggg u = \overrightarrow{i} \;\Rightarrow\; p' \ggg u' = \overrightarrow{i'} \;\Rightarrow\; \mathrm{P{\scriptstyle T}}\ p\ p' \ggg \mathrm{U{\scriptstyle T}}\ u\ u' = \overrightarrow{\mathit{fmap2}\ i\ i'} \qquad (7.7)$$

Given (7.7) we can now prove (7.6).

$$
\begin{array}{ll}
& p \ggg u = \overrightarrow{i} \Rightarrow ppack\ p \ggg punpack\ u = \overrightarrow{pmap\ i} \\
\Leftarrow & \quad \text{Definitions of } ppack,\ punpack,\ pmap, \text{ fixed point law (7.3)} \\
& p \ggg u = \overrightarrow{i} \quad \Rightarrow \quad p' \ggg u' = \overrightarrow{i'} \quad \Rightarrow \\
& \overrightarrow{out} \ggg \textsc{Pt}\ p\ p' \ggg \textsc{Ut}\ u\ u' \ggg \overrightarrow{inn} = \overrightarrow{out\ ;\ fmap2\ i\ i'\ ;\ inn} \\
\Leftarrow & \quad \text{Equation (7.7), simplification} \\
& True
\end{array}
$$

Packing constructors.

We want to construct functions $\textsc{Pt}$ and $\textsc{Ut}$ such that (7.7) holds. Furthermore, these functions should do the actual packing and unpacking of the constructors using $packCon\ ::\ Nat \leadsto ()$ and $unpackCon\ ::\ () \leadsto Nat$ from the $ArrowPack$ class:

$$
\begin{array}{lcl}
\textsc{Pt}\ p\ p' & = & packCon \lll \textsc{Ps}\ p\ p' \\
\textsc{Ut}\ u\ u' & = & unpackCon \ggg \textsc{Us}\ u\ u'
\end{array}
$$

The arrow $\textsc{Ps}\ p\ p'$ packs a value (using the argument packers p and p' for the parameters and the recursive structures, respectively) and returns the number of the top level constructor, by determining the position of the constructor in the pattern functor (a sum of products). The arrow $packCon$ prepends the constructor number to the output. As $packCon \ggg unpackCon = \overrightarrow{id}$ by assumption, the requirement that function $\textsc{Pt}$ can be fused with $\textsc{Ut}$ is now passed on to $\textsc{Ps}$ and $\textsc{Us}$ ('Unpack Sum'):

$$
\begin{array}{lcl}
\textsc{Ps} & :: & (a \leadsto ()) \to (b \leadsto ()) \to (f\ a\ b \leadsto Nat) \\
\textsc{Us} & :: & (() \leadsto a) \to (() \leadsto b) \to (Nat \leadsto f\ a\ b)
\end{array}
$$

$$
p \ggg u = \overrightarrow{i} \;\Rightarrow\; p' \ggg u' = \overrightarrow{i'} \;\Rightarrow\; \textsc{Ps}\ p\ p' \ggg \textsc{Us}\ u\ u' = \overrightarrow{fmap2\ i\ i'} \tag{7.8}
$$

The arrow $unpackCon$ reads the constructor number and passes it on to the arrow $\textsc{Us}\ u\ u'$, which selects the desired constructor and uses its argument parsers u and u' to fill in the parameter and recursive component slots in the functor value.

Calculating constructor numbers.

The pattern functor of a Haskell datatype with n constructors is an n-ary sum (of products) on the outermost level. In PolyP this sum is represented by a nested binary sum, which associates to the right. Consequently, we define $\textsc{Ps}$ by induction over the nested sum part of the pattern functor and defer the handling of the product part to $\textsc{Pp}$ ('Pack Product'). (The definitions of inn_{Nat} and out_{Nat} are in Figure 7.2.)

```
data Nat = Z | S Nat

inn_Nat :: Either () Nat → Nat
inn_Nat = (const Z) ▿ S

out_Nat :: Nat   → Either () Nat
out_Nat    (Z)   = Left   ()
out_Nat    (S n) = Right     n
```

Figure 7.2: The definitions of inn_{Nat} and out_{Nat} as Haskell code.

$$
\begin{array}{l}
\textbf{polytypic } \textsc{Ps}_f :: (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow Nat) \\
\quad = \lambda p\ p' \to \textbf{case } f \textbf{ of} \\
\qquad\qquad\qquad g + h \longrightarrow \overrightarrow{inn_{Nat}} \lll (\textsc{Pp}\ p\ p' +\!\!\!+\!\!\!+ \textsc{Ps}\ p\ p') \\
\qquad\qquad\qquad g \qquad\ \longrightarrow \overrightarrow{\lambda() \to 0} \lll \textsc{Pp}\ p\ p'
\end{array}
$$

$$
\begin{array}{l}
\textbf{polytypic } \textsc{Us}_f :: (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (Nat \rightsquigarrow f\ a\ b) \\
\quad = \lambda u\ u' \to \textbf{case } f \textbf{ of} \\
\qquad\qquad\qquad g + h \longrightarrow \overrightarrow{out_{Nat}} \ggg (\textsc{Up}\ u\ u' +\!\!\!+\!\!\!+ \textsc{Us}\ u\ u') \\
\qquad\qquad\qquad g \qquad\ \longrightarrow \overrightarrow{\lambda 0 \to ()} \ggg \textsc{Up}\ u\ u'
\end{array}
$$

The types for $\textsc{Pp}$ and $\textsc{Up}$ ('Unpack Product') and the corresponding fusion law are unsurprising:

$$
\begin{array}{lll}
\textsc{Pp} & :: & (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow ()) \\
\textsc{Up} & :: & (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (() \rightsquigarrow f\ a\ b)
\end{array}
$$

$$
p \ggg u = \overrightarrow{i} \;\Rightarrow\; p' \ggg u' = \overrightarrow{i'} \;\Rightarrow\; \textsc{Pp}\ p\ p' \ggg \textsc{Up}\ u\ u' = \overrightarrow{fmap2\ i\ i'} \tag{7.9}
$$

We prove Equation (7.8) by induction over the nested sum structure of the functor. The induction hypothesis is that (7.8) holds for $\textsc{Ps}_h$.

The sum case, $g + h$:

$$
\begin{array}{ll}
 & \textsc{Ps}_{g+h}\ p\ p' \ggg \textsc{Us}_{g+h}\ u\ u' \\
= & \quad \text{Definitions} \\
 & (\textsc{Pp}_g\ p\ p' +\!\!\!+\!\!\!+ \textsc{Ps}_h\ p\ p') \ggg \overrightarrow{inn_{Nat}} \ggg \\
 & \overrightarrow{out_{Nat}} \ggg (\textsc{Up}_g\ u\ u' +\!\!\!+\!\!\!+ \textsc{Us}_h\ u\ u') \\
= & \quad inn_{Nat}\ ;\ out_{Nat} = id \\
 & (\textsc{Pp}_g\ p\ p' +\!\!\!+\!\!\!+ \textsc{Ps}_h\ p\ p') \ \ggg\ (\textsc{Up}_g\ u\ u' +\!\!\!+\!\!\!+ \textsc{Us}_h\ u\ u') \\
= & \quad (+\!\!\!+\!\!\!+) \text{ is a bifunctor}
\end{array}
$$

$$
\begin{array}{l}
\textbf{polytypic } \mathrm{P\scriptstyle{P}}_f :: (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow ()) \\
\quad = \lambda p\ p' \to \textbf{case } f \textbf{ of} \\
\qquad\qquad\qquad \begin{array}{lll}
g * h & \longrightarrow & \overrightarrow{\lambda((),()) \to ()} \lll (\mathrm{P\scriptstyle{P}}_g\ p\ p' \mathbin{<\!\!\ast\!\!\ast} \mathrm{P\scriptstyle{P}}_h\ p\ p') \\
\mathit{Empty} & \longrightarrow & \overrightarrow{id} \\
\mathit{Par} & \longrightarrow & p \\
\mathit{Rec} & \longrightarrow & p' \\
d\ @\ g & \longrightarrow & \mathit{ppack}_d\ (\mathrm{P\scriptstyle{P}}_g\ p\ p')
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\textbf{polytypic } \mathrm{U\scriptstyle{P}}_f :: (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (() \rightsquigarrow f\ a\ b) \\
\quad = \lambda u\ u' \to \textbf{case } f \textbf{ of} \\
\qquad\qquad\qquad \begin{array}{lll}
g * h & \longrightarrow & \overrightarrow{\lambda() \to ((),())} \ggg (\mathrm{U\scriptstyle{P}}_g\ u\ u' \mathbin{\ast\!\!\ast\!\!>} \mathrm{U\scriptstyle{P}}_h\ u\ u') \\
\mathit{Empty} & \longrightarrow & \overrightarrow{id} \\
\mathit{Par} & \longrightarrow & u \\
\mathit{Rec} & \longrightarrow & u' \\
d\ @\ g & \longrightarrow & \mathit{punpack}_d\ (\mathrm{U\scriptstyle{P}}_g\ u\ u')
\end{array}
\end{array}
$$

Figure 7.3: The definition of PP ('Pack Product') and UP ('Unpack Product').

$$
\begin{array}{ll}
 & (\mathrm{P\scriptstyle{P}}_g\ p\ p' \ggg \mathrm{U\scriptstyle{P}}_g\ u\ u') +\!\!\!+\!\!\!+ (\mathrm{P\scriptstyle{S}}_h\ p\ p' \ggg \mathrm{U\scriptstyle{S}}_h\ u\ u') \\
= & \quad \text{Equation (7.9) and the induction hypothesis} \\
 & \overrightarrow{\mathit{fmap2}_g\ i\ i'} +\!\!\!+\!\!\!+ \overrightarrow{\mathit{fmap2}_h\ i\ i'} \\
= & \quad \overrightarrow{\cdot} \text{ preserves } (+\!\!\!+\!\!\!+)\text{, definition of } \mathit{fmap2}_{g+h} \\
 & \overrightarrow{\mathit{fmap2}_{g+h}\ i\ i'}
\end{array}
$$

The base case, g**:**

$$
\begin{array}{ll}
 & \mathrm{P\scriptstyle{P}}_g\ p\ p' \ggg \overrightarrow{\lambda() \to 0} \ggg \overrightarrow{\lambda 0 \to ()} \ggg \mathrm{U\scriptstyle{P}}_g\ u\ u' \\
= & \quad (\lambda() \to 0)\ ;\ (\lambda 0 \to ()) = id_{()} \\
 & \mathrm{P\scriptstyle{P}}_g\ p\ p' \ggg \mathrm{U\scriptstyle{P}}_g\ u\ u' \\
= & \quad \text{Equation (7.9)} \\
 & \overrightarrow{\mathit{fmap2}_g\ i\ i'}
\end{array}
$$

Sequencing the parameters.

The last part of the construction of the program consists of the two functions PP and UP defined in Figure 7.3. The earlier functions have calculated and printed the constructors, so what is left is "arrow plumbing". The arrow PP $p\ p'$ traverses the top level structure of the data and inserts the correct compact printers: p at argument positions and p' at substructure positions. The only difference between UP and PP is, as with *pmapAr* and *pmapAl* earlier, the traversal direction in the product case; visible in the use of ($\lessdot\!\ast$) and ($\ast\!\gtrdot$) respectively. The inverse proof is very similar to that for TR and TL, and is omitted.

7.6 Pretty printing

Modern programming languages allow the user to define new kinds of data. When testing or debugging a program, the user often wants to see values of these new datatypes. Many languages support the automatic derivation of printing functions for user-defined datatypes. For example, in Haskell one can write **deriving** (*Show*, *Read*) after a datatype definition, and obtain the function *show* (which prints values of the datatype) and *read* (which reads them back) for free. Thus a Haskell programmer can *use* (instances of) a few predefined polytypic functions, but she has no influence over their definitions nor any means of defining her own polytypic functions.

This section shows how one can *define* polytypic versions of the functions *show* and *read*. The polytypic functions *pshow* and *pread* are each others inverses by construction.

7.6.1 More arrow classes

This subsection introduces a class *ArrowReadShow* that provides the arrow operations that are used in pretty printing and parsing. The new operations are divided into four classes: *ArrowZero*, *ArrowPlus*, *ArrowSymbol* and *ArrowPrec*. The two first classes are used for error handling and are present already in Hughes' arrow paper [42], but the last two classes are new. The operations of *ArrowSymbol* are used to print and parse symbol, and the operations of *ArrowPrec* handle operator precedences.

Arrows that can fail

Up to now the data conversion programs did not have to handle failure. The unpacking algorithm would of course benefit from error handling to allow for bad input data, but no error handling or backtracking is essential for expressing the algorithm. But to parse a text representation of data values we really need to choose between different parsers (for different constructors) and hence some parser must be able to fail. Therefore we define the class *ArrowZero* for arrows that can fail:

$$
\begin{array}{l}
\textbf{class } \mathit{Arrow}\ (\rightsquigarrow)\ \Rightarrow \mathit{ArrowZero}\ (\rightsquigarrow)\ \textbf{where} \\
\quad \mathit{zeroA}\ ::\ a \rightsquigarrow b \\
\\
\quad \texttt{--}\ \mathit{Laws}: \\
\quad \overrightarrow{f} \ggg \mathit{zeroA} \;=\; \mathit{zeroA} \;=\; \mathit{zeroA} \ggg \overrightarrow{f}
\end{array}
$$

The arrow *zeroA* is the multiplicative zero for composition with (at least) pure arrows and, as we will see later, the additive zero of a plus operator for arrows.

Error handling

The operator ($\Leftrightarrow$) in the class *ArrowPlus* builds a parser that uses a second arrow if the first one fails. The operator ($<\!|\!>$) is a kind of dual to the choice operator $(|||)\ ::\ (a \rightsquigarrow c) \to (b \rightsquigarrow c) \to (\mathit{Either}\ a\ b \rightsquigarrow c)$ from *ArrowChoice*. The choice operator makes a choice depending on the input, while the operator ($<\!|\!>$) makes a choice depending on some hidden state and delivers the result in the corresponding summand in the output.

$$
\begin{array}{l}
\textbf{class } \mathit{ArrowZero}\ (\rightsquigarrow)\ \Rightarrow \mathit{ArrowPlus}\ (\rightsquigarrow)\ \textbf{where} \\
\quad (<\!|\!>)\ ::\ (a \rightsquigarrow b) \to (a \rightsquigarrow c) \to (a \rightsquigarrow \mathit{Either}\ b\ c) \\
\quad (\Leftrightarrow)\ ::\ (a \rightsquigarrow b) \to (a \rightsquigarrow b) \to (a \rightsquigarrow b) \\
\\
\quad \texttt{--}\ \mathit{Defaults}: \\
\quad f <\!|\!> g \;=\; (f \ggg \overrightarrow{\mathit{Left}}) \Leftrightarrow (g \ggg \overrightarrow{\mathit{Right}}) \\
\quad f \Leftrightarrow g \;=\; (f <\!|\!> g) \ggg \overrightarrow{\mathit{id} \triangledown \mathit{id}} \\
\\
\quad \texttt{--}\ \mathit{Laws}: \\
\quad \mathit{zeroA} \Leftrightarrow f = f = f \Leftrightarrow \mathit{zeroA} \\
\quad f <\!|\!> \mathit{zeroA} = f \ggg \overrightarrow{\mathit{Left}} \\
\quad \mathit{zeroA} <\!|\!> f = f \ggg \overrightarrow{\mathit{Right}} \\
\quad f \ggg (g <\!|\!> h) = (f \ggg g) <\!|\!> (f \ggg h)
\end{array}
$$

The default definitions show that only one of ($<\!|\!>$) or ($\Leftrightarrow$) need be defined — the relation between the *ArrowPlus* operators is the same as that between the *ArrowChoice* operators. The arrow *zeroA* is the zero of the plus operator ($\Leftrightarrow$).

Reading and writing symbols

Almost all arrow classes thus far have been very general and useful for a wide variety of applications, but for pretty printing and parsing we need a few more specific tools. To print and parse symbols (constructor names, parentheses and spaces) we use the class *ArrowSymbol*:

$$
\begin{array}{l}
\textbf{class } \mathit{Arrow}\ (\leadsto)\ \Rightarrow \mathit{ArrowSymbol}\ (\leadsto)\ \textbf{where} \\
\quad \mathit{readSym} \quad :: \quad \mathit{Symbol} \to (a \leadsto a) \\
\quad \mathit{showSym} \quad :: \quad \mathit{Symbol} \to (a \leadsto a) \\
\\
\quad \texttt{--}\ \mathit{Laws}: \\
\quad \mathit{showSym}\ s \ggg \mathit{readSym}\ s = \overrightarrow{id} \\
\quad \mathit{showSym}\ s \ggg \mathit{readSym}\ s' = \mathit{zeroA} \quad \Leftarrow \quad s \neq s' \\
\\
\textbf{type } \mathit{Symbol} = \mathit{String}
\end{array}
$$

The two laws capture the minimal requirements needed to prove that *pshow* and *pread* are inverses: reading a symbol is the inverse of writing the same symbol but trying to read another symbol back will fail. As examples we give one arrow for printing and one for parsing parenthesized expressions:

$$
\begin{array}{lcl}
\mathit{parenthesize}, \mathit{deparenthesize} & :: & \mathit{ArrowSymbol}\ (\leadsto) \Rightarrow (a \leadsto b) \to (a \leadsto b) \\
\mathit{parenthesize}\ f & = & \mathit{showSym}\ \texttt{"("} \lll f \lll \mathit{showSym}\ \texttt{")"} \\
\mathit{deparenthesize}\ f & = & \mathit{readSym}\ \texttt{"("} \ggg f \ggg \mathit{readSym}\ \texttt{")"}
\end{array}
$$

Precedence levels

Finally, we define the class *ArrowPrec* to handle precedence levels and parentheses. Our formulation is inspired by the functions *showsPrec* and *readsPrec* in the Haskell classes *Show* and *Read*.

$$
\begin{array}{l}
\mathit{showsPrec} :: \mathit{Show}\ a\ \Rightarrow\ \mathit{Int} \to a \to \mathit{String} \to \mathit{String} \\
\mathit{readsPrec} :: \mathit{Show}\ b\ \Rightarrow\ \mathit{Int} \to \mathit{String} \to [(b, \mathit{String})]
\end{array}
$$

The integer argument passed to *showsPrec* and *readsPrec* is the precedence level of the surrounding expression. It is used to determine whether or not the element of type a should be surrounded by parentheses. As the PolyP system does not handle infix constructors, the precedence levels of Haskell can be collapsed to two levels: one for atomic expressions like unary constructors (that never need parentheses) and one for complex expressions (that need parentheses when used as subexpressions).

Function *showParen* (*readParen*) is used to enclose its printer (parser) argument with parentheses when used in a subexpression. When $p' = \mathit{showParen}\ b\ p$, the printer p' encloses p with parentheses if and only if b is *True* and p' is used as a subexpression. The printer (parser) *likeParen* p tells p to behave as a subexpression (for example by changing a precedence level hidden in the arrow type).

$$
\begin{array}{l}
\textbf{class}\ \mathit{ArrowSymbol}\ (\rightsquigarrow)\ \Rightarrow \mathit{ArrowPrec}\ (\rightsquigarrow)\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{likeParen} & :: & (a \rightsquigarrow b) \rightarrow (a \rightsquigarrow b) \\
\mathit{readParen} & :: & \mathit{Bool} \rightarrow (a \rightsquigarrow b) \rightarrow (a \rightsquigarrow b) \\
\mathit{showParen} & :: & \mathit{Bool} \rightarrow (a \rightsquigarrow b) \rightarrow (a \rightsquigarrow b)
\end{array} \\
\\
\quad \text{-- } \mathit{Laws}: \\
\quad \begin{array}{lll}
x \ggg y = \overrightarrow{z} & \Rightarrow & \mathit{likeParen}\ x \ggg \mathit{likeParen}\ y = \overrightarrow{z} \\
x \ggg y = \overrightarrow{z} & \Rightarrow & \mathit{showParen}\ b\ (\mathit{showSym}\ n \lll x) \ggg \\
 & & \mathit{readParen}\ b\ (\mathit{readSym}\ n \ggg y) = \overrightarrow{z} \\
n \neq n' & \Rightarrow & \mathit{showParen}\ b\ (\mathit{showSym}\ n \lll x) \ggg \\
 & & \mathit{readParen}\ b'\ (\mathit{readSym}\ n' \ggg y) = \mathit{zeroA}
\end{array}
\end{array}
$$

Read and show

The functions *pshow* and *pread* use operations from *ArrowChoice* and from all of the four classes just defined, and to capture this succinctly in the types, we define the class synonym *ArrowReadShow*:

$$
\begin{array}{l}
\textbf{class}\ (\mathit{ArrowChoice}\ (\rightsquigarrow), \mathit{ArrowPlus}\ (\rightsquigarrow), \mathit{ArrowPrec}\ (\rightsquigarrow)) \\
\quad \Rightarrow \mathit{ArrowReadShow}\ (\rightsquigarrow)
\end{array}
$$

For the rest of this section, all occurrences of $(\rightsquigarrow)$ will denote an arrow in the class *ArrowReadShow*.

7.6.2 Definition of *pshow* and *pread*

The definition is divided into four levels, following the structure of datatype definitions: the top level (*pshow* and *pread*) is a recursive definition, the second level (Ss and Rs) breaks down the sum structure of the functor, the third level (Sp and Rp) analyzes the product structure and finally the forth level (Sr and Rr) deals with parameters and uses of other datatypes.

The top level calculates the list of constructors of the datatype and passes them down to the next level. The second level shows (reads) the constructor name and handles parentheses (depending on the precedence of the expression and arity of the constructor). The third level inserts spaces between the arguments of the constructors and marks the arguments as being subexpressions (potentially needing parentheses). Finally the bottom level just applies the appropriate show (read) functions passed down as parameters or calls *pshow* (*pread*) for occurrences of other datatypes.

We will define a polytypic show function *pshow* and a polytypic read function *pread* and we will prove that *pread* is the inverse of *pshow*:

$$\begin{array}{rcl} pshow & :: & (a \rightsquigarrow ()) \to (d\ \ a \rightsquigarrow ()) \\ pread & :: & (() \rightsquigarrow a) \to (() \rightsquigarrow d\ \ a) \end{array}$$

$$s \ggg r = \overrightarrow{i} \;\Rightarrow\; pshow\ s \ggg pread\ r = \overrightarrow{pmap\ i} \tag{7.10}$$

Top level recursion

We use the built-in polytypic definition $constructors_d$ to access the representations of the constructors of the datatype $d\ \ a$.

$$constructors_d \;::\; [Constructor]$$

A value of the abstract type *Constructor* can be though of as a pair of the constructors name and its arity. In the following proofs we use two properties of the constructor list: the list has at least one element (there are no 0-constructor datatypes in Haskell) and all the constructor names are distinct.

Function *pshow* uses *out* to expose the top level structure of the datatype value and handles the recursion by passing itself as an argument to Ss ('Show Sum', defined later). Similarly, *pread* calls Rs ('Read Sum') and converts the result to a datatype value using *inn*.

$$\begin{array}{rcl} pshow_d\ s & = & \mathrm{Ss}_{\Phi_d}\ constructors_d\ s\ (pshow_d\ s) \lll \overrightarrow{out_d} \\ pread_d\ r & = & \mathrm{Rs}_{\Phi_d}\ constructors_d\ r\ (pread_d\ r) \ggg \overrightarrow{inn_d} \end{array}$$

The two helper functions Ss and Rs have their own inverse law:

$$\begin{array}{rcl} \mathrm{Ss}_f & :: & [Constructor] \to (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ \ a\ \ b \rightsquigarrow ()) \\ \mathrm{Rs}_f & :: & [Constructor] \to (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (() \rightsquigarrow f\ \ a\ \ b) \end{array}$$

$$s \ggg r = \overrightarrow{i} \Rightarrow s' \ggg r' = \overrightarrow{i'} \Rightarrow \mathrm{Ss}_f\ \ cs\ \ s\ \ s' \ggg \mathrm{Rs}_f\ \ cs\ \ r\ \ r' = \overrightarrow{fmap2_f\ \ i\ \ i'} \tag{7.11}$$

We assume $s \ggg r = \overrightarrow{i}$, let cs be the list of constructors and calculate as follows for Equation (7.10):

$$\begin{array}{ll} & pshow\ s \ggg pread\ r = \overrightarrow{pmap\ i} \\ \Leftarrow & \quad \text{Fixed point induction (7.3)} \\ & s' \ggg r' = \overrightarrow{i'} \;\Rightarrow \\ & \overrightarrow{out} \ggg \mathrm{Ss}_f\ \ cs\ \ s\ \ s' \ggg \mathrm{Rs}_f\ \ cs\ \ r\ \ r' \ggg \overrightarrow{inn} = \overrightarrow{out\ ;\ fmap2_f\ \ i\ \ i'\ ;\ inn} \\ \Leftarrow & \quad \text{Simplification} \end{array}$$

$$
\begin{array}{ll}
 & s' \ggg r' = \overrightarrow{i'} \;\Rightarrow\; \mathrm{Ss}_f\ cs\ s\ s' \ggg \mathrm{Rs}_f\ cs\ r\ r' = \overrightarrow{\mathit{fmap2}_f\ i\ i'} \\
\Leftarrow & \text{Law (7.11)} \\
 & \mathit{True}
\end{array}
$$

Printing constructors

On the top level, every pattern functor is a right associative sum, and this is mirrored in the definitions of Ss and Rs as well as in the corresponding part of the proof. The abstract type *Constructor* has selectors for the name and the arity of the constructor.

$$
\begin{array}{lll}
\mathit{name} & :: & \mathit{Constructor} \to \mathit{String} \\
\mathit{arity} & :: & \mathit{Constructor} \to \mathit{Int}
\end{array}
$$

We use *arity* to check for nullary constructors, which are atomic and don't need parentheses.

$$
\begin{array}{l}
\textbf{polytypic}\ \mathrm{Ss}_f :: [\mathit{Constructor}] \to (a \leadsto ()) \to (b \leadsto ()) \to (f\ a\ b \leadsto ()) \\
\quad = \lambda(c : cs)\ s\ s' \to \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad g + h \longrightarrow \mathrm{Ss}_g\ [c]\ s\ s' \mathbin{|||} \mathrm{Ss}_h\ cs\ s\ s' \\
\qquad\qquad g \qquad \longrightarrow \mathit{showParen}\ (\mathit{arity}\ c > 0) \\
\qquad\qquad\qquad\qquad (\mathit{showSym}\ (\mathit{name}\ c) \lll \mathrm{Sp}_g\ s\ s')
\end{array}
$$

$$
\begin{array}{l}
\textbf{polytypic}\ \mathrm{Rs} :: [\mathit{Constructor}] \to (() \leadsto a) \to (() \leadsto b) \to (() \leadsto f\ a\ b) \\
\quad = \lambda(c : cs)\ r\ r' \to \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad g + h \longrightarrow \mathrm{Rs}_g\ [c]\ r\ r' \mathbin{<\!\!|\!\!>} \mathrm{Rs}_h\ cs\ r\ r' \\
\qquad\qquad g \qquad \longrightarrow \mathit{readParen}\ (\mathit{arity}\ c > 0) \\
\qquad\qquad\qquad\qquad (\mathit{readSym}\ (\mathit{name}\ c) \ggg \mathrm{Rp}_g\ r\ r')
\end{array}
$$

where functions Sp (for 'Show Product') and Rp (for 'Read Product') have the following properties:

$$
\begin{array}{l}
\mathrm{Sp}_f :: (a \leadsto ()) \to (b \leadsto ()) \to (f\ a\ b \leadsto ()) \\
\mathrm{Rp}_f :: (() \leadsto a) \to (() \leadsto b) \to (() \leadsto f\ a\ b)
\end{array}
$$

$$
s \ggg r = \overrightarrow{i} \Rightarrow s' \ggg r' = \overrightarrow{i'} \Rightarrow \mathrm{Sp}_f\ s\ s' \ggg \mathrm{Rp}_f\ r\ r' = \overrightarrow{\mathit{fmap2}_f\ i\ i'} \qquad (7.12)
$$

We prove (7.11) by induction over the nested sum part of the pattern functor. We strengthen the induction hypothesis to include also the following law. For all b, x and y, and for all $c' \notin cs'$:

$$
\begin{array}{rcll}
\mathrm{Rs}_f\ cs'\ r\ r' \lll \mathit{showParen}\ b\ (\mathit{showSym}\ (\mathit{name}\ c') \lll x) & = & \mathit{zeroA} & (7.13) \\
\mathrm{Ss}_f\ cs'\ s\ s' \ggg \mathit{readParen}\ b\ (\mathit{showSym}\ (\mathit{name}\ c') \ggg y) & = & \mathit{zeroA} & (7.14)
\end{array}
$$

We assume $s \ggg r = \overrightarrow{i}$ and $s' \ggg r' = \overrightarrow{i'}$ and calculate as follows for Equation (7.11):

The sum case, $g + h$:
We prove the three equations separately, starting with (7.11):

$$
\begin{aligned}
& \mathrm{Ss}_{g+h}\ (c : cs)\ s\ s' \ggg \mathrm{Rs}_{g+h}\ (c : cs)\ r\ r' \\
= \quad & \text{Definitions} \\
& (\mathrm{Ss}_g\ [c]\ s\ s' \mathbin{|||} \mathrm{Ss}_h\ cs\ s\ s') \ggg (\mathrm{Rs}_g\ [c]\ r\ r' <\!\!|\!\!> \mathrm{Rs}_h\ cs\ r\ r') \\
= \quad & \text{Distribution laws for } (|||) \text{ and } (<\!\!|\!\!>) \\
& ((\mathrm{Ss}_g\ [c]\ s\ s' \ggg \mathrm{Rs}_g\ [c]\ r\ r') <\!\!|\!\!> (\mathrm{Ss}_g\ [c]\ s\ s' \ggg \mathrm{Rs}_h\ cs\ r\ r')) \mathbin{|||} \\
& ((\mathrm{Ss}_h\ cs\ s\ s' \ggg \mathrm{Rs}_g\ [c]\ r\ r') <\!\!|\!\!> (\mathrm{Ss}_h\ cs\ s\ s' \ggg \mathrm{Rs}_h\ cs\ r\ r'))
\end{aligned}
$$

The first term is identical to the term in the default-case below. Use induction hypothesis (7.13) and (7.14) for the second and third terms, and induction hypothesis (7.11) for the fourth term.

$$
\begin{aligned}
& (\overrightarrow{\mathit{fmap2}_g\ i\ i'} <\!\!|\!\!> \mathit{zeroA}) \mathbin{|||} (\mathit{zeroA} <\!\!|\!\!> \overrightarrow{\mathit{fmap2}_h\ i\ i'}) \\
= \quad & \text{Laws for } \mathit{zeroA} \text{ and } (<\!\!|\!\!>) \\
& (\overrightarrow{\mathit{fmap2}_g\ i\ i'} \ggg \overrightarrow{\mathit{Left}}) \mathbin{|||} (\overrightarrow{\mathit{fmap2}_h\ i\ i'} \ggg \overrightarrow{\mathit{Right}}) \\
= \quad & \text{Relation between } (|||) \text{ and } (+\!\!+\!\!+) \\
& \overrightarrow{\mathit{fmap2}_g\ i\ i'} +\!\!+\!\!+ \overrightarrow{\mathit{fmap2}_h\ i\ i'} \\
= \quad & (+\!\!+\!\!+) \text{ preserves } \overrightarrow{\cdot} \\
& \overrightarrow{\mathit{fmap2}_g\ i\ i' + \mathit{fmap2}_h\ i\ i'} \\
= \quad & \text{Definition of } \mathit{fmap2}_{g+h} \\
& \overrightarrow{\mathit{fmap2}_{g+h}\ i\ i'}
\end{aligned}
$$

Now we turn to (7.13):

$$
\begin{aligned}
& \mathit{showParen}\ b\ (\mathit{showSym}\ (\mathit{name}\ c') \lll x) \ggg \mathrm{Rs}_{g+h}\ cs'\ r\ r' \\
= \quad & \text{Definition of } \mathrm{Rs}_{g+h}, \text{ let } (c : cs) = cs' \\
& \mathit{showParen}\ b\ (\mathit{showSym}\ (\mathit{name}\ c') \lll x) \ggg \\
& (\mathrm{Rs}_g\ [c]\ r\ r' <\!\!|\!\!> \mathrm{Rs}_h\ cs\ r\ r') \\
= \quad & \text{Distribution law for } (<\!\!|\!\!>) \\
& (\mathit{showParen}\ b\ (\mathit{showSym}\ (\mathit{name}\ c') \lll x) \ggg \mathrm{Rs}_g\ [c]\ r\ r') <\!\!|\!\!> \\
& (\mathit{showParen}\ b\ (\mathit{showSym}\ (\mathit{name}\ c') \lll x) \ggg \mathrm{Rs}_h\ cs\ r\ r') \\
= \quad & \text{The second law of } \mathit{showParen} \text{ and the induction hypothesis} \\
& \mathit{zeroA} <\!\!|\!\!> \mathit{zeroA} \\
= \quad & \text{Laws for } (<\!\!|\!\!>) \text{ and } \mathit{zeroA} \\
& \mathit{zeroA}
\end{aligned}
$$

The proof of (7.14) is very similar and omitted.

The default case, g:
As the constructor list has the same number of elements as the number of subfunctors in the sum structure of the functor, there will be only one element left in the constructor list in the base case. Thus we can match on $[c]$ instead of $(c : cs)$.

$$
\begin{array}{cl}
& \mathrm{Ss}_g\ [c]\ s\ s' \ggg \mathrm{Rs}_g\ [c]\ r\ r' \\
= & \quad \text{Definitions} \\
& \mathit{showParen}\ (\mathit{arity}\ c > \mathit{0})\ (\mathit{showSym}\ (\mathit{name}\ c) \lll \mathrm{Sp}_g\ s\ s') \ggg \\
& \mathit{readParen}\ (\mathit{arity}\ c > \mathit{0})\ (\mathit{readSym}\ (\mathit{name}\ c) \ggg \mathrm{Rp}_g\ r\ r') \\
= & \begin{cases} \text{With } b = \mathit{arity}\ c > 0,\ n = \mathit{name}\ c,\ x = \mathrm{Sp}_g\ s\ s' \text{ and} \\ y = \mathrm{Rp}_g\ r\ r' \text{ we can apply the first law for } \mathit{showParen} \\ \text{as } x \ggg y = \overrightarrow{\mathit{fmap2}_g\ i\ i'} \text{ is exactly (7.12).} \end{cases} \\
& \overrightarrow{\mathit{fmap2}_g\ i\ i'}
\end{array}
$$

Both (7.13) and (7.14) follow immediately from the second law of *showParen*.

Printing constructor arguments

The function Sp (Rp) inserts (reads) a space before each argument of a constructor, and marks each argument as a subexpression (potentially needing enclosing parentheses).

$$
\begin{array}{l}
\textbf{polytypic}\ \mathrm{Sp}_f :: (a \leadsto ()) \to (b \leadsto ()) \to (f\ a\ b \leadsto ()) \\
\quad = \lambda s\ s' \to \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad g * h \quad \longrightarrow \overrightarrow{\lambda((),()) \to ()} \lll (\mathrm{Sp}_g\ s\ s' \mathbin{<\!\!*} \mathrm{Sp}_h\ s\ s') \\
\qquad\qquad \mathit{Empty} \longrightarrow \overrightarrow{\lambda() \to ()} \\
\qquad\qquad g \qquad\ \longrightarrow \mathit{showSym}\ \texttt{" "} \lll \mathit{likeParen}\ (\mathrm{Sr}_g\ s\ s')
\end{array}
$$

$$
\begin{array}{l}
\textbf{polytypic}\ \mathrm{Rp}_f :: (() \leadsto a) \to (() \leadsto b) \to (() \leadsto f\ a\ b) \\
\quad = \lambda r\ r' \to \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad g * h \quad \longrightarrow \overrightarrow{\lambda() \to ((),())} \ggg (\mathrm{Rp}_g\ r\ r' \mathbin{*\!\!>} \mathrm{Rp}_h\ r\ r') \\
\qquad\qquad \mathit{Empty} \longrightarrow \overrightarrow{\lambda() \to ()} \\
\qquad\qquad g \qquad\ \longrightarrow \mathit{readSym}\ \texttt{" "} \ggg \mathit{likeParen}\ (\mathrm{Rr}_g\ r\ r')
\end{array}
$$

where functions Sr (for ‘Show Rest’) and Rr (for ‘Read Rest’) have the following properties:

$$
\begin{array}{l}
\mathrm{Sr}_f :: (a \leadsto ()) \to (b \leadsto ()) \to (f\ a\ b \leadsto ()) \\
\mathrm{Rr}_f :: (() \leadsto a) \to (() \leadsto b) \to (() \leadsto f\ a\ b)
\end{array}
$$

$$
s \ggg r = \overrightarrow{i} \Rightarrow s' \ggg r' = \overrightarrow{i'} \Rightarrow \mathrm{Sr}_f\ s\ s' \ggg \mathrm{Rr}_f\ r\ r' = \overrightarrow{\mathit{fmap2}_f\ i\ i'} \qquad (7.15)
$$

We prove (7.12) by induction over the product structure of the functor f:

The product case, $g * h$:

$$
\begin{array}{ll}
 & \mathrm{SP}_{g*h}\ s\ s' \ggg \mathrm{RP}_{g*h}\ r\ r' \\
= & \quad \text{Definitions} \\
 & (\mathrm{SP}_g\ s\ s' \lll\!\!* \ \mathrm{SP}_h\ s\ s') \ggg \overrightarrow{\lambda((),()) \to ()} \ggg \\
 & \overrightarrow{\lambda() \to ((),())} \ggg (\mathrm{RP}_g\ r\ r' *\!\!\ggg \mathrm{RP}_h\ r\ r') \\
= & \quad (\lambda((),()) \to ())\ ;\ (\lambda() \to ((),())) = id_{((),())} \\
 & (\mathrm{SP}_g\ s\ s' \lll\!\!* \ \mathrm{SP}_h\ s\ s') \ggg (\mathrm{RP}_g\ r\ r' *\!\!\ggg \mathrm{RP}_h\ r\ r') \\
= & \quad \text{Inverse law for } (\lll\!\!*) \text{, induction hypothesis (twice)} \\
 & \overrightarrow{fmap2_g\ i\ i' * fmap2_h\ i\ i'} \\
= & \quad \text{Definition of } fmap2_{g*h} \\
 & \overrightarrow{fmap2_{g*h}\ i\ i'}
\end{array}
$$

The empty case, *Empty*:
Trivial.

The base case, g:

$$
\begin{array}{ll}
 & \mathrm{SP}_g\ s\ s' \ggg \mathrm{RP}_g\ r\ r' \\
= & \quad \text{Definitions} \\
 & likeParen\ (\mathrm{SR}_g\ s\ s') \ggg showSym\ \texttt{" "} \ggg \\
 & readSym\ \texttt{" "} \ggg likeParen\ (\mathrm{RR}_g\ r\ r') \\
= & \quad \text{Law for } showSym \text{ and } readSym \\
 & likeParen\ (\mathrm{SR}_g\ s\ s') \ggg likeParen\ (\mathrm{RR}_g\ r\ r') \\
= & \quad \text{Law for } likeParen \text{ and Equation (7.15)} \\
 & \overrightarrow{fmap2_g\ i\ i'}
\end{array}
$$

Printing the rest

At the bottom level all that is left is to apply the correct printer (parser): *Par* and *Rec* select from the parameters and $d@g$ calls the top level *pshow* (*pread*) recursively.

$$
\begin{array}{l}
\textbf{polytypic}\ \mathrm{SR}_f :: (a \leadsto ()) \to (b \leadsto ()) \to (f\ a\ b \leadsto ()) \\
\quad = \lambda s\ s' \to \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad\qquad Par \quad \longrightarrow s \\
\qquad\qquad\qquad Rec \quad \longrightarrow s' \\
\qquad\qquad\qquad d\ @\ g \longrightarrow pshow_d\ (\mathrm{SR}_g\ s\ s')
\end{array}
$$

$$
\begin{array}{l}
\textbf{polytypic}\ \mathrm{R}\textsc{r} :: (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (() \rightsquigarrow f\ a\ b) \\
\quad = \lambda r\ r' \to \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad\qquad \begin{array}{lll} Par & \longrightarrow & r \\ Rec & \longrightarrow & r' \\ d\ @\ g & \longrightarrow & pread_d\ (\mathrm{R}\textsc{r}_g\ r\ r') \end{array}
\end{array}
$$

The only remaining proof obligation is (7.15) and the proof is once again by induction on the structure of the functor — the *Par* and *Rec* cases follow immediately from the assumptions, and the $d\ @\ g$ case from the top level induction hypothesis (7.10) and the local induction hypothesis (7.15). This completes the proof that *pread* is the inverse of *pshow*.

7.7 Generating arrow instances

Most of the code presented in this chapter is generic in two ways. We use polytypism to parametrize our definitions by a regular datatype, and we use Haskell's constructor classes to parametrize by the choice of concrete arrow implementation. Using PolyP, we obtain specific instances of the polytypic functions automatically, but we do have to write instances for the arrow classes. This section describes a few general arrow constructors and shows how to combine them to obtain an example instance for *ArrowReadShow* that satisfies the necessary laws.

We have already presented three arrow instances: the trivial function arrow $a \to b$, the Kleisli arrows $Kleisli\ m\ a\ b$ for every monad m and the state arrow $a \rightsquigarrow_s b$ for any state type s. The state arrow can be generalized to a state arrow *transformer* that adds state passing to any other arrow:

$$\textbf{newtype}\ StateArrT\ s\ (\rightsquigarrow)\ a\ b = SAT\ ((a, s) \rightsquigarrow (b, s))$$

With this definition the simple state arrow $SA\ s$ is equivalent to adding state passing to the trivial arrow: $StateArrT\ s\ (\to)$. The state arrow transformer instances for *Arrow*, *ArrowChoice*, *ArrowZero* and *ArrowPlus* are in Figure 7.4.

The Kleisli arrows were defined in Section 7.3 together with instances for *Arrow* and *ArrowChoice*. If the underlying monad has a zero and a plus operation (is an instance of the Haskell class *MonadPlus*), then we can define instances for *ArrowZero* and *ArrowPlus* as well:

$$\textbf{newtype}\ Kleisli\ m\ a\ b = Kleisli\ (a \to m\ b)$$

$$
\begin{array}{l}
\textbf{instance}\ MonadPlus\ m \Rightarrow ArrowZero\ (Kleisli\ m)\ \textbf{where} \\
\quad zeroA\ =\ Kleisli\ (const\ mzero)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ MonadPlus\ m \Rightarrow ArrowPlus\ (Kleisli\ m)\ \textbf{where} \\
\quad Kleisli\ f \mathbin{<\!+\!>} Kleisli\ g\ =\ Kleisli\ (\lambda x \to mplus\ (f\ x)\ (g\ x))
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Arrow}\ (\leadsto) \Rightarrow \mathit{Arrow}\ (\mathit{StateArrT}\ s\ (\leadsto))\ \textbf{where} \\
\quad \begin{array}{lcl}
\overrightarrow{f} & = & \mathit{SAT}\ (\overrightarrow{\mathit{first}\ f}) \\
\mathit{SAT}\ f \ggg \mathit{SAT}\ g & = & \mathit{SAT}\ (f \ggg g) \\
\mathit{first}\ (\mathit{SAT}\ f) & = & \mathit{SAT}\ (\overrightarrow{\mathit{swap23}} \ggg \mathit{first}\ f \ggg \overrightarrow{\mathit{swap23}})
\end{array}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{swap23} & :: & ((a,b),s) \rightarrow ((a,s),b) \\
\mathit{swap23} & = & \lambda((a,b),s) \rightarrow ((a,s),b)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{ArrowChoice}\ (\leadsto) \Rightarrow \mathit{ArrowChoice}\ (\mathit{StateArrT}\ s\ (\leadsto))\ \textbf{where} \\
\quad \mathit{SAT}\ f\ |||\ \mathit{SAT}\ g \ = \ \mathit{SAT}\ (\overrightarrow{\mathit{eitherout}} \ggg (f\ |||\ g))
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{eitherout} & :: & (\mathit{Either}\ a\ a', s) \rightarrow \mathit{Either}\ (a,s)\ (a',s) \\
\mathit{eitherout}\ (x,s) & = & (\mathit{pairs} + \mathit{pairs})\ x\ \textbf{where}\ \mathit{pairs}\ a = (a,s)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{ArrowZero}\ (\leadsto) \Rightarrow \mathit{ArrowZero}\ (\mathit{StateArrT}\ s\ (\leadsto))\ \textbf{where} \\
\quad \mathit{zeroA} \ = \ \mathit{SAT}\ \mathit{zeroA}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{ArrowPlus}\ (\leadsto) \Rightarrow \mathit{ArrowPlus}\ (\mathit{StateArrT}\ s\ (\leadsto))\ \textbf{where} \\
\quad \mathit{SAT}\ f \mathbin{<\!\!+\!\!>} \mathit{SAT}\ g \ = \ \mathit{SAT}\ (f \mathbin{<\!\!+\!\!>} g)
\end{array}
$$

Figure 7.4: Instance declarations for the state arrow transformer.

All the arrow constructors defined so far were defined also in Hughes' arrow paper [42], but the following construction is new. The monad arrow constructor *MonadArrT* wraps a monad around the arrow type:[4]

$$\textbf{newtype}\ \mathit{MonadArrT}\ m\ (\leadsto)\ a\ b = \mathit{MAT}\ (m\ (a \leadsto b))$$

For every monad we can lift an arrow to an arrow, but to support choice, failure and error handling we need to restrict the monad to, essentially, a reader monad. The reader arrow transformer *ReaderArrT* is a special case of the monad arrow transformer:

$$\textbf{type}\ \mathit{ReaderArrT}\ r = \mathit{MonadArrT}\ (r \rightarrow)$$

The transformer *ReaderArrT* r adds an environment of type r to any arrow. This can also be simulated with *StateArrT* but when no update is needed, *ReaderArrT* is more efficient and also simplifies the proofs. We use the shorthand notation $a \overset{r}{\leadsto} b$ for *ReaderArrT* $r\ (\leadsto)\ a\ b$. (Note the difference between the notation $a \leadsto_s b$ for the state arrow and $a \overset{r}{\leadsto} b$ for the reader arrow.) The instances for *MonadArrT* and *ReaderArrT* are in Figure 7.7.

[4]The monad arrow constructor is a special case of an even more general static arrow constructor (Paterson [88]) that wraps any cartesian functor around the arrow type.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Monad}\ m, \mathit{Arrow}\ (\leadsto)) \Rightarrow \mathit{Arrow}\ (\mathit{MonadArrT}\ m\ (\leadsto))\ \textbf{where} \\
\quad \begin{array}{lcl}
\overrightarrow{f} & = & \mathit{MAT}\ (\mathit{liftM0}\ \overrightarrow{f}) \\
\mathit{MAT}\ f \ggg \mathit{MAT}\ g & = & \mathit{MAT}\ (\mathit{liftM2}\ (\ggg)\ f\ g) \\
\mathit{first}\ (\mathit{MAT}\ f) & = & \mathit{MAT}\ (\mathit{liftM}\ \mathit{first}\ f)
\end{array} \\[1ex]
\textbf{instance}\ \mathit{ArrowChoice}\ (\leadsto) \Rightarrow \mathit{ArrowChoice}\ (\mathit{ReaderArrT}\ r\ (\leadsto))\ \textbf{where} \\
\quad \mathit{MAT}\ f\ |||\ \mathit{MAT}\ g \ = \ \mathit{MAT}\ (\mathit{liftM2}\ (|||)\ f\ g) \\[1ex]
\textbf{instance}\ \mathit{ArrowZero}\ (\leadsto) \Rightarrow \mathit{ArrowZero}\ (\mathit{ReaderArrT}\ r\ (\leadsto))\ \textbf{where} \\
\quad \mathit{zeroA} \ = \ \mathit{MAT}\ (\mathit{liftM0}\ \mathit{zeroA}) \\[1ex]
\textbf{instance}\ \mathit{ArrowPlus}\ (\leadsto) \Rightarrow \mathit{ArrowPlus}\ (\mathit{ReaderArrT}\ r\ (\leadsto))\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{MAT}\ f \lhd\!\rhd \mathit{MAT}\ g & = & \mathit{MAT}\ (\mathit{liftM2}\ (\lhd\!\rhd)\ f\ g) \\
\mathit{MAT}\ f \Leftrightarrow \mathit{MAT}\ g & = & \mathit{MAT}\ (\mathit{liftM2}\ (\Leftrightarrow)\ f\ g)
\end{array} \\[1ex]
\begin{array}{lcl}
\mathit{liftM0} & :: & \mathit{Monad}\ m \Rightarrow a \to m\ a \\
\mathit{liftM0}\ f & = & \mathit{return}\ f \\
\mathit{liftM} & :: & \mathit{Monad}\ m \Rightarrow (a \to b) \to (m\ a \to m\ b) \\
\mathit{liftM}\ f\ m & = & m \ggg\!= \lambda x \to \mathit{return}\ (f\ x) \\
\mathit{liftM2} & :: & \mathit{Monad}\ m \Rightarrow (a \to b \to c) \to (m\ a \to m\ b \to m\ c) \\
\mathit{liftM2}\ f\ m\ n & = & m \ggg\!= \lambda x \to n \ggg\!= \lambda y \to \mathit{return}\ (f\ x\ y)
\end{array}
\end{array}
$$

Figure 7.5: Instance declarations for *MonadArrT* and *ReaderArrT*.

Two useful operations on *ReaderArrT* arrows are *getEnv* and *setEnv*:

$$\begin{array}{l}
getEnv \ :: \ Arrow\ (\leadsto) \Rightarrow a \overset{r}{\leadsto} r \\
getEnv = MAT\ (\lambda x \to \overrightarrow{const\ x}) \\
setEnv :: r \to (a \overset{r}{\leadsto} b) \to (a \overset{s}{\leadsto} b) \\
setEnv\ x\ (MAT\ f) = MAT\ (const\ (f\ x))
\end{array}$$

The arrow *getEnv* ignores its input and returns the value of the environment. The arrow *setEnv* x f transforms f by shielding it from the outside environment so that all *getEnv*s from f give x.

7.7.1 An instance for *ArrowReadShow*

We can combine the three general arrow constructors to obtain an arrow *RS* that can be made an instance of *ArrowReadShow*:

$$\begin{array}{l}
\textbf{type}\ RS = ReaderArrT\ Int\ (StateArrT\ Tokens\ (Kleisli\ ([\,]))) \\
\textbf{type}\ Tokens = [String]
\end{array}$$

The transformer *ReaderArrT Int* adds an environment containing an integer to handle the precedence level, the transformer *StateArrT Tokens* adds a state containing a token list and the inner arrow *Kleisli* ([ ]) handles the list of alternative parses. By unfolding the definitions of the arrow constructors we get

$$RS\ a\ b \cong Int \to (a, Tokens) \to [(b, Tokens)]\ .$$

This can be compared with the types for *showsPrec* and *readsPrec* from the Haskell prelude.

$$\begin{array}{lll}
showsPrec :: Show\ a\ \Rightarrow\ Int \to a \to String & \to & String \\
readsPrec :: Show\ b\ \Rightarrow\ Int \to String & \to & [(b, String)]
\end{array}$$

These types use *String* where *RS* uses *Tokens*, but are otherwise very similar to $RS\ a\ ()$ and $RS\ ()\ b$, respectively.

The arrow *RS* is by construction an instance of *Arrow*, *ArrowChoice*, *ArrowZero* and *ArrowPlus*. Hence to make *RS* an instance of *ArrowReadShow* all we need is instances for *ArrowSymbol* and *ArrowPrec*. Function *readSym* is the standard item parser and *showSym* is even simpler (both ignore the precedence). The functions *likeParen*, *showParen* and *readParen* use the precedence level environment to determine when to read or write parentheses (using *readSym* and *showSym*). Example instantiations of *ArrowSymbol* and *ArrowPrec* for the arrow *RS* are in Figure 7.6, where *high* is the precedence level of expressions that need parentheses. The proofs that the instances satisfy the laws of the classes are long but relatively simple.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{ArrowSymbol}\ \mathit{RS}\ \textbf{where} \\
\quad \mathit{showSym}\ s \;=\; \mathit{MAT}\ (\mathit{return}\ (\mathit{SAT}\ (\mathit{second}\ \overrightarrow{(s:)}))) \\
\quad \mathit{readSym}\ s \;=\; \mathit{MAT}\ (\mathit{return}\ (\mathit{SAT}\ (\mathit{second}\ (\mathit{Kleisli}\ (\mathit{readToken}\ s)))))
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{readToken}\ t\ (s : ss) \mid s == t & = & \mathit{return}\ ss \\
\mathit{readToken}\ t\ \_ & = & \mathit{mzero}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{ArrowPrec}\ \mathit{RS}\ \textbf{where} \\
\quad \mathit{likeParen} \;=\; \mathit{setEnv}\ \mathit{high} \\
\quad \mathit{showParen}\ b\ f \;=\; \mathit{ifHighPrec}\ b\ (\mathit{parenthesize}\ f)\ f \\
\quad \mathit{readParen}\ b\ f \;=\; \mathit{ifHighPrec}\ b\ (\mathit{deparenthesize}\ f)\ f
\end{array}
$$

$$
\begin{array}{l}
\mathit{ifHighPrec} :: \mathit{ArrowChoice}\ (\leadsto) \Rightarrow \mathit{Bool} \rightarrow (a \overset{\mathit{Int}}{\leadsto} b) \rightarrow (a \overset{\mathit{Int}}{\leadsto} b) \rightarrow (a \overset{\mathit{Int}}{\leadsto} b) \\
\mathit{ifHighPrec}\ b = \mathit{ifA}\ (\mathit{getEnv} \ggg \overrightarrow{\lambda p \rightarrow b \wedge p == \mathit{high}})
\end{array}
$$

Figure 7.6: Instances for *ArrowSymbol* and *ArroParen*.

7.8 Results and conclusions

Overview of the results

We have defined the following pairs of data conversion programs and related them with inverse laws:

- Shape plus content: (Section 7.2)

$$
\begin{array}{lcl}
\mathit{separate} & :: & d\ a \leadsto_{[a]} d\ () \\
\mathit{combine} & :: & d\ () \leadsto_{[a]} d\ a \\
\multicolumn{3}{l}{\mathit{separate} \ggg \mathit{combine} = \overrightarrow{\mathit{id}}}
\end{array}
$$

- Arrow maps: (Section 7.4)

$$
\begin{array}{lcl}
\mathit{pmapAr} & :: & \mathit{ArrowChoice}\ (\leadsto)\ \Rightarrow\ (a \leadsto b) \rightarrow (d\ a \leadsto d\ b) \\
\mathit{pmapAl} & :: & \mathit{ArrowChoice}\ (\leadsto)\ \Rightarrow\ (a \leadsto b) \rightarrow (d\ a \leadsto d\ b) \\
\multicolumn{3}{l}{f \ggg g = \overrightarrow{i}\ \Rightarrow\ \mathit{pmapAr}\ f \ggg \mathit{pmapAl}\ g = \overrightarrow{\mathit{pmap}\ i}}
\end{array}
$$

- Packing: (Section 7.5)

$$
\begin{array}{lcl}
\mathit{ppack} & :: & \mathit{ArrowPack}\ (\leadsto)\ \Rightarrow\ (a \leadsto ()) \rightarrow (d\ a \leadsto ()) \\
\mathit{punpack} & :: & \mathit{ArrowPack}\ (\leadsto)\ \Rightarrow\ (() \leadsto a) \rightarrow (() \leadsto d\ a) \\
\multicolumn{3}{l}{p \ggg u = \overrightarrow{i}\ \Rightarrow\ \mathit{ppack}\ p \ggg \mathit{punpack}\ u = \overrightarrow{\mathit{pmap}\ i}}
\end{array}
$$

- Pretty printing: (Section 7.6)

$$
\begin{array}{lcl}
pshow & :: & ArrowReadShow\ (\rightsquigarrow)\ \Rightarrow\ (a \rightsquigarrow ()) \rightarrow (d\ a \rightsquigarrow ()) \\
pread & :: & ArrowReadShow\ (\rightsquigarrow)\ \Rightarrow\ (() \rightsquigarrow a) \rightarrow (() \rightsquigarrow d\ a) \\
\multicolumn{3}{l}{s \ggg r = \overrightarrow{i}\ \Rightarrow\ pshow\ s \ggg pread\ r = \overrightarrow{pmap\ i}}
\end{array}
$$

We can combine the last two applications to obtain compression and decompression. The composition of the polytypic read function *pread* with the packing function *ppack* gives a structured compression algorithm *pcompress* that takes a plain text representation of a datatype value to a bit stream. The corresponding decompression algorithm *pdecompress* is a composition of the unpacking function *punpack* and the polytypic show function *pshow*. Function *pdecompress* is the inverse of *pcompress* for all strings that represent a value. This fact follows from the inverse laws for pretty printing and packing.

Conclusions

We have constructed polytypic programs for several data conversion problems. As far as we are aware, these are the first implemented generic descriptions of programs for data conversion problems. Recent work by Hinze [33] also contains a polytypic show function and a simple packing function, but his language still lacks an implementation.

For each of the data conversion problems considered in this chapter we construct a pair of functions. These pairs of functions are inverse functions by construction. Since we started applying the inverse function requirement rigorously in the construction of the programs, the size and the complexity of the code have been reduced considerably. Compare for example Björk's [13] and Huisman's [43] definitions, with the polytypic read and show functions defined in this chapter. We firmly believe that such a rigorous approach is the only way to obtain elegant solutions to involved polytypic problems. Another concept that simplified the construction and form of the program is arrows. In our first attempts to polytypic programs for packing and unpacking we used monads instead of arrows. Although it is possible to construct the (un)packing functions with monads (see Halenbeek [31]), the inverse function construction, and hence the correctness proof, is simpler with arrows.

We have shown how to construct programs for several data conversion problems. We expect that our programs and proofs will be very useful in the construction of programs for other data conversion problems.

Although all our data conversion programs are linear, both time and space efficiency of our programs leave much to be desired. We expect that sufficiently sophisticated forms of partial evaluation will improve the performance of our programs considerably. We want to experiment with partial evaluation of polytypic functions in the future.

We have presented a few calculations of polytypic programs. We think that calculating with polytypic functions is still rather cumbersome, and we hope to obtain more theory, in the style of Meertens [76] and Hinze [36], to simplify calculations with polytypic programs. If we take Hinze's approach to polytypic programming [35], then we only have 4 constructors for pattern functors instead of 7, and this should reduce the length of the proofs. In collaboration with Hinze, we are currently working on an implementation of his approach as a successor to PolyP.

Chapter 8

Related work

In this chapter we describe work related to functional polytypic programming. We briefly describe a number of subject areas which have influenced the development of polytypism and give many references to further reading.

8.1 BMF $\cong$ Squiggol

Polytypism has its roots in the branch of constructive algorithmics that was named the Bird-Meertens Formalism (BMF) [10, 74] by Backhouse [3]. BMF is not really a well defined formalism, but rather a collection of definitions, transformations and laws for calculating with programs. In the "Theory of Lists" [10, 11, 60] many laws for calculating with programs are proved and used to derive efficient algorithms from clearly correct (but often hopelessly inefficient) specifications. Polytypic functions are widely used in the Squiggol community [4,7,24,72,75–77,79], where the list based calculus is generalized and extended to datatypes that can be defined by means of a regular functor. Polytypic versions of many list functions are defined: *cata*, *map*, *zip*, *sum* etc. and together with the functions, also the theorems and the transformation techniques developed in the theory of lists are generalized. Backhouse *et al.* [4] argues convincingly that the basis of the theory of polytypism is best described in a relational setting. Bird, de Moor and Hoogendijk [7] use this setting to generalize the theory of segments of lists to all datatypes. The connection between polytypic programming and dependent types (in the context of Martin-Löf type theory [83]) has been investigated by Backhouse [5], Pfeifer and Rueß [91] and Dybjer [18, 19].

8.2 Theories of datatypes

A polytypic value is a family of values indexed by (the structure of) datatypes. Thus the choice of formalism to represent datatypes is of central importance for polytypic programming. The Squiggol community takes the categorical view of modeling datatypes as initial functor-algebras. This is a relatively old idea, on which a large amount of literature exists, see, amongst others, Lehmann and Smyth [69], Manes and Arbib [73], and Hagino [30]. Böhm and Berarducci [14] have a more algebraic approach to modeling datatypes. They define a *data system* (a group of mutually recursive datatypes) to be a finite parametric heterogeneous term algebra. This is one of the few references where mutually recursive datatypes with multiple parameters are described in detail. Hoogendijk, de Moor and Backhouse [37–39] argue that a datatype (or, more specifically, a container type) is a relator with a membership relation.

8.3 Beyond regular datatypes

Polytypic functions are traditionally defined for regular datatypes. Regular datatypes are initial fixed points of regular functors or, in the relational setting, regular relators [4].

Jay [56, 57] has developed an alternative theory for polytypic functions, in which values are represented by their structure and their contents. He uses a category theoretic formulation of polytypism based on the notion of strong functor [80].

The class of datatypes on which polytypic functions can be defined can be extended (with some effort) to include datatypes with function spaces. Freyd [26] provides the category theoretic background for this extension. The problem with the extension is that if a datatype parameter occurs in a negative position (to the left of an odd number of function arrows) in a datatype definition, then the recursive definition of the catamorphism uses its own (right) inverse. Meijer and Hutton [78] apply Freyd's theory to the definition of catamorphisms for datatypes with embedded functions. They solve the problem of negative parameters by simultaneously defining both the catamorphism and its right inverse (an anamorphism). Fegaras and Sheard [21] point out that this solution is too restrictive: there are functions that can be defined as catamorphisms even though they lack a right inverse. They give an alternative definition of the catamorphism using an approximate inverse and give a type system that rejects the cases when this approximation would not be safe.

Recent results extend polytypic programming to work on non-regular, so called *nested* datatypes [8]. Bird and Paterson [9] suggest generalized folds and Hinze [34] shows how one can define most other polytypic functions so that they work also on nested datatypes.

Fokkinga extends the theory of datatypes to include "Datatype Laws without Signatures" [22] enabling abstract datatypes like stacks to be defined in a category theoretic setting. To extend polytypic programming in this direction would be an interesting subject for future work.

8.4 Specific polytypic functions

Generating instances for specific polytypic functions, such as $(==)$, *map*, *cata* etc. for a given type, is rather simple and has been demonstrated by several authors [14, 45, 46, 55, 75, 82, 96, 98]. Catamorphisms were generated by Böhm and Berarducci [14] (in the Λ-calculus) and Sheard [98] (in an ML-like language). Sheard also gave programs to automatically generate other kinds of traversal functions like accumulations and equality functions.

The paramorphism, a more general recursion operator than the catamorphism, was introduced by Meertens [75]. (The recursion pattern captured by the paramorphism in essentially the same as the pattern in the elimination rule for a datatype in constructive type theory.) Many other recursion operators are defined by de Moor and Fokkinga [24,82]. A catamorphism can also be generalized to a *monadic catamorphism* [25,79] that threads a monad through the structure. The use of anamorphisms is advocated in "The Under-Appreciated Unfold" by Gibbons and Jones [29] and monadic anamorphisms are defined by Pardo [87].

Polytypic functions for specific programming problems, such as the maximum segment sum problem and the pattern matching problem were first given by Bird *et al.* [7] and Jeuring [61]. (The first published use of the term *polytypic function* was by Jeuring [61].) Many other algorithms have also been expressed polytypically: unification [51], pattern matching [61], data compression [46, 53], parsing and pretty printing [53], rewriting [52,54,62], genetic programming [100], downwards accumulations [28], etc.

All these polytypic functions are parametrized on *one* datatype. There is, however, no theoretical problem with defining multiply parametrized polytypic functions. One example is the doubly parametric function *transpose* (also called *zip*) defined by Ruehr [94] and Hoogendijk and Backhouse [38]. In PolyP it could have the type:

$$transpose :: (Regular\ d, Regular\ e) \Rightarrow d\ (e\ a) \to e\ (d\ a)$$

It is a generalization of the transpose operation on matrices.

If the inner datatype e in the type for *transpose* is replaced by a monad m, then a polytypic (and monadic) *traversal* [47, 79, 81] function is obtained. Traversals can be used for a wide range of problems — examples are the data conversion programs in Chapter 7. Traversals are also widely used in "imperative polytypic programming" — the topic of the Section 8.8. As an example, the *Visitor* design pattern [27] is a traversal.

8.5 Type systems

Type systems for languages which allow the use of polytypic functions have been developed by several people:

- Ruehr [94] gives a full higher-order type pattern language. The higher-order aspects of the type system makes the language a bit impractical but he also presents a trade-off design for a more manageable language with type inference.
- Jones' type system [64,66] is based on qualified types and higher-order polymorphism. The type system is implemented in the Haskell system Hugs. Haskell has no construction for writing polytypic functions by induction on user defined datatypes but can be used to simulate and type check polytypic functions using constructor classes.
- Sheard and Nelson [97] gives a type system for a restricted version of Compile time Reflexive ML. CRML is a two-level language and a polytypic program is obtained by embedding second level type declarations as values in first level computations. The restriction is that recursion in the first level (that is executed at compile time) must be expressed using catamorphisms only, to guarantee termination. The type system uses dependent types and a special type construction for the types of catamorphisms.
- Harper and Morrisett [32] present a type system for a language with "intensional polymorphism". A dependently typed *typecase* construct for explicit matching on predefined types is used to define generic functions that work for different types. The *typecase* cannot, however, be used to match on the structure of user defined datatypes.
- Our type system (described in Chapter 4 and in Jansson and Jeuring [46]) extends Jones' system [64, 66] with the possibility to introduce and type check polytypic functions defined by induction on the structure of user defined datatypes.
- Jay *et al.* [55,58] describe a type system for "Functorial ML", an intermediate language with some predefined polytypic functions including *map* and *cata* (they call it *fold*). The language can deal with multiple parameter datatypes, but not mutual recursive datatypes.

- Hinze [35, 36] presents another approach to polytypic programming with a type system for *type indexed values*. In this setting polytypic functions can be defined not only for regular datatypes, but for a much wider class, including nested datatypes, mutually recursive datatypes and higher-order datatypes.

8.6 Implementations

In this dissertation we have argued that a polytypic programming system should

- type check polytypic code,
- allow definitions of new polytypic functions, and
- generate instances of these polytypic functions for regular datatypes.

In the language *Charity* [16] polytypic functions like the catamorphism and map are automatically provided for each user-defined datatype. But it is not possible to define new polytypic functions in *Charity*. The functional language **P2** [55] does not satisfy the second requirement, but a few generations later, Jay's ongoing work with the language *FISh* [59] supports all three requirements. Hinze [35] presents a promising approach to polytypic programming and proposes to add this as a generic programming extension for Haskell [33], but it is not yet implemented.

Our system PolyP, described in Chapter 4 satisfies these requirements and we know of only one other such system: that of Sheard [96] using a restricted compile time reflective setting. The reason we are not using Sheard's system is that it uses a two level language built on ML (Compile-time Reflexive ML [40]) extended with a type system using some dependent types. We did not want to move that far away from the Haskell (type) system. Using the explicit type parameters of Cayenne [1] (a language combining the programming power of Haskell with the specification power of Martin-Löf's type theory [83]) we could perhaps obtain a polytypic extension with less overhead than the current Haskell based system.

We are planning for a successor of PolyP: Generic Haskell [33] in which we combine the experiences from the PolyP system with Hinze's new ideas (extensions to handle multiple type arguments, mutually recursive datatypes) .

8.7 Polytypic transformations and proofs

Malcolm [72] and Fokkinga [22–24] develop categorical techniques for calculating and transforming programs. The most well know polytypic transformation is the fusion law, first described by Malcolm [71, 72]. (Malcolm calls it the promotion theorem following Bird's terminology for lists [10].) The fusion law (for applications, see Chapter 3) gives the conditions under which the composition of a function with a catamorphism can be fused to a single catamorphism. Takano and Meijer [99] use another polytypic law, the acid-rain theorem, to apply deforestation [103] transformations and Hu [41] uses a number of polytypic laws to eliminate multiple traversals of data by combining functions that recurse over the same structure. The calculational fusion system HYLO [85] can be used to calculated with programs expressed in terms of hylomorphisms (a generalized combination of catamorphisms, maps and anamorphisms).

Both the fusion law and the acid-rain theorem are examples of *free theorems* [101]. A free theorem can be derived automatically from the polymorphic type of a function. For example, the fusion law is the free theorem of function *cata*. Fegaras and Sheard [21, Appendix A.1] (in more detail: [20]) give a function that given a type constructs its free theorem.

More examples of polytypic calculation of programs can be found in Jeuring [61], Meertens [76], in the textbook 'Algebra of Programming' by Bird and de Moor [12] and, of course, in this dissertation.

Hutton and Gibbons present the *generic approximation lemma* [44] — a beautiful result that can be used to prove generic properties of functions that work on possibly infinite data. Hinze [36] presents a powerful proof rule for his kind of polytypic functions and uses this rule to prove that the polytypic *map* function is a functor.

8.8 Imperative Polytypic Programming

In the imperative world polytypic programming appears under the broader concept "design patterns" [27], and more specifically as "adaptive object-oriented programming" [70, 86]. Adaptive OOP addresses some of the same issues as polytypic programming and, although the associated programming style is very different from functional polytypic programming, the resulting programs have very similar behavior. In adaptive OOP a method (corresponding to a polytypic functions) is attached to a group of classes (corresponding to a datatype) that usually satisfies certain constraints (such as being regular).

Lieberherr *et al.* [70] describes a system that allows the programmer to write template programs containing a number of methods with associated 'propagation patterns'. The template programs are parametrized on (the structure of) a group of related classes and the system automatically instantiates these templates for different class dependency graphs. Each method in the template program has a signature (the type of the method), a pattern (that specifies the set of paths in the class dependency graph on which the method should be used) and a code part (to be executed for all matching paths).

Advanced uses of the C++ Standard Template Library (STL) [92] can also be considered polytypic programming, but as C++ lacks recursive types the style is very different from the style of functional polytypic programming in this dissertation. Much of the work with STL is programming against a "generic" interface (for example iterators) to obtain reusable code. The closest match in this dissertation is the *Term* interface used in Chapter 6, but many things that STL adds to C++ (for example, parametric polymorphism) are already present in Haskell without the polytypic extensions.

Acknowledgments

Special thanks to my supervisor Johan Jeuring for introducing me to the field of polytypic programming and for countless discussions during the last few years. I thank my opponent Mark Jones for his thorough work on reading and commenting on previous versions of this dissertation. I also thank the rest of my PhD committee, Thierry Coquand and Lennart Augustsson, and my professor John Hughes for their support during these years. I thank the Multi group here at Chalmers for providing an inspiring research environment, and finally I thank my wife Tünde Fülöp for her constant support.

Chapter 4: The discussions on type systems for polytypic programming with Alex Aiken, Graham Hutton, Mark Jones, Oege de Moor, Rinus Plasmeijer, Fritz Ruehr, Tim Sheard and the comments from an anonymous referee are gratefully acknowledged. Oege de Moor, Graham Hutton and Arjan van IJzendoorn helped implementing predecessors of PolyP.

Chapter 7: This chapter has been improved by constructive comments by Ralf Hinze on polytypism and general presentation, by Ross Paterson on arrows and their laws and by Roland Backhouse on the fixed point calculations. Joost Halenbeek implemented a polytypic data compression program using monads. The anonymous referees suggested many improvements regarding contents and presentation.

Bibliography

[1] L. Augustsson. Cayenne — a language with dependent types. *ACM SIGPLAN Notices*, 34(1):239–250, 1998. Presented at ICFP'98.

[2] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1999.

[3] R.C. Backhouse. An exploration of the Bird–Meertens formalism. Technical Report Computing Science Notes CS 8810, Department of Mathematics and Computing Science, University of Groningen, 1988.

[4] R.C. Backhouse, P.J. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J.C.S.P. van der Woude. Relational catamorphisms. In B. Möller, editor, *Constructing Programs from Specifications*, pages 287–318. North-Holland, 1991.

[5] Roland Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. In A. Avron, B. Harper, F. Honsell, I. Mason, and G. Plotkin, editors, *Proceedings of the Workshop on General Logic, Edinburgh, February 1987*. Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1988. ECS-LFCS-88-52.

[6] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice Hall, 1990.

[7] Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

[8] Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Mathematics of Program Construction*, volume 1422 of *LNCS*, pages 52–67. Springer-Verlag, 1998.

[9] Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, September 1999.

[10] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer-Verlag, 1987.

[11] R.S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume F55 of *NATO ASI Series*, pages 151–216. Springer-Verlag, 1989.

[12] R.S. Bird and O. de Moor. *Algebra of Programming.* Prentice-Hall International, 1997.

[13] Staffan Björk. Parsers, pretty printers and PolyP. Master's thesis, University of Göteborg, 1997. Examensarbeten 1997:31. Available from the Polytypic programming www page.

[14] C. Böhm and A. Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[15] Robert D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, 1988.

[16] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Dep. of Computer Science, Univ. of Calgary, 1992.

[17] L. Damas and R. Milner. Principal type-schemes for functional programs. In *9th Symposium on Principles of Programming Languages, POPL '82*, pages 207–212, 1982.

[18] P. Dybjer. Inductive families. *Formal Aspects of Computing*, pages 440–465, 1994.

[19] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, volume 1581 of *LNCS*, pages 129–146. Springer-Verlag, April 1999.

[20] Leonidas Fegaras. Fusion for free! Technical Report CSE-96-001, Department of Computer Science, Oregon Graduate Institute, 1996. Available by `ftp` from `cse.ogi.edu` in `/pub/tech-reports/1996/96-001.ps.gz`.

[21] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions. In *Proceedings Principles of Programming Languages, POPL '96*, 1996.

[22] Maarten M. Fokkinga. Datatype laws without signatures. *Mathematical Structures in Computer Science*, 6:1–32, 1996.

[23] M.M. Fokkinga. Calculate categorically! *Formal Aspects of Computing*, 4(4):673–692, 1992.

[24] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.

[25] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, June 1994.

[26] P. Freyd. Recursive types reduced to inductive types. In *Proceedings Logic in Computer Science, LICS '90*, pages 498–507, 1990.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[28] Jeremy Gibbons. Generic downwards accumulations. *Science of Computer Programming*, 2000. In press.

[29] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *International Conference on Functional Programming*, September 1998.

[30] T. Hagino. *Category Theoretic Approach to Data Types*. PhD thesis, University of Edinburgh, 1987.

[31] J. Halenbeek. Comparing approaches to polytypic programming. Master's thesis, Department of Computer Science, Utrecht University, 1998.

[32] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Symposium on Principles of Programming Languages, POPL '95*, pages 130–141, 1995.

[33] Ralf Hinze. A generic programming extension for Haskell. In Erik Meijer, editor, *Proceedings of the Third Haskell Workshop*, Technical report of Utrecht University, UU-CS-1999-28, 1999.

[34] Ralf Hinze. Polytypic functions over nested datatypes. *Discrete Mathematics and Theoretical Computer Science*, 3(4):159–180, September 1999.

[35] Ralf Hinze. A new approach to generic functional programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'00*, 2000.

[36] Ralf Hinze. Polytypic values possess polykinded types. In *Mathematics of Program Construction*, 2000.

[37] P. F. Hoogendijk. *A generic theory of datatypes*. PhD thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands, 1996.

[38] Paul Hoogendijk and Roland Backhouse. When do datatypes commute? In *Category Theory and Computer Science*, volume 1290 of *LNCS*, pages 242–260, 1997.

[39] Paul Hoogendijk and Oege de Moor. Container types categorically. *Journal of Functional Programming*, 2000.

[40] James Hook and Tim Sheard. A semantics of compile-time reflection. Oregon Graduate Institute of Science and Technology, Beaverton, OR, USA, 1993.

[41] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, Amsterdam, The Netherlands, 1997. ACM Press.

[42] John Hughes. Generalising monads to arrows. To appear in the special issue on Mathematics of Program Construction of Science of Computer Programming, 2000.

[43] Marieke Huisman. The calculation of a polytypic parser. Master's thesis, Utrecht University, 1996. INF/SRC-96-19.

[44] Graham Hutton and Jeremy Gibbons. The generic approximation lemma. Submitted for publication, 2000.

[45] P. Jansson. Polytypism and polytypic unification. Master's thesis, Computing Science, Chalmers University of Technology, 1995. Available from the Polytypic programming WWW page [49].

[46] P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[47] P. Jansson and J. Jeuring. PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand, June 1998. Available from the Polytypic programming WWW page [49].

[48] P. Jansson and J. Jeuring. Polytypic compact printing and parsing. In Doaitse Swierstra, editor, *Proceedings of the 8th European Symposium on Programming, ESOP'99*, volume 1576 of *LNCS*, pages 273–287. Springer-Verlag, 1999.

[49] Patrik Jansson. Polytypic programming. The WWW home page for polytypism: `http://www.cs.chalmers.se/~patrikj/poly/`.

[50] Patrik Jansson. Functional polytypic programming — use and implementation. Licentiate thesis, Computing Science, Chalmers University of Technology, 1997. Available from the Polytypic programming WWW page [49].

[51] Patrik Jansson and Johan Jeuring. Functional pearl: Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, September 1998.

[52] Patrik Jansson and Johan Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In *Workshop on Generic Programming*, 2000.

[53] Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. Submitted for publication, 2000.

[54] Patrik Jansson and Johan Jeuring. Rewriting as a polytypic application. Work in progress, 2000.

[55] C. Barry Jay. Polynomial polymorphism. In *Proceedings of the Eighteenth Australasian Computer Science Conference*, pages 237–243, 1995.

[56] C. Barry Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995. Also in ESOP '94.

[57] C.B. Jay. Functorial lambda-calculus. Work in progress, 2000.

[58] C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.

[59] C.B. Jay and P.A. Steckler. The functional imperative: shape! In Chris Hankin, editor, *Programming languages and systems: 7th European Symposium on Programming, ESOP'98*, volume 1381 of *LNCS*, pages 139–53. Springer-Verlag, 1998.

[60] J. Jeuring. Algorithms from theorems. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, pages 247–266. North-Holland, 1990.

[61] J. Jeuring. Polytypic pattern matching. In *FPCA'95*, pages 238–248. ACM Press, 1995.

[62] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming '96*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, 1996.

[63] Mark P. Jones. Dictionary-free overloading by partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, Florida, June 1994.

[64] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[65] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 97–136. Springer-Verlag, 1995.

[66] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, 1995.

[67] Mark P. Jones. Typing haskell in haskell. In Erik Meijer, editor, *Proceedings of the Third Haskell Workshop*, Technical Report of Utrecht University, UU-CS-1999-28, 1999.

[68] K. Knight. Unification: A multidisciplinary survey. *Computing Surveys*, 21(1):93–124, 1989.

[69] D.J. Lehmann and M.B. Smyth. Algebraic specification of data types: A synthetic approach. *Math. Systems Theory*, 14:97–139, 1981.

[70] K.J. Lieberherr, I. Silva-Lepe, and C. Xiao. Adaptive object-oriented programming — using graph-based customization. *Communications of the ACM*, pages 94–101, 1994.

[71] G. Malcolm. Homomorphisms and promotability. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *LNCS*, pages 335–347. Springer-Verlag, 1989.

[72] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[73] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Text and Monographs in Computer Science. Springer-Verlag, 1986.

[74] L. Meertens. Algorithmics — towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North–Holland, 1986.

[75] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

[76] L. Meertens. Calculate polytypically! In *PLILP'96*, volume 1140 of *LNCS*, pages 1–16. Springer-Verlag, 1996.

[77] E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In J. Hughes, editor, *FPCA'91: Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, 1991.

[78] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 324–333, 1995.

[79] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In *Advanced Functional Programming, AFP'95*, volume 925 of *LNCS*, pages 228–266. Springer-Verlag, 1995.

[80] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[81] E. Moggi, G. Bellè, and C.B. Jay. Monads, shapely functors and traversals. In *Category Theory and Computer Science, CTCS'99*, volume 29 of *ENTCS*, pages 265–286. Elsevier, 1999.

[82] O. de Moor. Categories, relations and dynamic programming. *Mathematical Structures in Computer Science*, 4:33–69, 1994.

[83] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[84] Mathematics of Program Construction Group (Eindhoven Technical University). Fixed-point calculus. *Information Processing Letters*, 53(3):131–136, 1995.

[85] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, February 1997.

[86] J. Palsberg, C. Xiao, and K. Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, 1995.

[87] A. Pardo. Monadic corecursion — definition, fusion laws, and applications. *Electronic Notes in Theoretical Computer Science*, 11, 1998.

[88] Ross Paterson. General arrow constructions. Work in progress, 1999.

[89] Ross Paterson. Embedding a class of domain-specific languages in a functional language. In submission, 2000.

[90] Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/definition/`, February 1999.

[91] Holger Pfeifer and Harald Rueß. Polytypic proof construction. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Proc. 12th Intl. Conf. on Theorem Proving in Higher Order Logics*, number 1690 in Lecture Notes in Computer Science, pages 55–72. Springer-Verlag, 1999.

[92] P.J. Plauger, Alexander A. Stepanov, Meng Lee, and David R. Musser. *The Standard Template Library*. Prentice Hall, 1997.

[93] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[94] Fritz Ruehr. *Analytical and Structural Polymorphism Expressed Using Patterns Over Types*. PhD thesis, University of Michigan, 1992.

[95] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown Publishers, 1988.

[96] T. Sheard and L. Fegaras. A fold for all seasons. In *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture FPCA '93*, pages 233–242. ACM Press, June 93.

[97] T. Sheard and N. Nelson. Type safe abstractions using program generators. Technical Report 95-013, Oregon Graduate Institute of Science and Technology, Portland, OR, USA, 1995.

[98] Tim Sheard. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, 1991.

[99] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, 1995.

[100] Måns Vestin. Genetic algorithms in Haskell with polytypic programming. Examensarbeten 1997:36, Göteborg University, Gothenburg, Sweden, 1997. Available from the Polytypic programming WWW page [49].

[101] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, FPCA '89*, pages 347–359. ACM Press, 1989.

[102] P. Wadler. Comprehending monads. In *Proceedings 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78, 1990.

[103] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990. Presented at ESOP'88.

[104] M. Wallace and C. Runciman. Heap compression and binary I/O in haskell. In *2nd ACM Haskell Workshop*, 1997.

Appendix A

An implementation of PolyLib

This appendix presents the implementation of the polytypic function library PolyLib (Chapter 5) as PolyP code. All functions from Chapter 5 are implemented and also a few variants and extensions. Each section is presented as a Haskell style module, but the current version of PolyP ignores the information in the module head. Each module is a literate script containing the source code and some typesetting information. The LaTeX source used to typeset this appendix was automatically generated by Ralf Hinze's *lhs2tex* program.

A.1 Structured recursion operators

$$\textbf{module}\ \mathit{Base}\ (\mathit{pmap}, \mathit{fmap2}, \mathit{cata}, \mathit{ana}, \mathit{hylo}, \mathit{para}, (\ \textrm{-}\!\!*\!\!\textrm{-}\ ), (\ \textrm{-}\!\!+\!\!\textrm{-}\ ))\ \textbf{where}$$

$$\begin{array}{l}
\mathit{pmap} :: \mathit{Regular}\ d \Rightarrow (a \to b) \to d\ a \to d\ b \\
\mathit{pmap}\ f\ =\ \mathit{inn} \circ \mathit{fmap2}\ f\ (\mathit{pmap}\ f) \circ \mathit{out} \\
\textbf{polytypic}\ \mathit{fmap2} :: (a \to c) \to (b \to d) \to f\ a\ b \to f\ c\ d \\
\quad =\ \lambda p\ r \to \textbf{case}\ f\ \textbf{of}
\end{array}$$

$$\begin{array}{lcl}
g + h & \to & (\mathit{fmap2}\ p\ r) \ \textrm{-}\!\!+\!\!\textrm{-}\ (\mathit{fmap2}\ p\ r) \\
g * h & \to & (\mathit{fmap2}\ p\ r) \ \textrm{-}\!\!*\!\!\textrm{-}\ (\mathit{fmap2}\ p\ r) \\
\mathit{Empty} & \to & \mathit{id} :: () \to () \\
\mathit{Par} & \to & p \\
\mathit{Rec} & \to & r \\
d @ g & \to & \mathit{pmap}\ (\mathit{fmap2}\ p\ r) \\
\mathit{Const}\ t & \to & \mathit{id}
\end{array}$$

$$\begin{array}{lcl}
\mathit{cata} & :: & \mathit{Regular}\ d \Rightarrow (\Phi_d\ a\ b \to b) \to (d\ a \to b) \\
\mathit{cata}\ i & = & i \circ \mathit{fmap2}\ \mathit{id}\ (\mathit{cata}\ i) \circ \mathit{out}
\end{array}$$

$$
\begin{array}{lcl}
ana & :: & Regular\ d \Rightarrow (b \to \Phi_d\ a\ b) \to (b \to d\ a) \\
ana\ o & = & inn \circ fmap2\ id\ (ana\ o) \circ o
\end{array}
$$

$$
\begin{array}{lcl}
hylo & :: & Bifunctor\ f \Rightarrow (f\ a\ b \to b) \to (c \to f\ a\ c) \to c \to b \\
hylo\ i\ o & = & i \circ fmap2\ id\ (hylo\ i\ o) \circ o
\end{array}
$$

$$
\begin{array}{lcl}
para & :: & Regular\ d \Rightarrow (d\ a \to \Phi_d\ a\ b \to b) \to d\ a \to b \\
para\ i\ x & = & i\ x\ (fmap2\ id\ (para\ i)\ (out\ x))
\end{array}
$$

Non-polytypic help functions

$$
\begin{array}{lcl}
(\ {-\!\!*\!\!-}\ ) & :: & (a \to c) \to (b \to d) \to ((a,b) \to (c,d)) \\
(\ {-\!\!+\!\!-}\ ) & :: & (a \to c) \to (b \to d) \to (Either\ a\ b \to Either\ c\ d) \\
f {-\!\!*\!\!-} g & = & \lambda(x,y) \to (f\ x, g\ y) \\
f {-\!\!+\!\!-} g & = & Left \circ f\ \triangledown\ Right \circ g
\end{array}
$$

A.2 *Crush*

module *Crush* $(crush, fcrush)$ **where**
import *Base* $(cata, pmap)$

$$
\begin{array}{l}
crush :: Regular\ d \Rightarrow (a \to a \to a) \to a \to d\ a \to a \\
crush\ op\ e\ =\ cata\ (fcrush\ op\ e)
\end{array}
$$

polytypic $fcrush :: (a \to a \to a) \to a \to f\ a\ a \to a$
$= \lambda op\ e \to$ **case** f **of**

$$
\begin{array}{lcl}
g + h & \to & fcrush\ op\ e\ \triangledown\ fcrush\ op\ e \\
g * h & \to & \lambda(x,y) \to op\ (fcrush\ op\ e\ x) \\
 & & \qquad\qquad\qquad (fcrush\ op\ e\ y) \\
Empty & \to & \lambda x \to e \\
Par & \to & id \\
Rec & \to & id \\
d@g & \to & crush\ op\ e \circ pmap\ (fcrush\ op\ e) \\
Const\ t & \to & \lambda x \to e
\end{array}
$$

A.3 Monadic recursion operators

```
module BaseM (pmapM, fmap2M, cataM, anaM, hyloM, paraM,
              innM, outM, idM, (@@)) where
import Base (( -*- ))

pmapM :: (Regular d, Monad m) ⇒ (a → m b) → d a → m (d b)
pmapM fM = liftM inn ∘ fmap2M fM (pmapM fM) ∘ out

polytypic fmap2M :: Monad m ⇒ (a → m c) → (b → m d) →
                                f a b → m (f c d)
  = λp r → case f of
             g + h     →  summapM (fmap2M p r) (fmap2M p r)
             g ∗ h     →  prodmapM (fmap2M p r) (fmap2M p r)
             Empty     →  return
             Par       →  p
             Rec       →  r
             d@g       →  pmapM (fmap2M p r)
             Const t   →  return

summapM :: Monad m ⇒ (a → m c) → (d → m e) →
                      Either a d → m (Either c e)
summapM f g = (liftM Left ∘ f) ▽ (liftM Right ∘ g)
prodmapM :: Monad m ⇒ (a → m c) → (d → m e) → (a, d) → m (c, e)
prodmapM f g p = f (fst p) ≫= λx → g (snd p) ≫= λy → return (x, y)
prodmapMr f g p = g (snd p) ≫= λy → f (fst p) ≫= λx → return (x, y)

cataM :: (Regular d, Monad m) ⇒ (Φ_d a b → m b) → (d a → m b)
cataM iM = iM @@ fmap2M idM (cataM iM) ∘ out

anaM :: (Regular d, Monad m) ⇒ (b → m Φ_d a b) → (b → m (d a))
anaM oM = liftM inn ∘ fmap2M idM (anaM oM) @@ oM

hyloM :: (Bifunctor f, Monad m) ⇒ (f a b → m b) → (c → m (f a c)) →
                                   c → m b
hyloM iM oM = iM @@ fmap2M idM (hyloM iM oM) @@ oM

paraM :: (Regular d, Monad m) ⇒ (d a → Φ_d a b → m b) → d a → m b
paraM iM x = iM x =≪ fmap2M idM (paraM iM) (out x)
```

New names for symmetry:

$$
\begin{aligned}
&\mathit{idM} :: \mathit{Monad}\ m \Rightarrow\ a \to m\ a\\
&\mathit{idM}\ =\ \mathit{return}\\
&\mathit{innM} :: (\mathit{Regular}\ d, \mathit{Monad}\ m) \Rightarrow\ \Phi_d\ a\ (d\ a) \to m\ (d\ a)\\
&\mathit{innM}\ =\ \mathit{idM} \circ \mathit{inn}\\
&\mathit{outM} :: (\mathit{Regular}\ d, \mathit{Monad}\ m) \Rightarrow\ d\ a \to m\ \Phi_d\ a\ (d\ a)\\
&\mathit{outM}\ =\ \mathit{idM} \circ \mathit{out}
\end{aligned}
$$

A synonym.

$$
\begin{aligned}
&\mathit{pmapMl} :: (\mathit{Regular}\ d, \mathit{Monad}\ m) \Rightarrow\ (a \to m\ b) \to d\ a \to m\ (d\ b)\\
&\mathit{pmapMl}\ =\ \mathit{pmapM}
\end{aligned}
$$

Reverse order traversals

$$
\begin{aligned}
&\mathit{pmapMr} :: (\mathit{Regular}\ d, \mathit{Monad}\ m) \Rightarrow\ (a \to m\ b) \to d\ a \to m\ (d\ b)\\
&\mathit{pmapMr}\ \mathit{fM}\ =\ \mathit{liftM}\ \mathit{inn} \circ \mathit{fmap2Mr}\ \mathit{fM}\ (\mathit{pmapMr}\ \mathit{fM}) \circ \mathit{out}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{polytypic}\ \mathit{fmap2Mr} :: \mathit{Monad}\ m \Rightarrow\\
&\qquad\qquad (a \to m\ c) \to (b \to m\ d) \to f\ a\ b \to m\ (f\ c\ d)\\
&\quad = \lambda p\ r \to \textbf{case}\ f\ \textbf{of}\\
&\qquad\qquad \begin{array}{lcl}
g + h & \to & \mathit{summapM}\ (\mathit{fmap2Mr}\ p\ r)\ (\mathit{fmap2Mr}\ p\ r)\\
g * h & \to & \mathit{prodmapMr}\ (\mathit{fmap2Mr}\ p\ r)\ (\mathit{fmap2Mr}\ p\ r)\\
\mathit{Empty} & \to & \mathit{return}\\
\mathit{Par} & \to & p\\
\mathit{Rec} & \to & r\\
d@g & \to & \mathit{pmapMr}\ (\mathit{fmap2Mr}\ p\ r)\\
\mathit{Const}\ t & \to & \mathit{return}
\end{array}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{cataMr} :: (\mathit{Regular}\ d, \mathit{Monad}\ m) \Rightarrow\ (\Phi_d\ a\ b \to m\ b) \to (d\ a \to m\ b)\\
&\mathit{cataMr}\ \mathit{iM}\ =\ \mathit{iM}\ @@\ \mathit{fmap2Mr}\ \mathit{idM}\ (\mathit{cataMr}\ \mathit{iM}) \circ \mathit{out}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{anaMr} :: (\mathit{Regular}\ d, \mathit{Monad}\ m) \Rightarrow\ (b \to m\ \Phi_d\ a\ b) \to (b \to m\ (d\ a))\\
&\mathit{anaMr}\ \mathit{oM}\ =\ \mathit{liftM}\ \mathit{inn} \circ \mathit{fmap2Mr}\ \mathit{idM}\ (\mathit{anaMr}\ \mathit{oM})\ @@\ \mathit{oM}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{hyloMr} :: (\mathit{Bifunctor}\ f, \mathit{Monad}\ m) \Rightarrow\ (f\ a\ b \to m\ b) \to (c \to m\ (f\ a\ c)) \to\\
&\qquad\qquad c \to m\ b\\
&\mathit{hyloMr}\ \mathit{iM}\ \mathit{oM}\ =\ \mathit{iM}\ @@\ \mathit{fmap2Mr}\ \mathit{idM}\ (\mathit{hyloMr}\ \mathit{iM}\ \mathit{oM})\ @@\ \mathit{oM}
\end{aligned}
$$

Traversal either way: $pmapM'$ ($True \rightarrow$ left to right, $False \rightarrow$ right to left)

$$pmapM' :: (Regular\ d, Monad\ m) \Rightarrow Bool \rightarrow (a \rightarrow m\ b) \rightarrow d\ a \rightarrow m\ (d\ b)$$
$$pmapM'\ ord\ fM\ =\ liftM\ inn \circ fmap2M'\ ord\ fM\ (pmapM'\ ord\ fM) \circ out$$

$$
\begin{array}{l}
\textbf{polytypic}\ fmap2M' :: Monad\ m \Rightarrow Bool \rightarrow (a \rightarrow m\ c) \rightarrow (b \rightarrow m\ d) \rightarrow \\
\qquad\qquad\qquad\qquad f\ a\ b \rightarrow m\ (f\ c\ d) \\
\quad = \lambda ord\ p\ r \rightarrow \textbf{case}\ f\ \textbf{of} \\
\qquad \begin{array}{lcl}
g + h & \rightarrow & summapM\ (fmap2M'\ ord\ p\ r)\ (fmap2M'\ ord\ p\ r) \\
g * h & \rightarrow & opM\ ord \circ (fmap2M'\ ord\ p\ r \mathbin{-\!\!*} fmap2M'\ ord\ p\ r) \\
Empty & \rightarrow & return \\
Par & \rightarrow & p \\
Rec & \rightarrow & r \\
d@g & \rightarrow & pmapM'\ ord\ (fmap2M'\ ord\ p\ r) \\
Const\ t & \rightarrow & return
\end{array}
\end{array}
$$

$$
\begin{array}{l}
opM :: Monad\ m \Rightarrow Bool \rightarrow (m\ a, m\ b) \rightarrow m\ (a, b) \\
opM\ b\ p\ =\ \textbf{case}\ b\ \textbf{of} \\
\qquad\qquad True \rightarrow fst\ p \gg\!= \lambda x \rightarrow snd\ p \gg\!= \lambda y \rightarrow return\ (x, y) \\
\qquad\qquad False \rightarrow snd\ p \gg\!= \lambda y \rightarrow fst\ p \gg\!= \lambda x \rightarrow return\ (x, y)
\end{array}
$$

Monad operations (that are not in PolyP's prelude)

$$
\begin{array}{l}
liftM :: Monad\ m \Rightarrow (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b \\
liftM\ f\ mx\ =\ mx \gg\!= \lambda x \rightarrow return\ (f\ x) \\
(@@) :: Monad\ m \Rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ b) \rightarrow (a \rightarrow m\ c) \\
f\ @@\ g\ =\ \lambda y \rightarrow g\ y \gg\!= f
\end{array}
$$

A.4 *Thread*

module *Thread* (*thread*, *pmapM*, *fthread*, *fmap2M*) **where**
import *Base* (*cata*, *inn*, *pmap*)
import *BaseM* (*pmapM*, *fmap2M*, (@@))

$$
\begin{array}{l}
\mathit{thread} :: (\mathit{Regular}\ d, \mathit{Monad}\ m) \Rightarrow d\ (m\ a) \rightarrow m\ (d\ a) \\
\mathit{thread}\ =\ \mathit{cata}\ (\mathit{liftM}\ \mathit{inn} \circ \mathit{fthread}) \\
\textbf{polytypic}\ \mathit{fthread}\ ::\ \mathit{Monad}\ m \Rightarrow f\ (m\ a)\ (m\ b) \rightarrow m\ (f\ a\ b) \\
\quad =\ \textbf{case}\ f\ \textbf{of} \\
\qquad g + h \quad \rightarrow\ \mathit{sumthread} \circ (\mathit{fthread} + \mathit{fthread}) \\
\qquad g * h \quad \rightarrow\ \mathit{prodthread} \circ (\mathit{fthread} * \mathit{fthread}) \\
\qquad \mathit{Empty} \quad \rightarrow\ \mathit{return} \\
\qquad \mathit{Par} \quad \rightarrow\ \mathit{id} \\
\qquad \mathit{Rec} \quad \rightarrow\ \mathit{id} \\
\qquad d@g \quad \rightarrow\ \mathit{thread} \circ (\mathit{pmap}\ \mathit{fthread}) \\
\qquad \mathit{Const}\ t \quad \rightarrow\ \mathit{return}
\end{array}
$$

$$
\begin{array}{l}
\mathit{sumthread} :: \mathit{Monad}\ m \Rightarrow \mathit{Either}\ (m\ a)\ (m\ b) \rightarrow m\ (\mathit{Either}\ a\ b) \\
\mathit{sumthread}\ =\ \mathit{liftM}\ \mathit{Left}\ \triangledown\ \mathit{liftM}\ \mathit{Right}
\end{array}
$$

$$
\begin{array}{l}
\mathit{prodthread} :: \mathit{Monad}\ m \Rightarrow (m\ a, m\ b) \rightarrow m\ (a, b) \\
\mathit{prodthread}\ (mx, my)\ =\ mx \ggg\!= \lambda x \rightarrow my \ggg\!= \lambda y \rightarrow \mathit{return}\ (x, y)
\end{array}
$$

Monad operations (that are not in PolyP's prelude)

$$
\begin{array}{l}
\mathit{liftM} :: \mathit{Monad}\ m \Rightarrow (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b \\
\mathit{liftM}\ f\ mx\ =\ mx \ggg\!= \lambda x \rightarrow \mathit{return}\ (f\ x)
\end{array}
$$

Alternative definitions of *pmapM* and *fmap2M*:

$$
\begin{array}{l}
\mathit{pmapM} :: (\mathit{Regular}\ d, \mathit{Monad}\ m) \Rightarrow (a \rightarrow m\ b) \rightarrow d\ a \rightarrow m\ (d\ b) \\
\mathit{pmapM}\ f\ =\ \mathit{thread} \circ \mathit{pmap}\ f \\
\mathit{fmap2M} :: (\mathit{Bifunctor}\ f, \mathit{Monad}\ m) \Rightarrow (a \rightarrow m\ c) \rightarrow (b \rightarrow m\ d) \rightarrow \\
\qquad\qquad f\ a\ b \rightarrow m\ (f\ c\ d) \\
\mathit{fmap2M}\ f\ g\ =\ \mathit{fthread} \circ \mathit{fmap2}\ f\ g
\end{array}
$$

A.5 *ThreadFuns*

$$
\begin{array}{l}
\textbf{module}\ \mathit{ThreadFuns}\ (\mathit{propagate}, \mathit{cross})\ \textbf{where} \\
\textbf{import}\ \mathit{Thread}\ (\mathit{thread})
\end{array}
$$

$$
\begin{array}{l}
\mathit{cross} :: \mathit{Regular}\ d \Rightarrow d\ [a] \rightarrow [d\ a] \\
\mathit{cross}\ =\ \mathit{thread}
\end{array}
$$

$$
\begin{array}{l}
\mathit{propagate} :: \mathit{Regular}\ d \Rightarrow d\ (\mathit{Maybe}\ a) \rightarrow \mathit{Maybe}\ (d\ a) \\
\mathit{propagate}\ =\ \mathit{thread}
\end{array}
$$

A.6 *Propagate*

$$\textbf{module}\ \mathit{Propagate}\ (\mathit{propagate}, \mathit{fprop}, \mathit{sumprop}, \mathit{prodprop}, \mathit{mapM})\ \textbf{where}$$
$$\textbf{import}\ \mathit{Base}\ (\mathit{cata}, \mathit{inn}, \mathit{pmap})$$

$$\begin{array}{l}
\mathit{propagate} :: \mathit{Regular}\ d \Rightarrow d\ (\mathit{Maybe}\ a) \to \mathit{Maybe}\ (d\ a) \\
\mathit{propagate}\ =\ \mathit{cata}\ (\mathit{mapM}\ \mathit{inn} \circ \mathit{fprop})
\end{array}$$

$$\begin{array}{lll}
\textbf{polytypic}\ \mathit{fprop} & :: & f\ (\mathit{Maybe}\ a)\ (\mathit{Maybe}\ b) \to \mathit{Maybe}\ (f\ a\ b) \\
\quad = \textbf{case}\ f\ \textbf{of} & & \\
\qquad g + h & \to & \mathit{sumprop} \circ (\mathit{fprop} +\!\!\!+ \mathit{fprop}) \\
\qquad g * h & \to & \mathit{prodprop} \circ (\mathit{fprop} -\!\!\!*\!\!- \mathit{fprop}) \\
\qquad \mathit{Empty} & \to & \mathit{Just} \\
\qquad \mathit{Par} & \to & \mathit{id} \\
\qquad \mathit{Rec} & \to & \mathit{id} \\
\qquad d@g & \to & \mathit{propagate} \circ (\mathit{pmap}\ \mathit{fprop}) \\
\qquad \mathit{Const}\ t & \to & \mathit{Just}
\end{array}$$

$$\begin{array}{l}
\mathit{sumprop} :: \mathit{Either}\ (\mathit{Maybe}\ a)\ (\mathit{Maybe}\ b) \to \mathit{Maybe}\ (\mathit{Either}\ a\ b) \\
\mathit{sumprop}\ =\ \mathit{mapM}\ \mathit{Left}\ \triangledown\ \mathit{mapM}\ \mathit{Right}
\end{array}$$

$$\begin{array}{lll}
\mathit{prodprop} :: (\mathit{Maybe}\ a, \mathit{Maybe}\ b) & \to & \mathit{Maybe}\ (a, b) \\
\mathit{prodprop}\ p\ =\ \textbf{case}\ p\ \textbf{of} & & \\
\qquad\qquad (\mathit{Just}\ x, \mathit{Just}\ y) & \to & \mathit{Just}\ (x, y) \\
\qquad\qquad _ & \to & \mathit{Nothing}
\end{array}$$

Maybe functions

$$\begin{array}{l}
\mathit{mapM} :: (a \to b) \to \mathit{Maybe}\ a \to \mathit{Maybe}\ b \\
\mathit{mapM}\ f\ =\ \mathit{maybe}\ \mathit{Nothing}\ (\mathit{Just} \circ f)
\end{array}$$

A.7 *Zip*

$$\textbf{module}\ \mathit{Zip}\ (\mathit{pzip}, \mathit{fzip}, \mathit{pzipWith}, \mathit{pzipWith}', \mathit{fzipWith}, \mathit{fzipWith})\ \textbf{where}$$
$$\textbf{import}\ \mathit{Base}\ (\mathit{pmap}, \mathit{fmap2}, (\ +\!\!\!+\ ), (\ -\!\!\!*\!\!-\ ))$$
$$\textbf{import}\ \mathit{Propagate}\ (\mathit{fprop}, \mathit{sumprop}, \mathit{prodprop}, \mathit{propagate}, \mathit{mapM})$$

In this module *Maybe* could be replaced by any *Monad* using *fail* `"err"` for *zeroM*.

$$\begin{array}{l}
\mathit{punzip} :: \mathit{Regular}\ d \Rightarrow d\ (a,b) \rightarrow (d\ a, d\ b) \\
\mathit{punzip}\ x\ =\ (\mathit{pmap}\ \mathit{fst}\ x, \mathit{pmap}\ \mathit{snd}\ x)
\end{array}$$

$$\begin{array}{l}
\mathit{funzip} :: \mathit{Bifunctor}\ f \Rightarrow f\ (a,c)\ (b,d) \rightarrow (f\ a\ b, f\ c\ d) \\
\mathit{funzip}\ x\ =\ (\mathit{fmap2}\ \mathit{fst}\ \mathit{fst}\ x, \mathit{fmap2}\ \mathit{snd}\ \mathit{snd}\ x)
\end{array}$$

$$\begin{array}{l}
\mathit{pzip} :: \mathit{Regular}\ d \Rightarrow (d\ a, d\ b) \rightarrow \mathit{Maybe}\ (d\ (a,b)) \\
\mathit{pzip}\ =\ (\mathit{innM}\ @@\ (\mathit{fprop} \circ \mathit{fmap2}\ \mathit{returnM}\ \mathit{pzip})\ @@\ \mathit{fzip}) \circ (\mathit{out} \mathbin{-\!\!*\!\!-} \mathit{out})
\end{array}$$

$$\begin{array}{l}
\mathit{pzipWith}' :: \mathit{Regular}\ d \Rightarrow (\Phi_d\ c\ e \rightarrow e) \rightarrow \\
\qquad\qquad\qquad\qquad\quad ((d\ a, d\ b) \rightarrow e) \rightarrow \\
\qquad\qquad\qquad\qquad\quad ((a,b) \rightarrow c) \rightarrow (d\ a, d\ b) \rightarrow e \\
\mathit{pzipWith}'\ \mathit{ins}\ \mathit{fail}\ \mathit{op}\ (x,y)\ = \\
\quad \mathit{maybe}\ (\mathit{fail}\ (x,y))\ (\mathit{ins} \circ \mathit{fmap2}\ \mathit{op}\ (\mathit{pzipWith}'\ \mathit{ins}\ \mathit{fail}\ \mathit{op})) \\
\qquad\quad (\mathit{fzip}\ (\mathit{out}\ x, \mathit{out}\ y))
\end{array}$$

A possible variant:

$$\begin{array}{l}
\mathit{pzipWith}'\ \mathit{ins}\ \mathit{fail}\ \mathit{op}\ (x,y)\ = \\
\quad \mathit{maybe}\ (\mathit{fail}\ (x,y))\ \mathit{ins}\ (\mathit{fzipWith}\ \mathit{op}\ (\mathit{pzipWith}'\ \mathit{ins}\ \mathit{fail}\ \mathit{op})\ (\mathit{out}\ x, \mathit{out}\ y))
\end{array}$$

$$\begin{array}{l}
\mathit{pzipWith} :: \mathit{Regular}\ d \Rightarrow ((a,b) \rightarrow \mathit{Maybe}\ c) \rightarrow (d\ a, d\ b) \rightarrow \mathit{Maybe}\ (d\ c) \\
\mathit{pzipWith}\ =\ \mathit{pzipWith}'\ (\mathit{mapM}\ \mathit{inn} \circ \mathit{fprop})\ (\mathit{const}\ \mathit{zeroM})
\end{array}$$

Note: the parameters to *fzipWith* do not have the the same types as the arguments to *pzipWith*.

$$\begin{array}{l}
\mathit{fzipWith} :: ((a,a') \rightarrow c) \rightarrow ((b,b') \rightarrow d) \rightarrow (f\ a\ b, f\ a'\ b') \rightarrow \mathit{Maybe}\ (f\ c\ d) \\
\mathit{fzipWith}\ f\ g\ =\ \mathit{mapM}\ (\mathit{fmap2}\ f\ g) \circ \mathit{fzip}
\end{array}$$

$$\begin{array}{llll}
\multicolumn{2}{l}{\textbf{polytypic}\ \mathit{fzip}} & :: & (f\ a\ b, f\ c\ d) \rightarrow \mathit{Maybe}\ (f\ (a,c)\ (b,d)) \\
\multicolumn{4}{l}{\quad = \textbf{case}\ f\ \textbf{of}} \\
 & g + h & \rightarrow & (\mathit{sumprop} \circ (\mathit{fzip} \mathbin{-\!\!+\!\!-} \mathit{fzip}))\ @@\ \mathit{sumzip} \\
 & g * h & \rightarrow & (\mathit{prodprop} \circ (\mathit{fzip} \mathbin{-\!\!*\!\!-} \mathit{fzip}))\ @@\ \mathit{prodzip} \\
 & \mathit{Empty} & \rightarrow & \mathit{const}\ (\mathit{returnM}\ ()) \\
 & \mathit{Par} & \rightarrow & \mathit{returnM} \\
 & \mathit{Rec} & \rightarrow & \mathit{returnM} \\
 & d@g & \rightarrow & (\mathit{propagate} \circ (\mathit{pmap}\ \mathit{fzip}))\ @@\ \mathit{pzip} \\
 & \mathit{Const}\ t & \rightarrow & \mathit{constzip}
\end{array}$$

$$\begin{array}{lll} \mathit{sumzip} :: (\mathit{Either}\ a\ b, \mathit{Either}\ c\ d) & \to & \mathit{Maybe}\ (\mathit{Either}\ (a,c)\ (b,d)) \\ \mathit{sumzip}\ p\ =\ \textbf{case}\ p\ \textbf{of} & & \\ \quad (\mathit{Left}\ s, \mathit{Left}\ t) & \to & \mathit{returnM}\ (\mathit{Left}\ (s,t)) \\ \quad (\mathit{Right}\ s, \mathit{Right}\ t) & \to & \mathit{returnM}\ (\mathit{Right}\ (s,t)) \\ \quad \_ & \to & \mathit{zeroM} \end{array}$$

$$\begin{array}{l} \mathit{prodzip} :: ((a,b),(c,d)) \to \mathit{Maybe}\ ((a,c),(b,d)) \\ \mathit{prodzip}\ ((a,b),(c,d))\ =\ \mathit{returnM}\ ((a,c),(b,d)) \end{array}$$

Using this definition of *constzip* in the $\mathit{Const}\ t$ case, formally requires an $\mathit{Eq}\ t$ constraint, which is inexpressible in PolyP (in this position). However the implementation of PolyP allows this for convenience, even though it is not really type safe.

$$\begin{array}{l} \mathit{constzip} :: \mathit{Eq}\ t \Rightarrow (t,t) \to \mathit{Maybe}\ t \\ \mathit{constzip}\ (x,y)\ =\ \textbf{if}\ x == y\ \textbf{then}\ \mathit{returnM}\ x\ \textbf{else}\ \mathit{zeroM} \end{array}$$

The intended (and implemented) meaning is fairly clear: one branch $\mathit{Const}\ T \to \mathit{constzip}$ in the polytypic case for each type T that is an instance of Eq.

Maybe-monad functions

$$\begin{array}{l} \mathit{returnM} :: a \to \mathit{Maybe}\ a \\ \mathit{returnM}\ x\ =\ \mathit{Just}\ x \end{array}$$

$$\begin{array}{l} \mathit{bindM} :: \mathit{Maybe}\ a \to (a \to \mathit{Maybe}\ b) \to \mathit{Maybe}\ b \\ \mathit{bindM}\ x\ f\ =\ \mathit{maybe}\ \mathit{Nothing}\ f\ x \end{array}$$

$$\begin{array}{l} (@@) :: (a \to \mathit{Maybe}\ b) \to (c \to \mathit{Maybe}\ a) \to c \to \mathit{Maybe}\ b \\ g\ @@\ f\ =\ \lambda a \to f\ a\ \text{`}\mathit{bindM}\text{`}\ g \end{array}$$

$$\begin{array}{l} \mathit{zeroM} :: \mathit{Maybe}\ a \\ \mathit{zeroM}\ =\ \mathit{Nothing} \end{array}$$

$$\begin{array}{l} \mathit{innM} :: \mathit{Regular}\ d \Rightarrow \Phi_d\ a\ (d\ a) \to \mathit{Maybe}\ (d\ a) \\ \mathit{innM}\ =\ \mathit{returnM} \circ \mathit{inn} \end{array}$$

A.8 *Equal*

$$\textbf{module}\ Equal\ (pequal, fequal, peq)\ \textbf{where}$$

$$\begin{array}{l}
peq :: (Regular\ d, Eq\ a) \Rightarrow d\ a \to d\ a \to Bool \\
peq\ =\ pequal\ (==)
\end{array}$$

$$\begin{array}{l}
pequal :: Regular\ d \Rightarrow (a \to b \to Bool) \to d\ a \to d\ b \to Bool \\
pequal\ eq\ x\ y\ =\ fequal\ eq\ (pequal\ eq)\ (out\ x)\ (out\ y)
\end{array}$$

$$\begin{array}{l}
\textbf{polytypic}\ fequal :: (a \to b \to Bool) \to (c \to d \to Bool) \to \\
\qquad\qquad\qquad f\ a\ c \to f\ b\ d \to Bool \\
\quad =\ \lambda p\ r \to \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad \begin{array}{lcl}
g + h & \to & sumequal\ (fequal\ p\ r)\ (fequal\ p\ r) \\
g * h & \to & prodequal\ (fequal\ p\ r)\ (fequal\ p\ r) \\
Empty & \to & \lambda_\ _ \to True \\
Par & \to & p \\
Rec & \to & r \\
d@g & \to & pequal\ (fequal\ p\ r) \\
Const\ t & \to & (==)
\end{array}
\end{array}$$

$$\begin{array}{l}
sumequal :: (a \to b \to Bool) \to (c \to d \to Bool) \to \\
\qquad\qquad Either\ a\ c \to Either\ b\ d \to Bool \\
sumequal\ f\ g\ a\ b\ =\ \textbf{case}\ (a, b)\ \textbf{of} \\
\qquad\qquad\qquad \begin{array}{lcl}
(Left\ x, Left\ v) & \to & f\ x\ v \\
(Right\ y, Right\ w) & \to & g\ y\ w \\
_ & \to & False
\end{array}
\end{array}$$

$$\begin{array}{l}
prodequal :: (a \to b \to Bool) \to (c \to d \to Bool) \to (a, c) \to (b, d) \to Bool \\
prodequal\ f\ g\ p\ q\ =\ f\ (fst\ p)\ (fst\ q)\ \wedge\ g\ (snd\ p)\ (snd\ q)
\end{array}$$

A slightly less lazy variant:

$$prodequal\ f\ g\ (x, y)\ (v, w)\ =\ f\ x\ v\ \wedge\ g\ y\ w$$

A.9 *Compare*

```
module Compare (pcompare', pcompare, fcompare) where


pcompare' :: (Regular d, Ord a) => d a -> d a -> Ordering
pcompare' = pcompare compare


pcompare :: Regular d => (a -> a -> Ordering) -> d a -> d a -> Ordering
pcompare eq x y = fcompare eq (pcompare eq) (out x) (out y)
polytypic fcompare :: (a -> a -> Ordering) -> (b -> b -> Ordering) ->
                        f a b -> f a b -> Ordering
  = \p r -> case f of
              g + h    -> sumcompare (fcompare p r) (fcompare p r)
              g * h    -> prodcompare (fcompare p r) (fcompare p r)
              Empty    -> \_ _ -> EQ
              Par      -> p
              Rec      -> r
              d@g      -> pcompare (fcompare p r)
              Const t  -> compare


sumcompare :: (a -> a -> Ordering) -> (b -> b -> Ordering) ->
              Either a b -> Either a b -> Ordering
sumcompare f g a b = case (a, b) of
                       (Left x, Left v)     -> f x v
                       (Right y, Right w)   -> g y w
                       (Left _, Right _)    -> LT
                       (Right _, Left _)    -> GT


prodcompare :: (a -> a -> Ordering) -> (b -> b -> Ordering) ->
               (a, b) -> (a, b) -> Ordering
prodcompare f g p q = f (fst p) (fst q) `ordop` g (snd p) (snd q)


ordop :: Ordering -> Ordering -> Ordering
ordop x y = case x of
              EQ -> y
              _  -> x
```

A.10 *Flatten*

module *Flatten* (*flatten*, *fflatten*, *fl_par*, *fl_rec*, *fl_all*, *singleton*, *nil*) **where**
import *Base* (*pmap*, *fmap2*)

$$
\begin{array}{l}
\mathit{flatten} :: \mathit{Regular}\ d \Rightarrow d\ a \to [\,a\,] \\
\mathit{flatten}\ =\ \mathit{fflatten} \circ \mathit{fmap2}\ \mathit{singleton}\ \mathit{flatten} \circ \mathit{out}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{polytypic}\ \mathit{fflatten} & :: & f\ [\,a\,]\ [\,a\,] \to [\,a\,] \\
\quad = \textbf{case}\ f\ \textbf{of} & & \\
\qquad g + h & \to & \mathit{fflatten}\ \triangledown\ \mathit{fflatten} \\
\qquad g * h & \to & \lambda(x, y) \to \mathit{fflatten}\ x +\!\!+\ \mathit{fflatten}\ y \\
\qquad \mathit{Empty} & \to & \mathit{nil} \\
\qquad \mathit{Par} & \to & \mathit{id} \\
\qquad \mathit{Rec} & \to & \mathit{id} \\
\qquad d@g & \to & \mathit{concat} \circ \mathit{flatten} \circ \mathit{pmap}\ \mathit{fflatten} \\
\qquad \mathit{Const}\ t & \to & \mathit{nil}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{fl\_par} & :: & \mathit{Bifunctor}\ f \Rightarrow f\ a\ b \to [\,a\,] \\
\mathit{fl\_rec} & :: & \mathit{Bifunctor}\ f \Rightarrow f\ a\ b \to [\,b\,] \\
\mathit{fl\_all} & :: & \mathit{Bifunctor}\ f \Rightarrow f\ a\ a \to [\,a\,] \\
\mathit{fl\_par} & = & \mathit{fflatten} \circ \mathit{fmap2}\ \mathit{singleton}\ \mathit{nil} \\
\mathit{fl\_rec} & = & \mathit{fflatten} \circ \mathit{fmap2}\ \mathit{nil}\ \mathit{singleton} \\
\mathit{fl\_all} & = & \mathit{fflatten} \circ \mathit{fmap2}\ \mathit{singleton}\ \mathit{singleton}
\end{array}
$$

A variant: defining *flatten* using *cata*:

$$
\mathit{flatten}\ =\ \mathit{cata}\ (\mathit{fflatten} \circ \mathit{fmap2}\ \mathit{singleton}\ \mathit{id})
$$

Function *flatten* can also be defined using *crush* (see the module *CrushFuns*).

$$
\begin{array}{l}
\mathit{substructures} :: \mathit{Regular}\ d \Rightarrow d\ a \to [\,d\ a\,] \\
\mathit{substructures}\ x\ =\ x : \mathit{fflatten}\ (\mathit{fmap2}\ \mathit{nil}\ \mathit{substructures}\ (\mathit{out}\ x))
\end{array}
$$

Help functions for lists

$$
\begin{array}{lll}
\mathit{singleton} & :: & a \to [\,a\,] \\
\mathit{singleton}\ x & = & [\,x\,] \\
\mathit{nil} & :: & a \to [\,b\,] \\
\mathit{nil}\ x & = & [\,]
\end{array}
$$

A.11 *Sum*

module *Sum* (*psum*, *fsum*, *size*) **where**
import *Base* (*cata*, *pmap*)

$psum :: Regular\ d \Rightarrow d\ Int \rightarrow Int$
$psum = cata\ fsum$

$$
\begin{array}{lll}
\textbf{polytypic}\ fsum & :: & f\ Int\ Int \rightarrow Int \\
\quad = \textbf{case}\ f\ \textbf{of} & & \\
\qquad g + h & \rightarrow & fsum \triangledown fsum \\
\qquad g * h & \rightarrow & \lambda(x, y) \rightarrow fsum\ x + fsum\ y \\
\qquad Empty & \rightarrow & \lambda x \rightarrow 0 \\
\qquad Par & \rightarrow & id \\
\qquad Rec & \rightarrow & id \\
\qquad d@g & \rightarrow & psum \circ pmap\ fsum \\
\qquad Const\ t & \rightarrow & \lambda x \rightarrow 0
\end{array}
$$

$size :: Regular\ d \Rightarrow d\ a \rightarrow Int$
$size = psum \circ pmap\ (\lambda\_ \rightarrow 1)$

The functions *psum* and *size* can also be defined using *crush* (see the module *CrushFuns*).

A.12 *CrushFuns*

module *CrushFuns* (*psum*, *prod*, *conc*, *pand*, *por*,
size, *flatten*, *pall*, *pany*, *pelem*) **where**

import *Crush* (*crush*)
import *Base* (*pmap*)

$$
\begin{array}{lll}
psum & :: & Regular\ d \Rightarrow d\ Int \rightarrow Int \\
prod & :: & Regular\ d \Rightarrow d\ Int \rightarrow Int \\
comp & :: & Regular\ d \Rightarrow d\ (a \rightarrow a) \rightarrow (a \rightarrow a) \\
conc & :: & Regular\ d \Rightarrow d\ [a] \rightarrow [a] \\
pand & :: & Regular\ d \Rightarrow d\ Bool \rightarrow Bool \\
por & :: & Regular\ d \Rightarrow d\ Bool \rightarrow Bool
\end{array}
$$

$$
\begin{array}{lcl}
psum & = & crush\ (+)\ 0 \\
prod & = & crush\ (*)\ 1 \\
comp & = & crush\ (\circ)\ id \\
conc & = & crush\ (+\!\!+)\ [\,] \\
pand & = & crush\ (\wedge)\ True \\
por & = & crush\ (\vee)\ False
\end{array}
$$

$$
\begin{array}{lcl}
size & :: & Regular\ d\ \Rightarrow\ d\ a \to Int \\
flatten & :: & Regular\ d\ \Rightarrow\ d\ a \to [\,a\,] \\
pall & :: & Regular\ d\ \Rightarrow\ (a \to Bool) \to d\ a \to Bool \\
pany & :: & Regular\ d\ \Rightarrow\ (a \to Bool) \to d\ a \to Bool \\
pelem & :: & (Regular\ d, Eq\ a)\ \Rightarrow\ a \to d\ a \to Bool
\end{array}
$$

$$
\begin{array}{lcl}
size & = & psum \circ pmap\ (\lambda\_ \to 1) \\
flatten & = & conc \circ pmap\ (\lambda x \to [\,x\,]) \\
pall\ p & = & pand \circ pmap\ p \\
pany\ p & = & por \circ pmap\ p \\
pelem\ x & = & pany\ (\lambda y \to x == y)
\end{array}
$$

A linear variant of *flatten* can be defined by using an accumulating parameter:

$$
\begin{array}{lcl}
flatten' & :: & Regular\ d\ \Rightarrow\ d\ a \to [\,a\,] \to [\,a\,] \\
flatten' & = & comp \circ pmap\ (:)
\end{array}
$$

A.13 *ConstructorName*

module *ConstructorName* **where**

Functions *datatypeName* and *fconstructorName* are built in.

$$
\begin{array}{l}
datatypeName :: Regular\ d\ \Rightarrow\ d\ a \to String \\
fconstructorName :: Bifunctor\ f\ \Rightarrow\ f\ a\ b \to String \\
constructorName :: Regular\ d\ \Rightarrow\ d\ a \to String \\
constructorName\ =\ fconstructorName \circ out
\end{array}
$$

$$
\begin{aligned}
&\mathit{constructorNames} :: \mathit{Regular}\ d \Rightarrow d\ a \rightarrow [\mathit{String}]\\
&\mathit{constructorNames}\ =\ \mathit{fconstructorNames} \circ \mathit{out}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fconstructorNames} :: \mathit{Bifunctor}\ f \Rightarrow f\ a\ b \rightarrow [\mathit{String}]\\
&\mathit{fconstructorNames}\ x\ =\\
&\quad \mathit{map}\ \mathit{fconstructorName}\ (\mathit{fconstructors}\ \text{`}\mathit{asTypeOf}\text{`}\ [x])
\end{aligned}
$$

The use of *asTypeOf* is a way to propagate type information to the correct destination. It is used to work around the lack of explicit functor arguments.

$$
\begin{aligned}
&\mathit{constructorNamesAndArities} :: \mathit{Regular}\ d \Rightarrow d\ a \rightarrow [(\mathit{String}, \mathit{Int})]\\
&\mathit{constructorNamesAndArities}\ =\ \mathit{fconstructorNamesAndArities} \circ \mathit{out}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fconstructorNamesAndArities} :: \mathit{Bifunctor}\ f \Rightarrow f\ a\ b \rightarrow [(\mathit{String}, \mathit{Int})]\\
&\mathit{fconstructorNamesAndArities}\ x\ =\\
&\quad \mathit{map}\ (\mathit{mapFst}\ \mathit{fconstructorName})\\
&\qquad\quad (\mathit{fconstructorsAndArities}\ \text{`}\mathit{asTypeOf}\text{`}\ [(x, \bot)])
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{constructors} :: \mathit{Regular}\ d \Rightarrow [d\ a]\\
&\mathit{constructors}\ =\ \mathit{map}\ \mathit{inn}\ \mathit{fconstructors}\\
&\textbf{polytypic}\ \mathit{fconstructors} :: [f\ a\ b]\ =\\
&\quad \textbf{case}\ f\ \textbf{of}\\
&\qquad g + h \quad \rightarrow \quad \mathit{map}\ \mathit{Left}\ \mathit{fconstructors} +\!\!+\ \mathit{map}\ \mathit{Right}\ \mathit{fconstructors}\\
&\qquad g \qquad\ \ \rightarrow \quad [\bot]
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{polytypic}\ \mathit{fconstructorsAndArities} :: [(f\ a\ b, \mathit{Int})]\ =\\
&\quad \textbf{case}\ f\ \textbf{of}\\
&\qquad g + h \quad \rightarrow \quad \mathit{map}\ (\mathit{mapFst}\ \mathit{Left})\ \mathit{fconstructorsAndArities} +\!\!+\\
&\qquad\qquad\qquad\quad\ \ \mathit{map}\ (\mathit{mapFst}\ \mathit{Right})\ \mathit{fconstructorsAndArities}\\
&\qquad g \qquad\ \ \rightarrow \quad (\lambda x \rightarrow [(x, \mathit{fconstructorArity}\ x)])\ \bot
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{polytypic}\ \mathit{fconstructorArity} :: f\ a\ b \rightarrow \mathit{Int}\ =\\
&\quad \textbf{case}\ f\ \textbf{of}\\
&\qquad g * h \qquad \rightarrow \quad \lambda p \rightarrow \mathit{fconstructorArity}\ (\mathit{fst}\ p) +\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{fconstructorArity}\ (\mathit{snd}\ p)\\
&\qquad \mathit{Empty} \quad \rightarrow \quad \mathit{const}\ 0\\
&\qquad f \qquad\quad\ \ \rightarrow \quad \mathit{const}\ 1
\end{aligned}
$$

$$
\begin{array}{l}
\mathit{constructor2Int} :: \mathit{Regular}\ d \Rightarrow d\ a \rightarrow \mathit{Int} \\
\mathit{constructor2Int} \ = \ \mathit{fconstructor2Int} \circ \mathit{out} \\
\textbf{polytypic}\ \mathit{fconstructor2Int} :: f\ a\ b \rightarrow \mathit{Int} \ = \\
\quad \textbf{case}\ f\ \textbf{of} \\
\qquad g + h \quad \rightarrow \quad \mathit{const}\ 0 \ \triangledown \ ((\lambda n \rightarrow 1 + n) \circ \mathit{fconstructor2Int}) \\
\qquad g \qquad\ \ \rightarrow \quad \mathit{const}\ 0
\end{array}
$$

$$
\begin{array}{l}
\mathit{int2constructor} :: \mathit{Regular}\ d \Rightarrow \mathit{Int} \rightarrow d\ a \\
\mathit{int2constructor}\ n \ = \ \mathit{constructors}\ !!\ n
\end{array}
$$

$$
\begin{array}{l}
\mathit{int2fconstructor} :: \mathit{Bifunctor}\ f \Rightarrow \mathit{Int} \rightarrow f\ a\ b \\
\mathit{int2fconstructor}\ n \ = \ \mathit{fconstructors}\ !!\ n
\end{array}
$$

$$
\begin{array}{l}
\mathit{mapFst} :: (a \rightarrow b) \rightarrow (a, c) \rightarrow (b, c) \\
\mathit{mapFst}\ f\ p \ = \ (f\ (\mathit{fst}\ p), \mathit{snd}\ p)
\end{array}
$$