Functional Polytypic Programming
Use and Implementation

Patrik Jansson

Department of Computing Science

1997

Functional Polytypic Programming
Use and Implementation

Patrik Jansson

Department of Computing Science
Chalmers University of Technology
Göteborg, Sweden
E-mail: `,,patrikj,@cs,.chalmers,.se,`

May 9, 1997

A dissertation for the Licentiate Degree in Computing Science at
Chalmers University of Technology

Department of Computing Science
S-412 96 Göteborg

ISBN 91-7197 486-5
CTH, Göteborg, 1997

Abstract

Many functions have to be written over and over again for different datatypes, either because datatypes change during the development of programs, or because functions with similar functionality are needed on different datatypes. Examples of such functions are pretty printers, pattern matchers, equality functions, unifiers, rewriting functions, etc. Such functions are called polytypic functions.

A polytypic function is a function that is defined by induction on the structure of user-defined datatypes. This thesis introduces polytypic functions, shows how to construct and reason about polytypic functions and describes the implementation of the polytypic programming system PolyP.

PolyP extends a functional language (a subset of Haskell) with a construct for writing polytypic functions. The extended language type checks definitions of polytypic functions, and infers the types of all other expressions. Programs in the extended language are translated to Haskell.

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

This licentiate thesis is based on the work presented in the following papers:

1. P. Jansson. Polytypism and polytypic unification. Master's thesis, Computing Science, Chalmers University of Technology and University of Göteborg, 1995. Available from `http://www.cs.chalmers.se/~patrikj/papers/masters/masters.html`.

2. P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

3. J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming, Second International School*, pages 68–114. Springer-Verlag, 1996. LNCS 1129.

4. P. Jansson and J. Jeuring. Polytypic unification. Submitted for publication. Available from `http://www.cs.chalmers.se/~patrikj/poly/unify2.ps`, 1997.

Chapter 2 is an extended version of the second paper, chapter 3 is a revised version of the third paper and chapter 4 is the fourth paper.

Acknowledgements

Special thanks to my supervisor Johan Jeuring for introducing me to the field of polytypic programming and for countless discussions during the last two years. I also thank the rest of my PhD-student committee, Lennart Augustsson and Thierry Coquand, and my professor John Hughes for reading and commenting on previous versions of this thesis. I thank the Multi group here at Chalmers for providing an inspiring research environment, and finally I thank Tünde Fülöp, the girl of my life, for her constant support.

Contents

Chapter 1

Introduction

The ability to name and reuse different program components is at the heart of the power of functional languages. Higher order functions like `map` and `foldr` 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.

A polymorphic function has a type parametrised on types. The next step is to parametrise a function definition on types. Functions that are parametrised in this way are called *polytypic functions* [44]. The equality function in ML and all the "derived" functions in Haskell are examples of polytypic functions. Other examples are the Squiggol community's [2, 9, 27, 59, 63, 65, 67] catamorphisms and maps. The functions `foldr` and `map` are instances of the polytypic functions `cata` and `map` for the datatype of lists.

While a normal polymorphic function is an algorithm that is independent of the type parameters, a polytypic function is a class of algorithms with related structure. Any algorithm in the class can be obtained by instantiating a template algorithm with (the structure of) a datatype. The possibility to define polytypic functions adds another level of flexibility in the reusability of algorithms and in the design of general purpose programming libraries.

Other terms used for polytypism in the literature are *structural polymorphism* (Ruehr [77]), *type parametric programming* (Sheard [78]), *generic programming* (Bird, de Moor and Hoogendijk [9]), *polynomial polymorphism* (Jay [41]) and *shape polymorphism* (Jay [42]) .

1.1 What is polytypism?

To give an example of what polytypism is we show that the definitions of the function `sum` on different datatypes share a common structure. The normal `sum` function for lists can be defined as follows in the functional language Haskell:

```
sum :: List Int -> Int
```

```
sum  Nil          =  0
sum  (Cons x xs)  =  x + sum xs
```

We will define `sum` on the following datatypes:

```
data List a   = Nil | Cons a (List a)
data Tree a   = Leaf a | Bin (Tree a) (Tree a)
data Maybe a  = Nothing | Just a
data Rose a   = Fork a (List (Rose a))
```

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 $\mathtt{cata}_D$ $\{$ `Ci` $\mapsto$ `ei` $\}$ for the catamorphism on the type `D` that replaces the constructors `Ci` with the expressions `ei`.

$$
\begin{array}{lll}
\mathtt{sum}_{List} & :: & \mathtt{List\ Int\ \ ->\ Int} \\
\mathtt{sum}_{List} & = & \mathtt{cata}_{List}\ \ \{\ \mathtt{Nil} \mapsto \mathtt{0,\ Cons} \mapsto \mathtt{(+)}\ \} \\
\\
\mathtt{sum}_{Tree} & :: & \mathtt{Tree\ Int\ \ ->\ Int} \\
\mathtt{sum}_{Tree} & = & \mathtt{cata}_{Tree}\ \ \{\ \mathtt{Leaf} \mapsto \mathtt{id,\ Bin} \mapsto \mathtt{(+)}\ \} \\
\\
\mathtt{sum}_{Maybe} & :: & \mathtt{Maybe\ Int\ ->\ Int} \\
\mathtt{sum}_{Maybe} & = & \mathtt{cata}_{Maybe}\ \ \{\ \mathtt{Nothing} \mapsto \mathtt{0,\ Just} \mapsto \mathtt{id}\ \} \\
\\
\mathtt{sum}_{Rose} & :: & \mathtt{Rose\ Int\ \ ->\ Int} \\
\mathtt{sum}_{Rose} & = & \mathtt{cata}_{Rose}\ \ \{\ \mathtt{Fork} \mapsto \backslash\mathtt{a\ l->\ a\ +\ sum}_{List}\ \mathtt{l}\ \}
\end{array}
$$

All these definitions are instances of the polytypic definition:

$$
\boxed{
\begin{array}{lll}
\mathtt{sum}_d & :: & \mathtt{d\ Int\ ->\ Int} \\
\mathtt{sum}_d & = & \mathtt{cata}_d\ \mathtt{fsum}_d
\end{array}
}
$$

where $\mathtt{fsum}_d$ is defined by induction on the structure of the datatype `d a`. We can already see some patterns in the parameters of the catamorphism: the two nullary constructors `Nil` 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 (`Cons`, `Bin` and `Fork`) are replaced by functions that sum the subexpressions. The complete definition of the polytypic `sum` function can be found in appendix B.2.

Exactly the same structure can be used to define the polytypic function `conc` (a variant of `flatten` defined in figure 2.1) if we replace `0` by `Nil` and `(+)` with list concatenation (`(++)`) in the above definitions.[1]

[1] If we parametrise one step further we obtain `crush :: (a->a->a) -> a -> d a -> a`, where `sum = crush (+) 0` and `conc = crush (++) Nil`. See Meertens [64] and appendix B.4 for definitions.

```
conc_d  ::  d (List a) -> List a
conc_d  =   cata_d fconc_d
```

The polymorphism of a polytypic function such as `sum` is somewhere in between parametric and *ad hoc* polymorphism (overloading). A parametric polymorphic function has one definition and all instances have the same structure. An *ad hoc* polymorphic function has many different (possibly unrelated) definitions and different instances are used in different contexts.

For a polytypic function like `sum` one definition suffices, but like *ad hoc* polymorphic functions, `sum` has different instances in different contexts. The compiler generates the instances from the definition of the polytypic function and the type in the context where it is used. A polytypic function can be parametric polymorphic, but it need not be: function `sum :: d Int -> Int` is polytypic but not parametric polymorphic while function `conc :: d (List a) -> List a` is both parametric polymorphic and polytypic.

1.2 Why polytypism?

Polytypism 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. Thus polytypic functions are very well suited for building program libraries.

Stability: Polytypic programs often work without change when a datatype is changed. This reduces the need for time consuming and boring rewrites of trivial functions.

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: data compression (section 2.2.6), term rewriting (section 3.4), unification (chapter 4), ...

Provability: More general functions give more general proofs. If we consider polytypic proofs, each of the benefits above obtains an additional interpretation: we get reusable proofs, stable proofs, less *ad hoc* semantics of programming languages and possibly new proofs of properties of rewriting, unification, etc.

Lambert Meertens [64] gives a nice example of the power of polymorphism: Suppose we want a function to swap two integers: `swap :: (Int, Int) -> (Int, Int)`. This is not a very hard problem to solve, but there are infinitely many type correct but wrong solutions. (Two are `id` and `\(x,y)->(y+1,x)`.) If we generalise this function to the polymorphic function `swap :: (a,b) -> (b,a)` we get a much more useful program and we *can't* make it wrong while type correct.[2] 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. Anyone who doubts this is encouraged to try to write a catamorphism for the (mutually recursive group of datatypes representing the) abstract syntax of Haskell!

1.3 Overview

The main part of this thesis consists of three papers about polytypism (with a fair amount of overlap between them), one chapter about our implementation of polytypism (called PolyP), and one chapter about related work.

Chapter 2 is a slightly extended version of the POPL'97 paper *PolyP — a polytypic programming language extension* [39]. This chapter gives an overview of what polytypism is and how polytypic functions can be expressed and implemented in a type safe way. If you have time to read only one chapter, this should be the one! The chapter describes the main features of PolyP: The `polytypic` construct which is used for defining polytypic functions by induction over the structure of user-defined datatypes, the type system that preserves type inference provided the `polytypic` construct is explicitly typed, and the translation of PolyP-programs into Haskell.

Chapter 3 is a revised version of the paper *Polytypic Programming* [46]. This chapter is the one you should read if you want to learn how to write and use polytypic functions: it defines catamorphisms, the polytypic `map` function, the polytypic `sum` function used in the example above and a number of other polytypic functions. A larger example, polytypic term rewriting, is also included.

Chapter 4 is the paper *Polytypic unification* [40] (submitted for publication). It is a short self-contained paper that shows how to write a polytypic unification algorithm in PolyP.

Chapter 5 describes the implementation of PolyP. Read this if you want to know how polytypic functions can be implemented and how the system PolyP really works.

Chapter 6 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.

Appendix A shows the translation of a PolyP program, appendix B shows the code for a small library of polytypic functions, and appendix C gives an example of simulating polytypism in Gofer.

---

[2] Strictly speaking this is true only in a language with strong normalisation.

Chapter 2

PolyP — a polytypic programming language extension

PolyP — a polytypic programming language extension[1]

Patrik Jansson and Johan Jeuring
Chalmers University of Technology and University of Göteborg
S-412 96 Göteborg, Sweden
{patrikj,johanj}@cs.chalmers.se
http://www.cse.chalmers.se/~{patrikj,johanj}/

Abstract

Many functions have to be written over and over again for different datatypes, either because datatypes change during the development of programs, or because functions with similar functionality are needed on different datatypes. Examples of such functions are pretty printers, equality functions, unifiers, pattern matchers, rewriting functions, etc. Such functions are called polytypic functions. A polytypic function is a function that is defined by induction on the structure of user-defined datatypes. This paper extends a functional language (a subset of Haskell) with a construct for writing polytypic functions. The extended language type checks definitions of polytypic functions, and infers the types of all other expressions using an extension of Jones' theories of qualified types and higher-order polymorphism. The semantics of the programs in the extended language is obtained by adding type arguments to functions in a dictionary passing style. Programs in the extended language are translated to Haskell.

[1]This is an extended version of a paper with the same title published in the 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, January 15–17, 1997.

2.1 Introduction

Complex software systems usually contain many datatypes, which during the development of the system change regularly. Developing innovative and complex software is typically an evolutionary process. Furthermore, such systems contain functions that have the same functionality on different datatypes, such as equality functions, print functions, parse functions, etc. Software should be written such that the impact of changes to the software is as limited as possible. Polytypic programs are programs that adapt automatically to changing structure, and thus reduce the impact of changes. This effect is achieved by writing programs such that they work for large classes of datatypes.

Consider for example the function `length :: List a -> Int`, which counts the number of values of type `a` in a list. There is a very similar function `length :: Tree a -> Int`, which counts the number of occurrences of `a`'s in a tree. We now want to generalise these two functions into a single function which is not only polymorphic in `a`, but also in the type constructor; something like `length :: d a -> Int`, where `d` ranges over type constructors. We call such functions *polytypic functions* [44]. Once we have a polytypic `length` function, function `length` can be applied to values of any datatype. If a datatype is changed, `length` still behaves as expected. For example, the datatype `List a` has two constructors with which lists can be built: the empty list constructor, and the `Cons` constructor, which prepends an element to a list. If we add a constructor with which we can append an element to a list, function `length` still behaves as expected, and counts the number of elements in a list.

The equality function in ML and Ada and the functions in the classes that can be derived in Haskell are examples of widely used polytypic functions. Instances of these functions are automatically generated by the compiler, but the definitions of these functions cannot be given in the languages themselves. In this paper we investigate a language extension with which such functions can be defined in the language. Polytypic functions are useful in many situations; more examples are given in Jeuring and Jansson [46].

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. A datatype `D a` is regular if it 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 all conventional recursive datatypes, such as `Nat`, `List a`, and different kinds of trees. We introduce the class `Regular` of all regular datatypes, and we write: `length :: Regular d => d a -> Int`

Polytypic functions can be defined on a larger class of datatypes, including datatypes with function spaces [29, 66], but we will not discuss these extensions.

2.1.1 Polymorphism and polytypism

Polytypism differs from both parametric polymorphism and *ad hoc* polymorphism (overloading). This subsection explains how.

A traditional polymorphic function such as

```
head  ::  [a] -> a
```

can be seen as a family of functions - one for each instance of `a` as a monomorphic type. There need only be one definition of `head`; the typing rules ensure that values of type `a` are only moved around, never tampered with. A polymorphic function can be implemented as a single function that works on boxed values.

An *ad hoc* polymorphic function such as

```
(+)  ::  Num a => a -> a -> 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. Helped by type inference a compiler can almost always find the correct instance.

The polymorphism of a polytypic function such as

```
length  ::  Regular d => d a -> Int
```

is somewhere in between parametric and *ad hoc* polymorphism. A single definition of `length` suffices, but `length` 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 be parametric polymorphic, but it need not be: function `sum :: Regular d => d Int -> Int`, which returns the sum of the integers in a value of an arbitrary datatype, is polytypic, but not parametric polymorphic.

2.1.2 Writing polytypic programs

There exist various ways to implement polytypic programs in a typed language. Three possibilities are:

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

Polytypic functions can be written 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 by providing translation functions to and from the universal datatype. However, using universal datatypes has several disadvantages: the user has to write all the translation functions, 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 [49], type information is preserved, and we can use current functional languages such as Gofer and Haskell for implementing polytypic functions. In this style all datatypes are represented by the type

```
data Mu f a = In (f a (Mu f a))
```

and the class system is used to overload functions like `map` and `cata`. See appendix C for some code examples. 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. Furthermore, it is hard to deal with mutual recursive datatypes.

Since 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. Thus polytypic functions can be implemented and type checked. We will use the name PolyP both for the extension and the resulting language. Consult the page

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

to obtain a preliminary version of a compiler that compiles PolyP into Haskell (which subsequently can be compiled with a Haskell compiler), and for the latest developments on PolyP.

2.1.3 PolyP

PolyP is an extension of a functional language that allows the programmer to define and use polytypic functions. The underlying language in this article 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. Since 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[1]. The type inference algorithm is based upon Jones' theories of qualified types [48] and higher-order polymorphism [51]. 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 completely remove the dictionary values at

[1]Just as in Haskell, sometimes explicit type annotations are needed to resolve overloading.

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 non mutual recursive, regular datatypes with one type argument. Multiple type arguments can be encoded in a single sum-type, but we are working on a more elegant treatment of multiple type arguments. One of PolyP's predecessors (a preprocessor that generated instances of polytypic functions [38]) could handle mutual recursive datatypes, and we hope to port this part of the predecessor to PolyP in the near future. In the future PolyP will be able to handle mutual recursive datatypes with an arbitrary number of type arguments and in which function spaces may occur.

2.1.4 Background and related work

Polytypic functions are standard in the Squiggol community, see [59, 63, 65]. Generating instances for specific polytypic functions, such as `(==)`, `map`, `cata`, `hylo`, etc. for a given type, is rather simple and has been demonstrated by several authors [13, 21, 38, 41, 81].

Given a number of predefined polytypic functions many others can be defined, and amongst others Jay *et al.*'s type system [7, 41], and Jones' type system based on qualified types and higher-order polymorphism [48, 51] can be used to type check expressions in a language with predefined polytypic functions. Our approach differs from these approaches in that we only give two predefined polytypic functions, and we supply a construct to define new polytypic functions by induction over the structure of datatype definitions. This difference is essential for polytypic programming, and can be compared with the difference between the first versions of ML that gave a number of predefined datatypes and the later versions of ML that provided a few built in types and a construct for defining user-defined datatypes.

Using a two level language, Sheard and Nelson [80] show how to write well-typed polytypic programs. A polytypic program is obtained by embedding second level type declarations as values in first level computations. The two level language is more powerful than our language, but it is also a much larger extension of common functional programming languages.

Adaptive object-oriented programming [57, 73] is a programming style similar to polytypic programming. In adaptive OOP methods (corresponding to our polytypic functions) are attached to groups of classes (types) that usually satisfy certain constraints (such as being regular). In adaptive OOP one abstracts from constructor names instead of datatype structure. This results in a programming style in which typing plays a much less prominent role than in polytypic programming. However, the resulting programs have very similar behaviour.

2.1.5 About this paper

We use Haskell syntax for programs and types in our examples but we sometimes write `b <- a` as syntactic sugar for `a -> b` in types (and kinds).

Section 2.2 introduces polytypic programming. Section 2.3 discusses the type inference and checking algorithms used in PolyP. Section 2.4 gives the semantics of PolyP, and Section 2.5 shows how to generate code for PolyP programs. Section 2.6 concludes the paper.

2.2 Polytypic programming

In this section we will show how to write polytypic programs using PolyP. For an extensive introduction to polytypic programming see Jeuring and Jansson [46].

2.2.1 Functors for datatypes

To define a polytypic function, we have to be able to define functions by induction over the structure of a datatype. The structure of a datatype is described by means of the *functor* defining the datatype.

Consider the datatype `List a` defined by

```
data List a = Nil | Cons a (List a)
```

Values of this datatype are built by prepending values of type `a` to a list. This datatype can be viewed as the fixed point with respect to the second argument of the datatype `FList a x` defined by

```
data FList a x = FNil | FCons a x
```

The datatype `FList a x` describes the structure of the datatype `List a`. Since we are only interested in the structure of `List a`, the names of the constructors of `FList` are not important. We define *FList* using a conventional notation by removing `FList`'s constructors (writing *Empty* for the empty space we obtain by removing `FNil`), replacing `|` with $+$, and replacing juxtaposition with $\times$.

$$\mathit{FList}\ a\ x \quad = \quad \mathit{Empty} + a \times x$$

We now abstract from the arguments a and x in *FList*. Constructor *Par* returns the parameter a (the first argument), and *Rec* returns the recursive parameter x (the second argument). Operators $+$ and $\times$ and the empty product *Empty* are lifted.

$$\mathit{FList} \quad = \quad \mathit{Empty} + \mathit{Par} \times \mathit{Rec}$$

FList is the functor[2] of `List a`.

The datatype `Tree a` is defined by

[2] In fact, *FList* is a bifunctor: a functor that takes two arguments; we will use both terms functor and bifunctor for bifunctors.

```
data Tree a = Leaf a | Bin (Tree a) (Tree a)
```

Applying the same procedure as for the datatype `List a`, we obtain the following definition.

$$FTree \;=\; Par + Rec \times Rec$$

$FTree$ is the functor of `Tree a`.

We have given functors that describe the structure of the datatypes `List a` and `Tree a`. We have that for each regular datatype there exists a (bi)functor F that describes the structure of the datatype[3].

For our purposes, a functor is a value generated by the following grammar.

$$F ::= f \mid F + F \mid F \times F \mid Empty \mid Par \mid Rec \mid D@F \mid Const\ \tau$$

where f is a functor variable, D generates datatype constructors and τ in $Const\ \tau$ is a type. The alternative $D@F$ is used to describe the structure of types that are defined in terms of other user-defined types, such as the datatype of *rose-trees*:

```
data Rose a = Fork a (List (Rose a))
```

The functor we obtain for this datatype is

$$FRose \;=\; Par \times (List\ @Rec)$$

The alternative $Const\ \tau$ in the grammar is used in the description of the structure of a datatype that contains constant types such as `Bool`, `Char`, etc.

2.2.2 The `polytypic` construct

We introduce a new construct `polytypic` for defining polytypic functions by induction on the structure of a functor:

```
polytypic p :: t = case f of {fi -> ei}
```

where `p` is the name of the value being defined, `t` is its type, `f` is a functor variable, `fi` are functor patterns and `ei` are PolyP expressions. The explicit type in the `polytypic` construct is needed since we cannot in general infer the type from the cases.

The informal meaning is that we define a function that takes a functor (a value describing the structure of a datatype) as its first argument. This function selects the expression in the first branch of the case matching the functor. Thus the polytypic construct is a template for constructing instances of polytypic functions given the 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.

```
flatten  :: Regular d => d a -> [a]
flatten  =  cata fl

polytypic fl :: f a [a] -> [a] =
  case f of
    g + h   -> either fl fl
    g * h   -> \(x,y) -> fl x ++ fl y
    Empty   -> \x -> []
    Par     -> \x -> [x]
    Rec     -> \x -> x
    d @ g   -> concat . flatten . pmap fl
    Const t -> \x -> []

data Either a b  =  Left a | Right b

either :: (a->c) -> (b->c) -> Either a b -> c
either f g (Left x)   =  f x
either f g (Right x)  =  g x
```

Figure 2.1: The definition of `flatten`

As a running example throughout the paper we take the function `flatten` defined in figure 2.1. When `flatten` is used on an element of type `Tree a`, the compiler performs roughly the following rewrite steps to construct the actual instance of `flatten` for `Tree`:

$$\texttt{flatten}_{Tree} \quad \rightarrow \quad \texttt{cata}_{Tree}\ \texttt{fl}_{FTree}$$

It follows that we need an instance of `cata` on the datatype `Tree a`, and an instance of function `fl` on the functor of `Tree a`. For the latter instance, we use the definition of *FTree* and the definition of `fl` to transform $\texttt{fl}_{FTree}$ as follows.

$$\texttt{fl}_{FTree} \rightarrow \texttt{fl}_{Par+Rec\times Rec} \rightarrow \texttt{either}\ \texttt{fl}_{Par}\ \texttt{fl}_{Rec\times Rec}$$

We transform the functions $\texttt{fl}_{Par}$ and $\texttt{fl}_{Rec\times Rec}$ separately. For $\texttt{fl}_{Par}$ we have

$$\texttt{fl}_{Par} \quad \rightarrow \quad \texttt{\\x -> [x]}$$

and for $\texttt{fl}_{Rec\times Rec}$ we have

$$\begin{array}{ll} & \texttt{fl}_{Rec\times Rec} \\ \rightarrow & \texttt{\\(x,y) -> fl}_{Rec}\ \texttt{x ++ fl}_{Rec}\ \texttt{y} \\ \rightarrow & \texttt{\\(x,y) -> (\\x -> x) x ++ (\\x -> x) y} \end{array}$$

[3] A datatype can be modelled as the initial algebra in the category of F a-algebras [59], where F is the the functor of the datatype.

The last function can be rewritten into `uncurry (++)`, and thus we obtain the following function for flattening a tree:

$$\texttt{cata}_{Tree}\ \texttt{(either (\textbackslash x -> [x]) (uncurry (++)))}$$

By expanding $\texttt{cata}_{Tree}$ in a similar way we obtain a Haskell function for the instance of `flatten` on `Tree`.

The catamorphism, or generalised fold, on a datatype takes as many functions as the datatype has constructors (combined into a single argument by means of function `either`), and recursively replaces constructor functions with corresponding argument functions. It is a generalisation to arbitrary regular datatypes of function `foldr` defined on lists. We will give the definition of `cata` in the next subsection.

2.2.3 Basic polytypic functions

In the definition of function `flatten` we used functions like `cata` and `pmap`. This subsection defines these and other basic polytypic functions.

Since polytypic functions cannot refer to constructor names of specific datatypes, we introduce the predefined functions `out` and `inn`. Function `out` is used in polytypic functions instead of pattern matching on the constructors of a datatype. For example `out` on `Tree` is defined as follows:

$$\begin{array}{lll} \texttt{out}_{Tree}\ \texttt{(Leaf x)} & = & \texttt{Left x} \\ \texttt{out}_{Tree}\ \texttt{(Bin l r)} & = & \texttt{Right (l,r)} \end{array}$$

Function `inn` is the inverse of function `out`. It collects the constructors of a datatype into a single constructor function.

```
out :: Regular d => d a -> (FunctorOf d) a (d a)
inn :: Regular d => d a <- (FunctorOf d) a (d a)
```

The constructor `FunctorOf` is a special type constructor that takes a datatype constructor, and returns its functor. With datatypes as fix-points of functors, `FunctorOf` is the 'unfix'. `FunctorOf` is our main means for expressing the relation between datatypes and functors.

In category theory, a functor is a mapping between categories that preserves the algebraic structure of the category. Since a category consists of objects (types) and arrows (functions), a functor consists of two parts: a definition on types, and a definition on functions. The functors we have seen until now are functions that take two types and return a type. The part of the functor that takes two functions and returns a function is called `fmap`, see figure 2.2.

Using `fmap` we can define the polytypic version of function map, `pmap`, as follows:

```
polytypic fmap :: (a -> c) -> (b -> d) -> f a b -> f c d
  = \p r -> case f of
              g + h   -> fmap p r -+- fmap p r
              g * h   -> fmap p r -*- fmap p r
              Empty   -> id
              Par     -> p
              Rec     -> r
              d @ g   -> pmap (fmap p r)
              Const t -> id

f -+- g = either (Left . f) (Right . g)
(f -*- g) (x,y)  =  (f x , g y)
```

Figure 2.2: Definition of `fmap`.

```
pmap :: Regular d => (a -> b) -> d a -> d b
pmap f  =  inn . fmap f (pmap f) . out
```

where `out` takes the argument apart, `fmap` applies `f` to parameters and `(pmap f)` recursively to substructures and `inn` puts the parts back together again.

Function `cata` is also defined in terms of function `fmap`:

```
cata :: Regular d => (FunctorOf d a b -> b) -> (d a -> b)
cata f = f . fmap id (cata f) . out
```

This one-liner, together with the definition of `fmap` is all that is needed to obtain a catamorphism for every regular datatype.

2.2.4 Catamorphisms on specific datatypes

Since catamorphisms are not only useful when defining polytypic functions, but also when defining functions on specific datatypes, we provide a shorthand notation for creating the function argument to `cata`: {`ci -> ei`}. As an example, consider the following datatype of simple expressions.

```
data Expr a  =  Con a
             |  Add (Expr a) (Expr a)
             |  Mul (Expr a) (Expr a)
```

Function `eval` evaluates an expression:

```
eval  :: Num a => Expr a -> a
eval  =  cata { Con -> id
                Add -> (+)
                Mul -> (*) }
```

Evaluating `eval expr` for some `expr :: Expr a` will result in replacing each constructor in `expr` with its corresponding function.

2.2.5 More polytypic functions

We can define a polytypic equality function using a polytypic `zip` function:[4]

```
(==) :: (Regular d,Eq a) => d a -> d a -> Bool
a == b = maybe False
               (all (uncurry (==)) . flatten)
               (pzip (a,b))

pzip :: Regular d =>   (d a  ,d b  ) -> Maybe (d (a,b))
fzip :: Bifunctor f => (f a b,f c d) -> Maybe (f (a,c) (b,d))
```

where `maybe`, `all` and `uncurry` are predefined Haskell functions. Function `pzip` is a generalisation of the Haskell function `zip :: [a] -> [b] -> [(a,b)]`. Function `zip` takes a pair of lists to a list of pairs. If the lists are of unequal length (that is their structures are different) the longer list is truncated (replaced by the empty structure). In `pzip` a pair of structures is mapped to `Just` a structure of pairs if the structures are equal, and `Nothing` otherwise, since it is in general impossible to know what 'truncate' or an 'empty structure' means for a type `d a`.

Function `pzip` is defined using the non-recursive variant `fzip`, which is defined by means of the `polytypic` construct. The evaluation of `a == b` gives `False` if `pzip (a,b)` gives `Nothing` and checks that all pairs in the zipped structure are equal otherwise.

In the next subsection we will use function `separate` which separates a value into its structure and its contents.

```
separate :: Regular d => d a -> (d (),[a])
separate x = (pmap (const ()) x, flatten x)
```

Function `separate` is the central function in Jay's [42] representation of values of shapely types: a value of a shapely type is represented by its structure, obtained by replacing all contents of the value with `()`, and its contents, obtained by flattening the value.

2.2.6 Polytypic data compression

A considerable amount of Internet traffic consists of files that possess structure — examples are databases, html files, and Java-Script programs — and it will pay to compress these structured files. Structure-specific compression methods give

[4] This definition has other strictness properties than Haskell's `zip`. See section 3.3.4 for an alternative definition.

much better compression results than conventional compression methods such as the Unix compress utility [6, 87, 88]. For example, Unix compress typically requires four bits per byte of Pascal program code, whereas Cameron [14] reports compression results of one bit per byte Pascal program code.

The basic idea of the structure-specific compression methods is simple: parse the input file into a structured value, separate structure from contents, compress the structure into a bit-string by representing constructors by numbers, and compress the resulting string and the contents with a conventional compression method. For example, suppose we have the following datatype for trees:

```
data Tree a = Leaf a | Bin (Tree a) (Tree a)
```

and a file containing the following tree `t`

```
Bin (Bin (Leaf "com") (Leaf "pre")) (Bin (Leaf "ssi")(Leaf "on"))
```

Separating structure from contents gives:

```
Bin (Bin (Leaf ())    (Leaf ()))    (Bin (Leaf ())   (Leaf ()))
```

and a list containing the four words `com`, `pre`, `ssi` and `on`. Assigning `0` to `Leaf` and `1` to `Bin` , the above structure can be represented by `1100100`. This bit-string equals 100 when read as a binary number, and hence this list can be represented by the 100'th ASCII character 'd'. So the tree `t` can be represented by the list of words `[d,com,pre,ssi,on]`. The tree `t` is stored in 65 bytes, and its compressed counterpart requires 18 bytes. This list can be further compressed using a conventional compression method.

Most authors of program code compression programs [14, 16, 18, 52, 82] observe that this method works for arbitrary structured objects, but most results are based on compressing Pascal programs. To compress Java-Script programs we will have to write a new compression program. It is desirable to have a polytypic data compression program.

The description of the basic idea behind polytypic data compression is translated into a polytypic program `pcompress` as follows. Function `pcompress` takes as argument a description (`concrete_syntax`) of how to print values of a datatype (the abstract syntax).

```
pcompress :: (Regular d,Text a) => Syntax d -> String -> String
pcompress concrete_syntax = ccompress
                          . structure_compress -*- show
                          . separate
                          . parse concrete_syntax
```

We will describe each of the new functions above in turn. We will omit the precise definitions of these functions.

- Function `parse` is a polytypic function of type

```
parse :: Regular d => Syntax d -> String -> d a
```

It takes a description of the concrete syntax of a datatype, and returns a parser for that concrete syntax. It is only defined if the grammar for the concrete syntax satisfies certain properties. For a definition, see Huisman [37] or Björk [12].

- Function `structure_compress` takes a structure, replaces its constructors by numbers, and turns the resulting structure into a string.

```
structure_compress :: d () -> String
```

- Function `show :: Text a => a -> String` prints the content list generated by `separate` and, finally,

- Function `ccompress` uses a conventional compression program to compress the pair of strings.

```
ccompress :: (String,String) -> String
```

2.3 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.

The first subsection introduces the core language without the `polytypic` construct, but with qualified and higher-order polymorphic types. The second subsection extends the core with PolyP in two steps. The third subsection discusses unification in the extended language, and the fourth subsection shows how to type check a polytypic value definition.

2.3.1 The core language

Our core language is an extension of core-ML with qualified types and higher order polymorphism [51], see figure 2.3. Each constructor in this language has a superscript denoting its kind. For example, a basic type has kind `*`, and a datatype constructor such as `List` has kind `* -> *`. We call the resulting language QML. The set of constructor constants contains:

```
->, (,), Either  ::  * -> * -> *
```

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

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

Figure 2.3: The core language QML

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

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

Figure 2.4: Some of the typing rules for QML

polymorphic types, see Jones [48, 51]. Compared to the traditional Hindley-Milner system the type judgements are extended with a set of predicates P. The rules involving essential changes in the predicate set are shown in figure 2.4. 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 [48].

2.3.2 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 generalising the unification algorithm and by adding new types, kinds and classes to the initial type environment. The initial typing environment of the language polyQML consists of four components: the typings of the functions `inn` and `out`, the type classes `Regular` and `Bifunctor`, two type constructors `FunctorOf` and `Mu`, and the collection of functor constructors (`+`, `*`, `@`, `Empty`, `Par`, `Rec` and `Const t`).

- Functions `inn` and `out` were introduced in section 2.2.3.

  ```
  out :: Regular d => d a -> (FunctorOf d) a (d a)
  inn :: Regular d => d a <- (FunctorOf d) a (d a)
  ```

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

- 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 (FunctorOf d)`

- `FunctorOf` is a type constructor that takes a datatype constructor and represents its functor. Type constructor `Mu` is the inverse of `FunctorOf`: it takes a functor, and represents the datatype that has the functor as structure. As there may be different datatypes with the same structure, we add a second argument of `Mu` to disambiguate types. The type constructor `Mu` is useful when we want to relate similar but different types. `FunctorOf` and `Mu` have the following kinds:

  ```
  FunctorOf :: 1 -> 2
  Mu        :: 1 <- (2,1)
  ```

  where `1` abbreviates the kind of regular type constructors (`*->*`) and `2` abbreviates the kind of bifunctors (`*->*->*`).

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

  ```
  * , +                    :: 2 -> 2 -> 2
  @                        :: 1 -> 2 -> 2
  Empty,Par,Rec,Const t    :: 2
  ```

  Each of these constructors has one rule in the entailment relation of one of the following forms:

  `Bifunctor f,Bifunctor g` $\Vdash$ `Bifunctor(f+g)`
  `Regular d, Bifunctor g` $\Vdash$ `Bifunctor (d@g)`
  $\Vdash$ `Bifunctor Par`

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 function 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 typing environment. This would give us roughly the same expressive power as the language given by Jay [41] extended with qualified types. As a minimal example we could add `fmap` to the initial environment:

```
fmap :: Bifunctor f => (a->b) -> (c->d) -> f a c -> f b d
```

letting us define and type check polytypic functions like `pmap` and `cata`. The type checking algorithm would for example derive `pmap (+1) (Leaf 4) :: Regular Tree => Tree 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 will make 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

$$\texttt{polytypic}\ x : \rho = \texttt{case}\ f^{*\to*\to*}\ \texttt{of}\ \{F_i \to E_i\}$$

where f is a functor variable[5], and F is the nonterminal that describes the language of functors defined in Section 2.2.1. The resulting language is polyQML. To be able to do the case analysis over a functor, it must be built up using 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` is always included in the type ρ of a function defined by the `polytypic` construct. (But it need not be given explicitly.)

The typing rules for polyQML are the rules from QML together with the rule for typing the `polytypic` construct given in figure 2.5. For the notation used, see [48]. 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 2.6.

2.3.3 Unification

The (omitted) typing rule for application uses a unification algorithm to unify the argument type of a function with the type of its argument. The presence of the equalities concerning `Mu` and `FunctorOf` complicate unification.

[5]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.

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

Figure 2.5: The typing rule for `polytypic`

```
type (g + h) a b  =  Either (g a b) (h a b)
type (g * h) a b  =  (g a b , h a b)
type Empty a b    =  ()
type Par a b      =  a
type Rec a b      =  b
type (d @ g) a b  =  d (g a b)
type Const t a b  =  t
```

Figure 2.6: Interpreting functors as type synonyms

The unification algorithm we use is an extension of the kind-preserving unification algorithm of Jones [51], which in its turn is an extension of Robinson's well-known unification algorithm. We unify under the equalities

$$\texttt{Mu (FunctorOf d,d)} = \texttt{d} \tag{2.1}$$

$$\texttt{FunctorOf (Mu (f,d))} = \texttt{f} \tag{2.2}$$

$$\texttt{Mu(}\mathcal{F}_{\texttt{D}}\texttt{,D)} = \texttt{Mu(FunctorOf D,D)} \tag{2.3}$$

where $\mathcal{F}_{\texttt{D}}$ is the functor corresponding to the datatype `D a` built with the functor constructors. The last equality represents a set of equalities: one such equality is generated for each regular datatype declared in the program. For example, if a program declares the datatype `List a`, the equality

```
Mu (Empty + Par*Rec, List) = Mu (FunctorOf List, List)
```

is generated.

We will write $C \sim^{\sigma} C'$ if C and C' are unified under equalities (2.1), (2.2), and (2.3) by substitution σ. For example, we have

$$\begin{array}{rcl} \texttt{Tree a} & \sim^{\sigma_1} & \texttt{Mu (f,d) a} \\ \texttt{Mu (f + g,d)} & \sim^{\sigma_2} & \texttt{Mu (FunctorOf List,List)} \end{array}$$

$$\text{where} \left\{ \begin{array}{rcl} \sigma_1 & = & \{\texttt{f} \mapsto \texttt{FunctorOf Tree}, \texttt{d} \mapsto \texttt{Tree}\} \\ \sigma_2 & = & \{\texttt{f} \mapsto \texttt{Empty}, \texttt{g} \mapsto \texttt{Par*Rec}, \texttt{d} \mapsto \texttt{List}\} \end{array} \right.$$

Unification under equalities is known as semantic unification, and is considerably more complicated than syntactic unification. In fact, for many sets of equalities it is impossible to construct a (most general) unifier. However, if we can turn the set of equalities under which we want to unify into a complete (normalising and confluent) set of rewriting rules, we can use one of the two algorithms (using narrowing or lazy term rewriting) from Martelli *et al.* [53, 61] to obtain a most general unifier for terms that are unifiable.

If we replace the equality symbol by $\rightarrow$ in our equalities, we obtain a complete set of rewriting rules. We use the recursive path orderings technique as developed by Dershowitz [19, 53] to prove that the rules are normalising, and we use the Knuth-Bendix completion procedure [53, 55] to prove that the rules are confluent. Both proofs are simple.

Theorem. If there is a unifier for two given types C, C', then $C \sim^{\sigma} C'$ using Jones [51] for kind-preserving unification and Martelli *et al.*'s [61] algorithm for semantic unification, and σ is a most general unifier for C and C'. Conversely, if no unifier exists, then the unification algorithm fails.

2.3.4 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 `polytypic` $x : \rho =$ `case` f `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 alternative 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 2.6. The complete type inference/checking algorithm $\mathcal{W}$ is obtained by extending Jones' type inference algorithm [51] with the alternative for the `polytypic` construct. Some of the rules of the algorithm are given in figure 2.7. As an example we will sketch how the definition of `fl` in figure 2.1 is type checked:

In the `g*h` branch of the polytypic case, we first infer the type of the expression $e_* =$ `\(x,y) -> fl x ++ fl y`. Using fresh instances of the explicit type $\rho =$ `f a [a] -> [a]` for the two occurrences of `fl` we get $\tau_* =$ `(x b [b] , y b [b]) ->`

(var)	$$\frac{x : \forall t_i.P \Rightarrow \tau \in \Gamma, \quad s_i \text{ new}, \quad S = \{t_i \mapsto s_i\}}{SP \mid \Gamma \vdash^w x : S\tau}$$
(let)	$$\frac{\begin{array}{c} S(\Gamma) \vdash^w q : \gamma \\ Q \mid T(S\Gamma, \gamma) \vdash^w e : \tau \end{array}}{Q \mid TS(\Gamma) \vdash^w \texttt{let}\ q\ \texttt{in}\ e : \tau}$$
(bind)	$$\frac{P \mid S(\Gamma) \vdash^w e : \tau, \quad \gamma = (x : \forall_{S\Gamma}(P \Rightarrow \tau))}{S(\Gamma) \vdash^w x = e : \gamma}$$
(poly)	$$\frac{\begin{array}{c} \Gamma' = (\Gamma, \gamma), \quad \gamma = (x : \forall_{\{\}}(\rho)) \\ P_i \mid S_i(T_{i-1}\Gamma') \vdash^w e_i : \tau_i \\ \forall_{T_n\Gamma'}(S_n \cdots S_{i+1}(P_i \Rightarrow \tau_i)) \geq \{f \mapsto f_i\}\rho \\ T_0 = \{\}, \quad T_i = S_i T_{i-1} \end{array}}{\Gamma \vdash^w \texttt{polytypic}\ x : \rho = \texttt{case}\ f\ \texttt{of}\ \{f_i \rightarrow e_i\} : \gamma}$$

Figure 2.7: Some parts of $\mathcal{W}$

`[b]`. We then calculate the type $\rho_* = \{\texttt{f} \mapsto \texttt{g*h}\}\rho =$ `(g*h) a [a] -> [a]` $=$ `(g a [a], h a [a]) -> [a]`. Since $\rho_* = \{\texttt{x} \mapsto \texttt{f}, \texttt{y} \mapsto \texttt{g}, \texttt{b} \mapsto \texttt{a}\}\tau_*$ 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 $e_{Rec} =$ `\x -> x`. The type of this expression is $\tau_{Rec} =$ `b -> b`. We then calculate the type $\rho_{Rec} = \{\texttt{f} \mapsto \texttt{Rec}\}\rho =$ `Rec a [a] -> [a]` $=$ `[a] -> [a]`. Since $\rho_{Rec} = \{\texttt{b} \mapsto \texttt{[a]}\}\tau_{Rec}$ 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, 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 [48] we obtain the following theorem.

Theorem. 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 some lemmas about substitutions and type ordering ($\geq$) together with the induction hypothesis we can show that the algorithm succeeds iff there is a derivation using the type rules.

2.4 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. For example, if the function `(==)` of type `∀a . Eq a => a -> a -> Bool` is used on integers somewhere, we need evidence for the fact `Eq Int`, meaning that `Int` has an equality. One way to give evidence in this case is simply to supply the function `primEqInt`. Again, the results from this section are heavily based on Jones' work on qualified types [48].

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

E	$::=$	$\cdots$	same as for QML expressions
	$\mid$	E_e	evidence application
	$\mid$	$\lambda v.E$	evidence abstraction
	$\mid$	$\texttt{case}\ v\ \texttt{of}\ \{e_i \to E_i\}$	dependent case over evidence
σ	$::=$	C^*	types
	$\mid$	$P \Rightarrow \sigma$	qualified types
	$\mid$	$\forall t_i^\kappa.\sigma$	polymorphic types

The special `case`-statement is used in the translation of the `polytypic` construct. The typing rules for QP are omitted.

The translation rules for variables, let expressions, variable bindings and for the the `polytypic` construct are given in figure 2.8. The remaining rules are simple and omitted. A translation rule of the form $P \mid S(\Gamma) \vdash^w e \leadsto e' : \tau$ 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 synthesised attributes (the output data) are the evidence context P, the substitution S, the translated QP expression e' and the inferred type τ.

For example, if we translate function `fl :: Bifunctor f => f a [a] -> [a]`, we obtain after simplification the code in figure 2.9. 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 $C : \sigma \to \sigma'$. The inputs to function $\geq$ are the two type schemes σ

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

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

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

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

Figure 2.8: Some translation rules

$$\begin{array}{lll} \texttt{fl} = & \lambda v.\texttt{case}\ v\ \texttt{of} & \\ & g + h & \to \ \texttt{either}\ \texttt{fl}_g\ \texttt{fl}_h \\ & g * h & \to \ \texttt{\\(x,y) -> fl}_g\ x\ \texttt{++ fl}_h\ y \\ & \mathit{Empty} & \to \ \texttt{\\x -> []} \\ & \mathit{Par} & \to \ \texttt{\\x -> [x]} \\ & \mathit{Rec} & \to \ \texttt{\\x -> x} \\ & d\ @\ g & \to \ \texttt{concat . flatten}_d\ \texttt{. pmap}_d\ \texttt{fl}_g \\ & \mathit{Const}\ t & \to \ \texttt{\\x -> []} \end{array}$$

Figure 2.9: The translation of function `fl` into QP

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 take a symbolic representation f of the functor (an element of the datatype described by nonterminal F from Section 2.2.1). So f : `Bifunctor f` for all `f` for which $\Vdash$ `Bifunctor f` holds. 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. 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' thesis for the other syntactic constructs.

2.5 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 [47]. 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.

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 $\mathcal{F}_{\mathtt{D}}$. PolyP constructs these dictionaries with a number of straightforward inductive functions over the abstract syntax of regular datatypes. Functions `inn` and `out` are now 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 (Hugs). In the appendix we have given an example PolyP program and the code that is generated for this program.

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, can be removed by a more clever implementation.

2.6 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. For example, for the purpose of multiple type arguments we will introduce a class `Functor n`, with `Regular` = `Functor 1`, and `Bifunctor` = `Functor 2`. The constructors `Mu` and `FunctorOf` have to be extended in a similar fashion.

The partial evaluation approach to code generation implies that we cannot compile a module containing a definition of a polytypic function separately from a module in which it is used. A solution might be to translate polytypic programs into a language with intensional polymorphism [32] instead of translating polytypic programs into QP.

Acknowledgements

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 3

Polytypic Programming

Polytypic Programming[1]

Johan Jeuring and Patrik Jansson
Chalmers University of Technology and University of Göteborg
S-412 96 Göteborg, Sweden
email: {johanj,patrikj}@cs.chalmers.se

Abstract

Many functions have to be written over and over again for different datatypes, either because datatypes change during the development of programs, or because functions with similar functionality are needed on different datatypes. Examples of such functions are pretty printers, equality functions, unifiers, pattern matchers, rewriting functions, etc. Such functions are called polytypic functions. A polytypic function is a function that is defined by induction on the structure of user-defined datatypes. This paper introduces polytypic functions, and shows how to construct and reason about polytypic functions. A larger example is studied in detail: polytypic functions for term rewriting and for determining whether or not a collection of rewrite rules is normalising.

[1] An earlier version of this paper (with the same title) is published in J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, LNCS, Springer Verlag, 1996.

3.1 Introduction

Complex software systems contain many datatypes, which change during the development of the system. Developing innovative and complex software is typically an evolutionary process. Furthermore, such systems contain functions that have the same functionality on different datatypes, such as equality functions, print functions, parse functions, etc. Software should be written so that the impact of changes to the software is as limited as possible. Polytypic programs are programs that adapt automatically to changing structure, and thus reduce the impact of changes. This effect is achieved by writing programs such that they work for large classes of datatypes.

Consider for example the function `length :: List a -> Int`, which counts the number of values of type `a` in a list. There exists a very similar function `numOfNodes :: Tree a -> Int`, which counts the number of occurrences of `a`'s in a tree. We can generalise these two functions into a single function, `size :: d a -> Int`, which is not only polymorphic in `a`, but also in the type constructor `d`.[1] We call such functions *polytypic functions* [44]. Once we have a polytypic `size` function, function `size` can be applied to values of any datatype. If a datatype is changed, `size` still behaves as expected. For example, the datatype `List a` has two constructors with which lists can be built: the empty list constructor, and the cons constructor, which prepends an element to a list. If we add a constructor with which we can append an element to a list, function `size` still behaves as expected, and counts the number of elements in a list.

Polytypic functions are useful in many situations, for example in implementing rewriting systems.

3.1.1 A problem

Suppose we want to write a term rewriting module. An example of a term rewriting system is the algebra of numbers constructed with `Zero`, `Succ`, `:+:`, and `:*:`, together with the following term rewrite rules [53].

```
x :+: Zero    ->  x
x :+: Succ y  ->  Succ (x :+: y)
x :*: Zero    ->  Zero
x :*: Succ y  ->  (x :*: y) :+: x
```

where `x` and `y` are variables. For confluent and normalising term rewriting systems, the relation $\overset{*}{\rightarrow}$, which rewrites a term to its normal form, is a func-

[1] We will later see that `d` has to be restricted to *regular* datatype constructors.

tion. For example `Succ (Succ Zero) :*: Succ (Succ Zero)` $\stackrel{*}{\to}$ `Succ (Succ (Succ (Succ Zero)))`.

We want to implement function $\stackrel{*}{\to}$ in a functional language such as Haskell [21], that is, we want to define a function `rewrite` that takes a list of rewrite rules and a term, and reduces redeces until no further reduction is possible. For the above example, we first define two datatypes: the datatype of numbers, and the datatype of numbers with variables, which is used for representing the rewrite rules. Variables are represented by integers.

```
data Number = Zero
            | Succ Number
            | Number :+: Number
            | Number :*: Number

data VNumber = Var Int
             | VZero
             | VSucc VNumber
             | VNumber :++: VNumber
             | VNumber :**: VNumber
```

A rewrite rule is represented by a pair of values of type `VNumber`.

We want to use function `rewrite` on different datatypes: rewriting is independent of the specific datatype. For example, we also want to be able to rewrite `SKI` terms, where an `SKI` term is a term built with the constant constructors `S`, `K`, `I`, and the application constructor `:@:`. We have the following rewrite rules for `SKI` terms:

```
((S :@: x) :@: y) :@: z  ->  (x :@: z) :@: (y :@: z)
(K :@: x) :@: y          ->  x
I :@: x                  ->  x
```

Since the type of function `rewrite` is independent of the specific datatype on which it is going to be used, we want to define function `rewrite` in a class.

```
class Rewrite a b where
  rewrite      :: [(b,b)] -> a -> a
  rewrite rs   =  fp (rewrite_step rs)
  rewrite_step  :: [(b,b)] -> a -> a

fp f x | f x == x   =  x
       | otherwise  =  fp f (f x)
```

Function `rewrite_step` finds a suitable redex (depending on the reduction strategy used), and rewrites it.

There are a number of problems with this solution. First, the solution is illegal Haskell, because of the two type variables in the class declaration. More important is that the relation between a datatype without and with variables is lost in the above declaration. But most important: although the informal description of `rewrite_step` is independent of a specific datatype, we have to give an instance of function `rewrite_step` on each datatype we want to use it. We would like to have a module that supplies a rewrite function for each conceivable datatype.

3.1.2 A solution

We have extended Haskell with the possibility to define polytypic functions. A polytypic function is a function parametrised on a type and can thus be viewed as a family of functions: one function for each datatype. This parametrisation can be implemented using constructor classes [51] (a higher order variant of so called *ad hoc* polymorphism). But unlike *ad hoc* polymorphic functions which need one instance declaration for every datatype they are used on, we define polytypic functions by induction on the structure of user-defined datatypes.

If we define function `rewrite_step` as a polytypic function, then each time we use function `rewrite_step` on a specific datatype, code for that instance of `rewrite_step` is automatically generated. Polytypic function definitions are type checked, and the generated functions are guaranteed to be type correct. Polytypic functions add the possibility to define functions over large classes of datatypes in a strongly typed language.

3.1.3 Why polytypic programming?

Polytypic functions are general and abstract functions which occur often in everyday programming; examples are equality `==` and `map`. Polytypic functions are useful when building complex software systems, because they adapt automatically to changing structure, and they are useful for:

- Implementing term rewriting systems, program transformation systems, pretty printers, theorem provers, debuggers, and other general purpose systems that are used to reason about and manipulate different datatypes in a structured way.

- Generalising Haskell's [21] `deriving` construct. Haskell's `deriving` construct can be used to generate code for for example the equality function and the printing function on a lot of datatypes. A fixed set of predefined Haskell classes can be used in the `deriving` construct, and programmers cannot add new classes to be used in it. The functions in these classes are easily written as polytypic functions (see section 3.3.4 for some examples).

- Implementing Squiggol's [59, 63, 65, 67] general purpose datatype independent functions such as `cata`, `map`, `zip`, `para` etc.

- Implementing general purpose, datatype independent programs for unification [38, 40], pattern matching [44], data compression [39], etc.

3.1.4 Writing polytypic programs

There are various ways to implement polytypic programs in a typed language.[2] Three possibilities are:

- using a universal datatype;

- using higher-order polymorphism and constructor classes;

- using a special syntactic construct.

Polytypic functions can be written 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 by providing 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 [49], type information is preserved, and we can use current functional languages such as Gofer and Haskell for implementing polytypic functions. However, writing such programs is rather cumbersome: programs become cluttered with instance declarations, and type declarations become cluttered with contexts. Furthermore, it is hard to deal with mutual recursive datatypes.

Since the first two solutions to writing polytypic functions are dissatisfying, we have extended Haskell with a syntactic construct for defining polytypic functions [39]. Thus polytypic functions can be implemented and type checked. The

[2] Polytypic programs can be written in untyped languages like `lisp` but, of course, without any type safety. We will only consider strongly typed languages in this paper.

resulting language is called PolyP. Consult the web site

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

to obtain a compiler that compiles PolyP into Haskell (which subsequently can be compiled with a Haskell compiler), and for the latest developments on PolyP.

In order to be able to define polytypic functions we need access to the structure of a datatype `D a`. In this paper we will restrict `D a` to be a so-called *regular* datatype. A datatype `D a` is regular if it contains no function spaces, and if the arguments of the type constructor `D` on the left- and right-hand side in its definition are the same. The collection of regular datatypes contains all conventional recursive datatypes, such as `Nat`, `List a`, and different kinds of trees. Polytypic functions can, with some effort, be defined on a larger class of datatypes, including datatypes with function spaces [29, 66], but regular datatypes suffice for our purposes.

3.1.5 Background and related work

The basic idea behind polytypic programming is the idea of modelling 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 [56], Manes and Arbib [60], and Hagino [31].

Polytypic functions are widely used in the Squiggol community, see [27, 59, 63, 64, 65, 67], where the 'Theory of Lists' [10, 11, 43] is extended to datatypes that can be defined by means of a regular functor. The polytypic functions used in Squiggol are general recursive combinators such as catamorphisms (generalised folds), paramorphisms, maps, etc. Böhm and Berarducci [13], and Sheard [81] give programs that automatically synthesise these functions. In the language `Charity` [15] polytypic functions like the catamorphism and map are automatically provided for each user-defined datatype. Polytypic functions for specific problems, such as the maximum segment sum problem and the pattern matching problem were first given by Bird *et al.* [9] and Jeuring [44]. Special purpose polytypic functions such as the generalised version of function `length` and the operator `(==)` can be found in [38, 63, 70, 71, 78]. Jay [42] has developed an alternative theory for polytypic functions (in his terminology: *shapely* functions), in which values are represented by their structure and their contents.

Type systems for languages with constructs for writing polytypic functions have been developed by Jay [41], Ruehr [77], Sheard and Nelson [80], and Jansson and Jeuring [39]. Our extension of Haskell is based on the type system described in [39].

In object-oriented programming polytypic programming appears under the names 'design patterns' [30], and 'adaptive object-oriented programming' [57, 73]. In adaptive object-oriented programming methods are attached to groups of classes that usually satisfy certain constraints. The adaptive object-oriented programming style is very different from polytypic programming, but the resulting programs have very similar behaviour.

3.1.6 Overview

This paper is organised as follows. Section 3.2 explains the relation between datatypes and functors, and defines some basic (structured recursion) operators on some example datatypes. Section 3.3 introduces polytypic functions. Section 3.4 describes polytypic functions for rewriting terms, and for determining whether a set of rewrite rules is normalising. Section 3.5 concludes the paper.

3.2 Datatypes and functors

A datatype can be modelled by an initial object in the category of F-algebras, where F is the functor describing the *structure* of the datatype. The essence of polytypic programming is that functions can be defined by induction on the structure of datatypes. This section introduces functors, and shows how they are used in describing the structure of datatypes. The first subsection presents some notation and the second discusses a simple non-recursive datatype. The next three subsections discuss recursive datatypes, and give the definitions of basic structured recursion operators on these datatypes. The sixth subsection introduces the type operators *FunctorOf* and *Mu* used to express the relationship between a regular datatype and its functor and the last subsection defines the polytypic Fusion law and gives examples of its use.

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 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) optimisations than their unstructured counterparts. Furthermore, since polytypic functions are defined for arbitrary datatypes, we cannot use traditional pattern matching in definitions of polytypic functions, and the only resources for polytypic function definitions are structured recursion operators. One of the most basic structured recursion operators is the *catamorphism*. This section defines catamorphisms on datatypes, and shows how catamorphisms can be used in the definitions of many other functions. Furthermore, it briefly discusses the fusion law for catamorphisms.

3.2.1 Notation

We will use a Haskell inspired notation for types and functions. The datatype *Either a b* is a labeled sum consisting of left-tagged elements of type a, and right-tagged elements of type b, and has constructors *Left* and *Right* :

$$\begin{array}{lcl} \mathit{data}\ \mathit{Either}\ a\ b & = & \mathit{Left}\ a \ \mid\ \mathit{Right}\ b \\ \mathit{Left} & :: & a \to \mathit{Either}\ a\ b \\ \mathit{Right} & :: & b \to \mathit{Either}\ a\ b \end{array}$$

In category theory, a functor is a mapping between categories that preserves the algebraic structure of the category. A bifunctor is a two-argument functor. Since a category consists of objects (types) and arrows (functions), a functor consists of two parts: a definition on types, and a definition on functions. The type constructor *Either* is the part of a bifunctor that takes two types and returns a type. The operator $+$ is the part of this bifunctor that takes two functions and returns a function.

$$\left.\begin{array}{lcl} \multicolumn{3}{l}{(+)\ ::\ (a \to c) \to (b \to d) \to \mathit{Either}\ a\ b \to \mathit{Either}\ c\ d} \\ (f+g)\ (\mathit{Left}\ x) & = & \mathit{Left}\ (f\ x) \\ (f+g)\ (\mathit{Right}\ y) & = & \mathit{Right}\ (g\ y) \end{array}\right\} \tag{3.1}$$

To combine functions to operate on sums we also use the operator *either*, which takes a function f of type $a \to c$ and a function g of type $b \to c$, and applies f to left-tagged values, and g to right-tagged values, throwing away the tag information:[3]

$$\left.\begin{array}{lcl} \multicolumn{3}{l}{\mathit{either}\ ::\ (a \to c) \to (b \to c) \to \mathit{Either}\ a\ b \to c} \\ (f\ `\mathit{either}`\ g)\ (\mathit{Left}\ x) & = & f\ x \\ (f\ `\mathit{either}`\ g)\ (\mathit{Right}\ y) & = & g\ y \end{array}\right\} \tag{3.2}$$

We write $()$ to denote the datatype containing one element, which is also denoted by $()$. The product type (a, b) denotes the datatype of pairs (x, y) with the usual destructors *fst*, and *snd*:

$$\begin{array}{rcl} \mathit{fst} & :: & (a,b) \to a \\ \mathit{snd} & :: & (a,b) \to b \end{array}$$

The pairing operator, $(\_,\_)$, together with the operator $\times$ form a bifunctor:

$$\begin{array}{rcl} (\times) & :: & (a \to c) \to (b \to d) \to (a,b) \to (c,d) \\ (f \times g)\ (x,y) & = & (f\ x, g\ y) \end{array}$$

[3] Function *either* is actually the catamorphism on the datatype *Either a b*. Both *Either* and *either* are predefined in Haskell 1.3

3.2.2 A datatype for computations that may fail

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

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

For example, we can define the expression *divide m n* to be equal to *Nothing* if n equals zero, and *Just* (m/n) otherwise.

We model datatypes as fix-points, even non-recursive ones like *Maybe*, as we want to use the same representation for all datatypes. To be able to use polytypic functions on the datatype *Maybe a* we have to extract the structure of this datatype. The datatype *Maybe a* can be modelled by the type *Mu FMaybe a*, where *Mu* is the fix-point constructor, *FMaybe* is a functor which describes the structure of the datatype *Maybe a*, and a is the parameter of the datatype. Since we are only interested in the structure of *Maybe a*, the names of the constructors of *Maybe a* are not important. We define *FMaybe* by removing *Maybe*'s constructors (writing () for the empty space we obtain by removing *Nothing*), and replacing the infix | with a prefix *Either*:[4]

$$\mathit{FMaybe\ a\ r} \quad = \quad \mathit{Either}\ ()\ a$$

We now abstract from the arguments a and r in *FMaybe*. Function *Par* returns the parameter a (the first argument to the functor). The operators are lifted to the function level: *Either* is lifted to $+$ and the empty product () is lifted to *Empty*.

$$\mathit{FMaybe} \quad = \quad \mathit{Empty} + \mathit{Par}$$

The function $\mathit{inn}_{\mathit{Maybe}}$ injects values of type *Either* () a into the type *Maybe a*. Function $\mathit{out}_{\mathit{Maybe}}$ is the inverse of function $\mathit{inn}_{\mathit{Maybe}}$: it projects values out of the type *Maybe a*.

$$\begin{aligned} \mathit{inn}_{\mathit{Maybe}} &\quad :: \quad \mathit{FMaybe\ a} \to \mathit{Maybe\ a} \\ \mathit{out}_{\mathit{Maybe}} &\quad :: \quad \mathit{Maybe\ a} \to \mathit{FMaybe\ a} \end{aligned}$$

The definitions of these functions are simple and omitted; in the polytypic programming system PolyP these functions are automatically supplied by the system for each user-defined datatype.

A function that handles values of type *Maybe a* consists of two components: a component that deals with the constructor *Nothing*, and a component that deals

[4] *FMaybe* is only the type part of the functor, the function part is $\mathit{fmap}_{\mathit{Maybe}}\ a\ r = \mathit{id} + a$.

with values of the form $Just\ x$. Such functions are called catamorphisms (abbreviated to *cata*). In general, a catamorphism is a function that replaces constructors by functions. The definition of a catamorphism on the datatype $Maybe\ a$ is very simple; definitions of catamorphisms on recursive types are more involved.

Function $cata_{Maybe}$ takes an argument n ‘$either$‘ j of type $FMaybe\ a \to b$, and replaces the representation of $Nothing$ in $Maybe\ a$ by n (), and the representation of $Just$ in $Maybe\ a$ by j.

$$\begin{array}{rcl} cata_{Maybe} & :: & (FMaybe\ a \to b) \to Maybe\ a \to\ b \\ cata_{Maybe}\ h & = & h\ \cdot\ out_{Maybe} \end{array}$$

For example, the function $size_{Maybe}$ that takes a $Maybe\ a$-value and returns 0 if it is of the form $Nothing$, and 1 otherwise, is defined by

$$size_{Maybe} \quad = \quad cata_{Maybe}\ ((\backslash x \to 0)\ \text{‘}either\text{‘}\ (\backslash x \to 1))$$

This might seem a complicated way to define function $size_{Maybe}$, but we will see later that this definition easily generalises to other datatypes.

The prelude of Haskell 1.3 contains a function $maybe$ defined by:

$$\begin{array}{llllllll} maybe & :: & a & \to & (a \to b) & \to & Maybe\ a & \to\ b \\ maybe & & n & & j & & Nothing & =\ n \\ maybe & & n & & j & & (Just\ a) & =\ j\ a \end{array}$$

This function has the same functionality as function $cata_{Maybe}$, and we will use it in the rest of the paper whenever we need a catamorphism on $Maybe\ a$.

3.2.3 A datatype for lists

Consider the datatype $List\ a$ defined by

$$data\ List\ a \quad = \quad Nil \mid Cons\ a\ (List\ a)$$

Values of this datatype are built by prepending values of type a to a list. Again, since we are only interested in the structure of $List\ a$, the names of the constructors are not important. As an element of $List$ is either a nullary constructor or a binary constructor with its two arguments we can represent it by the fix-point of the functor $FList$:

$$FList\ a\ r \quad = \quad Either\ ()\ (a, r)$$

Note that the arguments of $Cons$ are replaced by a pair. We now abstract from the arguments a and r in $FList$. Function Par returns the parameter a (the first argument), and function Rec returns the recursive parameter r (the second argument). The operators are lifted to the function level: $Either$ is lifted to $+$, the pairing operator $(\_,\_)$ is lifted to $\times$ and the empty product $()$ is lifted to $Empty$. As usual, $\times$ binds stronger than $+$.

$$FList \quad = \quad Empty + Par \times Rec$$

The initial object in the category of $FList\ a$-algebras (i.e. 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 constructor function inn_{List}, that constructs elements of the datatype $List\ a$. Function inn_{List} combines the constructors Nil and $Cons$ in a single constructor function for the datatype $List\ a$:[5]

$$\begin{array}{lcl} inn_{List} & :: & FList\ a\ (List\ a) \to List\ a \\ inn_{List} & = & const\ Nil\ \text{`}either\text{`}\ \ uncurry\ Cons \end{array}$$

For example, the list containing only the integer 3, $Cons\ 3\ Nil$, can be constructed as $inn_{List}\ (Right\ (3, inn_{List}\ (Left\ ())))$. Function out_{List} is the inverse of function inn_{List}:

$$\begin{array}{llcl} out_{List} :: & List\ a & \to & FList\ a\ (List\ a) \\ out_{List} & Nil & = & Left\ () \\ out_{List} & (Cons\ a\ l) & = & Right\ (a, l) \end{array} \tag{3.3}$$

$FList$ takes two types and returns a type, and function $fmap_{List}$ takes two functions and returns a function. Together they form a bifunctor.

$$\begin{array}{lcl} fmap_{List} & :: & (a \to c) \to (b \to d) \to FList\ a\ b \to FList\ c\ d \\ fmap_{List} & = & \backslash f\ g \to id + g \times f \end{array} \tag{3.4}$$

As examples of programs that can be defined using the combinators introduced so far we take $take\ n\ l$, which returns a list containing the first n elements of the list l, and $map_{List}\ f\ l$, which applies function f on all elements of the list l:

$$\begin{array}{rcl} take & :: & Int \to List\ a \to List\ a \\ take\ 0 & = & const\ Nil \\ take\ n & = & inn_{List}\ \cdot\ fmap_{List}\ id\ (take\ (n-1))\ \cdot\ out_{List} \end{array}$$

$$\begin{array}{rcl} map_{List} & :: & (a \to b) \to List\ a \to List\ b \\ map_{List}\ f & = & inn_{List}\ \cdot\ fmap_{List}\ f\ (map_{List}\ f)\ \cdot\ out_{List} \end{array} \tag{3.5}$$

The type constructor $List$ and the function map_{List} form a functor, just as $FList$ and $fmap_{List}$ form a functor.

[5] The definitions of *inn* and *out* are those generated by PolyP given the datatype of lists.

3.2.4 Catamorphisms on lists

Function $size_{List}$ returns the number of elements in a $List\ a$ (function `length` in Haskell). Given an argument list, the value of function $size_{List}$ can be computed by replacing the constructor Nil by 0, and the constructor $Cons$ by 1+, for example,

$$\begin{array}{llll} Cons\ 2 & (Cons\ 5 & (Cons\ 3 & Nil\ )) \\ 1+ & (1+ & (1+ & 0\ \ )) \end{array}$$

So the $size_{List}$ of this list is 3. We use a higher-order function to describe functions that replace constructors by functions: the catamorphism. The catamorphism on $List\ a$ is the equivalent of function `foldr` on lists in Haskell. It is the basic structured recursion operator on $List\ a$. Function $cata_{List}$ takes an argument e '$either$' f of type $FList\ a\ b \to b$, and replaces Nil by e, and $Cons$ by f:

$$\begin{array}{llllllll} Cons & 2 & (Cons & 5 & (Cons & 3 & Nil &)) \\ f & 2 & (f & 5 & (f & 3 & e &)) \end{array}$$

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

$$\begin{array}{rcl} cata_{List} & :: & (FList\ a\ b \to b) \to List\ a \to b \\ cata_{List}\ f & = & f\ \cdot\ fmap_{List}\ id\ (cata_{List}\ f)\ \cdot\ out_{List} \end{array}$$

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 $cata_{List}\ f :: List\ a \to b$. The initiality of this algebra also means that $cata_{List}\ inn_{List}$ is the identity function on $List\ a$. See [56, 59] for more information about the underlying theory.

We use function $cata_{List}$ to define the function $size_{List}$ and to give an alternative definition of map_{List}.

$$\begin{array}{rcl} size_{List} & :: & List\ a \to Int \\ size_{List} & = & cata_{List}\ ((\backslash x \to 0)\ \text{`}either\text{`}\ (\backslash(x,n) \to n+1)) \\ map'_{List} & :: & (a \to b) \to List\ a \to List\ b \\ map'_{List}\ f & = & cata_{List}\ (inn_{List}\ \cdot\ fmap_{List}\ f\ id) \end{array} \tag{3.6}$$

3.2.5 A datatype for trees

The datatype $Tree\ a$ is defined by

$$data\ Tree\ a \quad = \quad Leaf\ a \mid Bin\ (Tree\ a)\ (Tree\ a)$$

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

$$FTree \quad = \quad Par + Rec \times Rec$$

Functions inn_{Tree} and out_{Tree} are similar to functions *inn* and *out* on *List a*. The functions $cata_{Tree}$ and map_{Tree} on *Tree a* are defined in terms of functions inn_{Tree}, out_{Tree} and $fmap_{Tree}$:

$$\begin{array}{rcl} fmap_{Tree} & :: & (a \to c) \to (b \to d) \to FTree\ a\ b \to FTree\ c\ d \\ fmap_{Tree} & = & \backslash f \to \backslash g \to f + g \times g \end{array} \tag{3.7}$$

$$\begin{array}{rcl} map_{Tree} & :: & (a \to b) \to Tree\ a \to Tree\ b \\ map_{Tree}\ f & = & inn_{Tree}\ \cdot\ fmap_{Tree}\ f\ (map_{Tree}\ f)\ \cdot\ out_{Tree} \end{array}$$

$$\begin{array}{rcl} cata_{Tree} & :: & (FTree\ a\ b \to b) \to Tree\ a \to b \\ cata_{Tree}\ f & = & f\ \cdot\ fmap_{Tree}\ id\ (cata_{Tree}\ f)\ \cdot\ out_{Tree} \end{array}$$

Note that the the only difference between the definition of $cata_{Tree}$ and the definition of $cata_{List}$ are the indices.

Function $size_{Tree}$ is defined by

$$\begin{array}{rcl} size_{Tree} & :: & Tree\ a \to Int \\ size_{Tree} & = & cata_{Tree}\ (\backslash x \to 1\ \text{`}either\text{`}\ \backslash(m, n) \to m + n) \end{array}$$

3.2.6 Functors for datatypes

We have given functors that describe the structure of the datatypes *Maybe a*, *List a* and *Tree a*. For each regular datatype *D a* there exists a bifunctor *F* such that the datatype is the fixed point in the category of *F a*-algebras [59]. The argument *a* of *F* encodes the parameters of the datatype *D a*. From the PolyP-users point of view, a functor is a value generated by the following grammar.

$$F \ ::= \ F + F \mid F \times F \mid Empty \mid Par \mid Rec \mid D\ @\ F \mid Const\ t \tag{3.8}$$

Here *t* is one of the basic types *Bool*, *Int*, etc., +, ×, and @ are binary infix constructors, and *Empty*, *Par*, *Rec* and *Const t* are nullary constructor (for each base type *t*). Using this grammar, it is impossible to differentiate between the structure of datatypes such as:

$$\begin{array}{rcl} data\ Point\ a & = & Point\ (a, a) \\ data\ Point'\ a & = & Point'\ a\ a \\ FPoint & = & Par \times Par \end{array}$$

Functor *FPoint* describes the structure of both *Point a* and *Point' a*. This implies that it is impossible to use the fact that a constructor is curried or not in the definition of a polytypic function. (This is a design choice that simplifies the implementation of PolyP but restricts the expressiveness of the functor language. We will try to remove this restriction in the future.) PolyP's internal representation of a functor is (of course) more involved. We note the following about the datatype of functors:

- The operators $+$ and $\times$ are right-associative, so $f+g+h$ is represented as $f+(g+h)$. Operator $\times$ binds stronger than $+$. The empty product *Empty* is the unit of $\times$. Operator $+$ may only occur at top level, so $f \times (g+h)$ 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.

- The alternative D @ F in this grammar 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*:

$$\begin{array}{rcl} \mathit{data\ Rose\ a} & = & \mathit{Fork\ a\ (List\ (Rose\ a))} \\ \mathit{FRose} & = & \mathit{Par} \times (\mathit{List}\ @\ \mathit{Rec}) \end{array}$$

- Datatypes with more than one type argument can not be directly represented by these functors[6] but we are working on extending the system to allow an index in the parameter case, Par_i, indicating which of the parameters is intended. Bellè *et al.* [7] describe a type system that allows some polytypic functions that handles multiple parameters.

- In this paper we will not discuss mutually recursive datatypes. However, it will be possible to define polytypic functions over mutually recursive datatypes in PolyP. (Expressed in functors by indexing the *Rec* case.)

- For a datatype that is defined using a constant type such as *Int* or *Char* we use the *Const* functor. As an example we give the functor the following simple datatype of types:

$$\begin{array}{rcl} \mathit{data\ Type\ a} & = & \mathit{Con\ String} \mid \mathit{Var\ a} \mid \mathit{Fun\ (Type\ a)\ (Type\ a)} \\ \mathit{FType} & = & \mathit{Const\ String} + \mathit{Par} + \mathit{Rec} \times \mathit{Rec} \end{array}$$

The use of functors in the representation of datatypes is central to polytypic programming: polytypic functions are defined by induction on functors.

[6]But they can be simulated using a one parameter datatype over a sum of the parameter types.

To express the relationship between regular datatypes and the corresponding functors we use the type operators *FunctorOf* and *Mu* satisfying the following equalities:

$$\begin{array}{rcl} \mathit{Mu}\ (\mathit{FunctorOf}\ d) & = & d \\ \mathit{FunctorOf}\ (\mathit{Mu}\ f) & = & f \end{array}$$

For the types defined in this chapter we have:

$$\begin{array}{llll} \mathit{FunctorOf\ Maybe} & = \mathit{FMaybe} & = \mathit{Empty} + \mathit{Par} \\ \mathit{FunctorOf\ List} & = \mathit{FList} & = \mathit{Empty} + \mathit{Par} \times \mathit{Rec} \\ \mathit{FunctorOf\ Tree} & = \mathit{FTree} & = \mathit{Par} + \mathit{Rec} \times \mathit{Rec} \\ \mathit{FunctorOf\ Rose} & = \mathit{FRose} & = \mathit{Par} \times (\mathit{List}\ @\ \mathit{Rec}) \\ \mathit{FunctorOf\ Type} & = \mathit{FType} & = \mathit{Const\ String} + \mathit{Par} + \mathit{Rec} \times \mathit{Rec} \end{array}$$

3.2.7 Fusion

Function $cata_D$ satisfies the so-called *Fusion law* defined in figure 3.1 where we use the notation $\mathcal{F}_d$ for *FunctorOf* d.

$$\begin{array}{ll} & h \cdot cata_D\ f \ = \ cata_D\ g \\ \Leftarrow & \quad (\text{Fusion}) \\ & h \cdot f \ = \ g \cdot fmap_D\ id\ h \end{array} \qquad \text{where} \quad \left\{ \begin{array}{lcl} f & :: & \mathcal{F}_D\ a\ b \to b \\ g & :: & \mathcal{F}_D\ a\ c \to c \\ h & :: & b \to c \\ cata_D\ f & :: & D\ a \to b \\ cata_D\ g & :: & D\ a \to c \end{array} \right.$$

Figure 3.1: The Fusion law

The fusion law gives conditions under which intermediate values produced by a catamorphism can be eliminated. The fusion law is a polytypic law: it is parametrised on the datatype constructor D. For each regular datatype $D\ a$, fusion is a direct consequence of the free theorem [86] of the functional $cata_D$. It can also be proved using induction over the datatype D. If we allow partial or infinite objects we get the extra requirement that h be strict.

We give two examples of the use of the fusion law: one proof for trees and one for any datatype $D\ a$. As the example for trees we show that the *size* of a tree is the same as the length of the list of all elements in the tree: $\# \cdot \mathit{flatten}_{Tree} = \mathit{size}_{Tree}$, where # is an abbreviation for the function that computes the length of a list.[7]

[7] The theorem $\# \cdot \mathit{flatten}_D = \mathit{size}_D$ holds for any regular datatype $D\ a$ but this particular proof only holds for trees.

The function $flatten_{Tree}$ returns a list containing the elements of the argument tree:

$$\begin{array}{lcl} \mathit{flatten}_{Tree} & :: & \mathit{Tree}\ a \to [a] \\ \mathit{flatten}_{Tree} & = & \mathit{cata}_{Tree}\ (\backslash x \to [x]\ \text{`}\mathit{either}\text{`}\ \backslash(x,y) \to x{+\!\!+}y) \end{array}$$

In the proof we use abbreviations for the arguments to the catamorphisms in the definitions of $\mathit{flatten}_{Tree}$ and $size_{Tree}$.

$$\begin{array}{ll}
 & \# \cdot \mathit{flatten}_{Tree} = size_{Tree} \\
\Leftrightarrow & \quad \text{(by definition, introducing the abbreviations } \phi \text{ and } \sigma\text{)} \\
 & \# \cdot cata_{Tree}\ \phi = cata_{Tree}\ \sigma \\
\Leftarrow & \quad \text{(Fusion)} \\
 & \# \cdot \phi \ = \ \sigma \cdot (fmap_{Tree}\ id\ \#) \\
\Leftrightarrow & \quad \text{(by definition of } fmap_{Tree}\text{)} \\
 & \# \cdot \phi \ = \ \sigma \cdot (id + (\# \times \#)) \\
\Leftrightarrow & \\
 & \# \cdot (\phi_1\ \text{`}either\text{`}\ \phi_2) \ = \ (\sigma_1\ \text{`}either\text{`}\ \sigma_2) \cdot (id + (\# \times \#)) \\
\Leftrightarrow & \quad \text{(rules for } either \text{ [27])} \\
 & (\# \cdot \phi_1)\ \text{`}either\text{`}\ (\# \cdot \phi_2) \ = \ (\sigma_1 \cdot id)\ \text{`}either\text{`}\ (\sigma_2 \cdot (\# \times \#)) \\
\Leftrightarrow & \quad \text{(split the } either\text{s and simplify)} \\
 & \# \cdot \phi_1 = \sigma_1 \ \ \wedge \ \ \# \cdot \phi_2 = \sigma_2 \cdot (\# \times \#) \\
\Leftrightarrow & \quad \text{(introduce arguments, implicitly } \forall\text{-quantified (Leibniz))} \\
 & \#(\phi_1\ x) = \sigma_1\ x \ \ \wedge \ \ \#(\phi_2\ (l, l')) = \sigma_2\ (\#l, \#l') \\
\Leftrightarrow & \quad \text{(definition of } \phi_i \text{ and } \sigma_j\text{)} \\
 & \#[x] = 1 \ \ \wedge \ \ \#(l{+\!\!+}l') = \#l + \#l' \\
\Leftrightarrow & \quad \text{(properties of } \#\text{)} \\
 & True \ \ \wedge \ \ True
\end{array}$$

As an example of a polytypic proof we we prove that the functions $map_D\ f$ and $map'_D\ f$ are equal, where map_D and map'_D are generalised versions of map_{List} and map'_{List} from equations 3.5 and 3.6:

$$\begin{array}{rcl} map_D, map'_D & :: & (a \to b) \to D\ a \to D\ b \\ map_D\ f & = & inn_D \cdot fmap_D\ f\ (map_D\ f) \cdot out_D \\ map'_D\ f & = & cata_D\ (inn_D \cdot fmap_D\ f\ id) \end{array}$$

In the proof we assume that that id is a catamorphism and that $fmap$ is a bifunctor. This is true for all regular datatypes.

$$
\begin{array}{ll}
& map_D\ f = map'_D\ f \\
\Leftrightarrow & \quad \text{(identity)} \\
& map_D\ f\ \cdot\ id = map'_D\ f \\
\Leftrightarrow & \quad (id \text{ is a catamorphism, definition of } map'_D) \\
& map_D\ f\ \cdot\ cata_D\ inn_D = cata_D\ (inn_D\ \cdot\ fmap_D\ f\ id) \\
\Leftarrow & \quad \text{(Fusion)} \\
& map_D\ f\ \cdot inn_D = (inn_D \cdot\ fmap_D\ f\ id)\ \cdot fmap_D\ id\ (map_D\ f) \\
\Leftrightarrow & \quad (\text{definition of } map_D,\ fmap_D \text{ is a bifunctor}) \\
& inn_D \cdot fmap_D\ f\ (map_D\ f) \cdot out_D \cdot inn_D = \\
& inn_D \cdot fmap_D\ (f \cdot id)\ (id \cdot (map_D\ f)) \\
\Leftrightarrow & \quad (out_D \text{ is the inverse of } inn_D, \text{ identity}) \\
& inn_D\ \cdot\ fmap_D\ f\ (map_D\ f) = inn_D\ \cdot\ fmap_D\ f\ (map_D\ f) \\
\Leftrightarrow & \\
& True
\end{array}
$$

3.3 Polytypic functions

This section introduces polytypic functions and shows how they are expressed in PolyP. We will define the polytypic versions of functions *fmap*, *cata*, *size* and *map*. We will briefly discuss a type system that supports writing polytypic functions, and we will show how some of the functions that can be derived in Haskell can be defined as polytypic functions.

To express that a polytypic function is defined only for *regular* datatypes `d` we prepend `Regular d =>` to the type of the function.

3.3.1 Basic polytypic functions

Functions `inn` and `out`

Functions `inn` and `out` are the basic functions with which elements of datatypes are constructed and decomposed in definitions of polytypic functions. These two functions are the *only* functions that can be used to manipulate values of datatypes in polytypic functions.

```
out :: Regular d => d a -> (FunctorOf d) a (d a)
inn :: Regular d => d a <- (FunctorOf d) a (d a)
```

Function `out` is our main means for avoiding definitions by pattern matching. Informally, instead of defining for example `f (Pattern x) = foo x` we now define `f = foo . out`. Function `inn` (the inverse of `out`) is *the* polytypic constructor of all regular datatypes. Both `inn` and `out` are primitive functions built into PolyP.

Functions `pmap` and `cata`

A polytypic function is a function that is defined by induction on the structure of user-defined datatypes, i.e., by induction on functors, or a function defined in terms of such inductive functions. Except for the type declarations (and the indices), the definitions of `pmap`[8] and `cata` are the same as their definitions on *List a* and *Tree a*.

```
pmap   :: Regular d => (a -> b) -> d a -> d b
pmap p  =  inn . fmap p (pmap p) . out

cata   :: Regular d => ((FunctorOf d) a b -> b) -> d a -> b
cata h  =  h . fmap id (cata h) . out
```

The polytypic *map*-function, `pmap` applies function `p` to all elements in a structure. Function `cata` recursively replaces constructors by functions. Except for the context part `Regular d =>` that simply states that `d a` must be a regular datatype, the typing of the definition of `cata` is shown in figure 3.2 where we use the notation $\mathcal{F}_d$ for `FunctorOf d`.

$$cata_d\ h \ = \ h \cdot fmap_d\ id\,(cata_d\ h) \cdot out_d$$

$$\begin{array}{llllllll}
out_d & :: & d\ a & \to & \mathcal{F}_d\ a\ (d\ a) & & & \\
fmap_d\ id\,(cata_d\ h) & :: & & & \mathcal{F}_d\ a\ (d\ a) & \to & \mathcal{F}_d\ a\ b & \\
h & :: & & & & & \mathcal{F}_d\ a\ b & \to\ b \\
cata_d\ h & :: & d\ a & & & & & \to\ b
\end{array}$$

Figure 3.2: The typing of the catamorphism.

The polytypic function `size` is an example of a catamorphism. It takes a value `x` of datatype `d a` and counts the number of occurrences of values of type `a` in `x`.

```
size  :: Regular d => d a -> Int
size  =  cata fsize
```

[8]We call the polytypic `map` function `pmap` to avoid a name clash with the normal Haskell function `map`.

The argument to `cata` must be defined by induction on the structure of the datatype `d a`.

A definition of a polytypic function by induction on functors starts with the keyword `polytypic`, followed by the name of the function and its type. The type declaration and the inductive definition of the function are separated by an equality sign. As a first example, consider the definition of `fsize`. We will explain what we mean with this definition below.

```
polytypic fsize :: f a Int -> Int
  = case f of
      g + h     ->  fsize `either` fsize
      g * h     ->  \(x,y) -> fsize x + fsize y
      Empty     ->  \x -> 0
      Par       ->  \x -> 1
      Rec       ->  \x -> x
      d @ g     ->  \x -> sum (pmap fsize x)
      Const t   ->  \x -> 0

either :: (a -> c) -> (b -> c) -> Either a b -> c
```

where function `sum` sums the integers of a value of an arbitrary datatype. If function `size` is applied to a value of the datatype *List a* or *Tree a*, PolyP generates the right instantiation for function `size`.

One can see this definition as a definition of a family of functions, one for each `f` on which `fsize` is used. Note that the type variable `f` has kind `* -> * -> *`, that is, `f` takes two types and produces a type. We call variable `f` a *functor* variable. Note that the different cases in the definition of a polytypic function correspond to the different components of the datatype for functors described in Section 3.2.

The definition of function `fsize` requires the existence of polytypic functions `sum :: Regular d => d Int -> Int`, which sums the integers in a value of an arbitrary datatype. The definition of function `sum` is very similar that of `size` and can be found in appendix B.2.

The first argument of function `cata` is a function of type `(FunctorOf d) a b -> b`. This kind of functions can only be constructed by means of functions `inn`, `out`, `fmap`, and functions defined by means of the `polytypic` construct (like `fsize`). In all these cases the resulting function is also polytypic. If we want to use `cata` on a specific datatype we must use a special syntactic construct for the argument to `cata`: {`ci -> ei`}.

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

```
data BoolExp a =  Con a
```

```
                | Not (BoolExp a)
                | And (BoolExp a) (BoolExp a)
                | Or  (BoolExp a) (BoolExp a)
-- FunctorOf BoolExp = Par + Rec + Rec*Rec + Rec*Rec

eval  :: BoolExp Bool -> Bool
eval  =  cata { Con  ->  id
                Not  ->  not
                And  ->  (&&)
                Or   ->  (||) }
```

This evaluation function is an example of a function that cannot be made polytypic: The functor for `BoolExp a` 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 of an expression type produces a semantics for that type!

Function `fmap`

Consider the function `fmap`, the definition of a functor on functions:

```
polytypic fmap :: (a -> c) -> (b -> d) -> f a b -> f c d
  = \p r -> case f of
              g + h    ->  (fmap p r) -+- (fmap p r)
              g * h    ->  (fmap p r) -*- (fmap p r)
              Empty    ->  id
              Par      ->  p
              Rec      ->  r
              d @ g    ->  pmap (fmap p r)
              Const t  ->  id

(-+-) :: (a -> c) -> (b -> d) -> Either a b -> Either c d
(-*-) :: (a -> c) -> (b -> d) -> (a,b) -> (c,d)
```

If `fmap` is used on an element of type *FList a b*, then definition (3.4) of *fmap* is generated, and if `fmap` is used on an element of type *FTree a b*, then definition (3.7) of *fmap* is generated. Function `fmap` and function `pmap`, which is used in the `d @ g` case, are mutually recursive.[9] The functions `(-+-)` and `(-*-)` are implementation of the operators $+$ and $\times$ from equations 3.1 and 3.3.

[9]This recursive dependence is only in the generation phase. The generated instances for `fmap` and `pmap` will not be mutually recursive as they will be instantiated on different types.

As an example we give the instance of function `fmap` for the functor *FRose*:

```
fmap_FRose = \p r -> (fmap_Par p r) -*- (fmap_(List@Rec) p r)
           = \p r -> p -*- (pmap_List (fmap_Rec p r))
           = \p r -> p -*- (pmap_List r)
```

3.3.2 Type checking polytypic definitions

We want to be sure that functions generated by polytypic functions are type correct, so that no run-time type errors occur. For that purpose the polytypic programming system type checks definitions of polytypic functions. This section briefly discusses how to type check polytypic functions, the details of the type checking algorithm can be found in [39] and in section 2.3.

In order to type check inductive definitions of polytypic functions the system has to know the type of the polytypic function: higher-order unification is needed to infer the type from the types of the functions in the `case` branches, and general higher-order unification is undecidable.[10] This is the reason why inductive definitions of polytypic functions need an explicit type declaration. Given an inductive definition of a polytypic function

```
polytypic foo :: ... f ...
  = case f of
      g + h -> bar
       ...
```

where `f` is a functor variable, the rule for type checking these definitions checks among others that the declared type of function `foo`, with `g + h` substituted for `f`, is an instance of the type of expression `bar`. For all of the expressions in the branches of the `case` it is required that the declared type is an instance of the type of the expression in the branch with the left-hand side of the branch substituted for `f` in the declared type. The expression `g + h` can be seen as a 'hungry' type synonym (in desperate need of two type expression), so when we have substituted `g + h` (or any of the other abstract type expressions) for `f` in the type of `foo` we must also rewrite the expression using the following type synonym definitions:

```
type (g + h) a b  =  Either (g a b) (h a b)
type (g * h) a b  =  (g a b, h a b)
type Empty a b    =  ()
type Par a b      =  a
type Rec a b      =  b
```

[10]See Ruehr's thesis [77] for a discussion on when it is possible to infer types for polytypic functions.

```
type (d @ g) a b  =  d (g a b)
type Const t a b  =  t
```

As an example we take the case `g * h` in the definition of `fsize`.

```
polytypic fsize :: f a Int -> Int
  = case f of
      ...
      g * h     ->  \(x,y) -> fsize x + fsize y
      ...
```

The type of the expression `\(x,y) -> fsize x + fsize y` is `(f a Int, g a Int) -> Int`. Substituting the functor to the left of the arrow in the case branch, `f * g`, for `f` in the declared type `f a Int -> Int` gives `(f * g) a Int -> Int`, and expanding out the type synonyms gives `(f a Int, g a Int) -> Int`. This type is equal to (and hence certainly an instance of) the type of the expression to the right of the arrow in the case branch, so this part of the polytypic function definition is type correct. For more examples, see section 2.3.

3.3.3 More examples of polytypic functions

In this section we define three polytypic functions: `flatten`, `propagate` and `zip`, which are useful in many situations.

Function `flatten`

Function `flatten` takes a value `v` of a datatype `d a`, and returns the list containing all elements of type `a` occurring in `v`. For example, `flatten (Bin (Bin (Leaf 17) (Leaf 3)) (Leaf 8))` equals `[17,3,8]`. This function is the central function in Jay's [42] representation of values of shapely types: a value of a shapely type is represented by its contents, obtained by flattening the value, and its structure, obtained by removing all contents of the value.[11]

```
flatten  :: Regular d => d a -> [a]
flatten  =  cata fl

polytypic fl :: f a [a] -> [a] =
  case f of
    g + h   -> either fl fl
```

[11] The polytypic function `structure :: Regular d => d a -> d ()` has a very simple definition: `structure = pmap (const ())`.

```
    g * h   -> \(x,y) -> fl x ++ fl y
    Empty   -> \x -> []
    Par     -> \x -> [x]
    Rec     -> \x -> x
    d @ g   -> concat . flatten . pmap fl
    Const t -> \x -> []
```

This definition of `flatten` returns a preorder traversal of a datatype, but other variants are also possible.

Variants of function `fl` are the functions `fl_rec`, which returns the list of elements that occur at the recursive (second argument) position, `fl_par`, which returns the list of elements that occur at the parameter (first argument) position, and `fl_all`, which returns the list of elements that occur at both the recursive and the parameter position.

```
fl_par ::  f a b -> [a]
fl_rec ::  f a b -> [b]
fl_all ::  f a a -> [a]
fl_par  =          fl . fmap id  nil
fl_rec  = concat . fl . fmap nil (singleton.singleton)
fl_all  =          fl . fmap id  singleton

singleton x = [x]
nil x = []
```

Function `zip`

Haskell's `zip` function takes two lists, and pairs the elements at corresponding positions. If one list is longer than the other the extra elements are ignored. The polytypic version of function `zip`, called `pzip`, zips two elements of datatype `d a`; for example, `pzip (Bin (Leaf 1) (Leaf 3)) (Bin (Leaf '7') (Leaf '8'))` equals the tree `Bin (Leaf (1, '7') ) (Leaf (3, '8') )`. Since it is impossible to zip two elements that have different structure, `pzip` returns a value of the form `Just x` if the elements have the same shape, and `Nothing` otherwise. This implies that we need some functions manipulating values with occurrences of `Maybe` values. Functions `resultM` and `bindM` are the functions from the `Maybe`-monad.

```
resultM    :: a -> Maybe a
resultM x  =  Just x

bindM      :: Maybe a -> (a -> Maybe b) -> Maybe b
bindM x f  =  maybe Nothing f x
```

```
(>>=)  =  bindM

(@@)        :: (a -> Maybe b) -> (c -> Maybe a) -> c -> Maybe b
(g @@ f) a  =  f a >>= g
```

where `maybe :: b -> (a -> b) -> Maybe a -> b` is an implementation of the catamorphism on the datatype `Maybe a`. Function `propagate` is a polytypic function that propagates occurrences of `Nothing` in a element of a datatype to top level.[12] For example, if we apply `propagate` to `Bin (Leaf Nothing) (Leaf (Just 1))` we obtain `Nothing`.

```
propagate :: Regular d => d (Maybe a) -> Maybe (d a)
propagate  =  cata (mapMaybe inn . fprop)

polytypic fprop :: f (Maybe a) (Maybe b) -> Maybe (f a b)
  = case f of
      g + h     ->  sumprop   . (fprop -+- fprop)
      g * h     ->  prodprop  . (fprop -*- fprop)
      Empty     ->  resultM
      Par       ->  id
      Rec       ->  id
      d @ g     ->  propagate . (pmap fprop)
      Const t   ->  resultM

sumprop  :: Either (Maybe a) (Maybe b) -> Maybe (Either a b)
sumprop  =  mapMaybe Left `either` mapMaybe Right

prodprop :: (Maybe a,Maybe b) -> Maybe (a,b)
prodprop (Just x,Just y)  =  Just (x,y)
prodprop _                =  Nothing

mapMaybe f = maybe Nothing (Just.f)
```

where `mapMaybe` is an implementation of the map function on the datatype `Maybe a`.

Function `pzip` first determines whether or not the outermost constructors are equal by means of the auxiliary function `fzip`, and then applies `pzip` recursively to the children of the argument.

```
pzip :: Regular d => (d a,d b) -> Maybe (d (a,b))
```

[12] Function `propagate` is an instance of the more general function `thread :: (Regular d, Monad m) => d (m a) -> m (d a)`, see appendix B.6.

```
pzip  = ((resultM.inn) @@ (fprop.fmap resultM pzip) @@ fzip )
      . (out -*- out)

polytypic fzip :: (f a b,f c d) -> Maybe (f (a,c) (b,d))
  = case f of
      g + h     ->  (sumprop   . (fzip -+- fzip)) @@ sumzip
      g * h     ->  (prodprop  . (fzip -*- fzip)) @@ prodzip
      Empty     ->  const (resultM ())
      Par       ->  resultM
      Rec       ->  resultM
      d @ g     ->  (propagate . (pmap fzip    )) @@ pzip
      Const t   ->  constzip

sumzip ::(Either a b,Either c d)-> Maybe (Either (a,c) (b,d))
sumzip (x,y)  =  case (x,y) of
                   (Left s,Left t)  ->  resultM (Left (s,t))
                   (Right s,Right t)->  resultM (Right (s,t))
                   _                ->  Nothing

prodzip :: ((a,b),(c,d)) -> Maybe   ((a,c),(b,d))
prodzip    ((x,y),(s,t)) =  resultM ((x,s),(y,t))

constzip :: (t,t) -> Maybe   t
constzip    (x,y) =  resultM x
--constzip (x,y) = if x==y then resultM x else Nothing
```

Note that when `fzip` is applied to a pair of values that are represented by means of the `Const` functor, we have (arbitrarily) chosen to return the first of these. Ideally we should return `Nothing` if the constants are not equal, but there are technical problems with using an equality test in this position. Function `pzip` is a strict function. To obtain a nonstrict version of function `pzip`, we define a polytypic version of function `zipWith`, called `pzipWith`.

Function `pzipWith`

Function `pzipWith` is more general than its specialised version on the datatype of lists. It takes the three functions `ins`, `op`, and `fail`, and a pair of values `(x,y)`. If `x` and `y` have the same outermost constructor, function `pzipWith` is applied recursively to the fzipped children of the constructors. The pairs at the parameters are combined with function `op`, and the result of these applications is combined with function `ins`. If `x` and `y` have different outermost constructors, `fail` computes the result from `x` and `y`.

```
pzipWith :: Regular d =>
              ((FunctorOf d) c e -> e) ->
              ((a,b)->c) ->
              ((d a,d b)->e) ->
              (d a,d b) -> e
pzipWith ins op fail (x,y)  =
    maybe (fail (x,y))
          (ins.fmap op (pzipWith ins op fail))
          (fzip (out x,out y))
```

The expression `pzipWith inn` has type `Regular d => ((a,b) -> c) -> ((d a, d b) -> d c) -> (d a, d b) -> d c`, and is the natural generalisation of Haskell's function `zipWith`. We can now give an alternative definition of function `pzip`.

```
pzip :: Regular d => (d a, d b) -> Maybe (d (a, b))
pzip = pzipWith (mapMaybe inn.fprop) resultM (const Nothing)
```

This function is still not as lazy as the Haskell `zip` for lists. If we try to zip two infinite lists the program will loop even if we request only the first few elements of the result. This is due to the use of a `Maybe` type in the result: the zip function can not determine whether or not the structures are equal until the last element has been checked. This can be worked around be using the result type `(d (a,b), Bool)` where `(x,True)` corresponds to `Just x` and `(y,False)` corresponds to `Nothing`:

```
pzip' :: Regular d => (d a, d b) -> (d (a,b), Bool)
pzip' p = ( pzipWith inn id (const undefined) p
          , pzipWith (and.fl_all) (const True) (const False) p)
```

This version is lazy in the first component of the resulting pair (but of course still strict in the second component).

3.3.4 Haskell's `deriving` construct

In Haskell there is a possibility to *derive* the functions in some classes for most datatypes. For example, writing

```
data Tree a = Leaf a | Bin (Tree a) (Tree a) deriving (Eq, Ord)
```

generates instances of the class `Eq` (containing `(==)` and `(/=)`), and the class `Ord` (containing `compare`, `(<)`, `(<=)`, `(>=)`, `(>)`, `max`, and `min`) for the datatype `Tree a`. There are six classes that can be derived in Haskell, besides the two classes above: `Bounded`, functions for obtaining the bounds of bounded datatypes, `Enum`, enumerating values of a datatype, and `Read` and `Show`[13], functions for printing and parsing values of a datatype. The functions in the derived classes are typical examples of polytypic functions. In fact, one reason for developing a language with which polytypic functions can be written was to generalise the rather *ad hoc* `deriving` construct. All functions in the classes that can be derived can easily be written as polytypic functions, except for the functions in the classes `Show` and `Read`. To be able to write the functions in the classes `Show` and `Read` as polytypic functions we have to introduce a separate built-in function that gives access to constructor names.[14]

In this section we will define the polytypic versions of functions `(==)` (written `(!==!)`) and `compare` by means of which all functions in the classes `Eq` and `Ord` are defined. Furthermore, we will give the function with which the constructors of a datatype can be printed. The definitions of the polytypic versions of the functions in the other classes that can be derived can be found in PolyP's libraries.

Function `(!==!)`

Equality on datatypes is defined as follows. Function `(!==!)` zips its arguments, flattens the result to a list of pairs, and checks that each pair of values in this list consists of equal values. The arguments have equal shape if `pzip` returns a value of the form `Just z` for some `z`, and the arguments have equal contents if all pairs in the list of pairs we obtain by flattening `z` consist of equal elements.

```
(!==!) :: (Regular d, Eq a) => d a -> d a -> Bool
x !==! y  =  maybe False
                   (all (uncurry (==)) . flatten)
                   (pzip (x,y))
```

The laziness behaviour of this definition is not the same as in Haskell: It first compares the structures (with `pzip`) and only then the contents. This means that the comparison `[1..] !==! [2..]` never terminates. This can be remedied if we define function `!==!` by means of function `pzipWith`:

```
(!==!)  :: (Regular d, Eq a) => d a -> d a -> Bool
l !==! r = pzipWith ins op fail (l,r)
      where ins  =  and . fl_all
            op   =  uncurry (==)
            fail =  \_ -> False
```

[13] The class `Text` was split into `Read` and `Show` in Haskell 1.3.

[14] We also need access to the fixities of the constructors, but we have not implemented that yet.

This version of `(!==!)` compares the structure and the values at the top level before looking at the subtrees. With the new definition the calculation of `[1..] !==! [2..]` terminates with the value `False`.

Function `compare`

The definition of function `compare` is more complicated than the definition of function `(!==!)`. The Haskell report defines `x <= y` for an arbitrary datatype as follows: the outermost constructor of `x` appears earlier in the datatype definition than the outermost constructor of `y`, or `x` and `y` have the same outermost constructor, and the children of `x` are lexicographically smaller than or equal to (under the ordering `(<=)`) the children of `y`. This implies that we need to be able to obtain the position of a constructor in its datatype definition. For this purpose we introduce the polytypic function `fcnumber`, which given a value of type `f a b` (usually obtained by means of function `out`, so the `b` is often `d a`), returns the position of the constructor in the definition of the datatype corresponding to `d a`.

```
polytypic fcnumber :: f a b -> Int
  = case f of
      g + h  ->  fcnumber `either` ((1+) . fcnumber)
      _      ->  const 0
```

Here we use the fact that `+` is right-associative. Function `compare` is now defined as follows. It first fzips its arguments. If the arguments cannot be fzipped (i.e., `fzip` returns `Nothing`), function `order` determines which of the arguments comes first in the definition of the datatype, and returns `LT`, `EQ`, or `GT` accordingly. If the arguments of `compare` can be fzipped, functions `order` and `compare` are applied recursively, and the results are combined by flattening the result, and folding from left to right until we encounter a value unequal to `EQ`.

```
data Ordering  =  LT | EQ | GT

compare    :: Ord a => d a -> d a -> Ordering
compare x y = maybe (order (fcnumber (out x),fcnumber (out y)))
                    (foldr op EQ
                    .fl_all
                    .fmap order (uncurry compare) )
                    (fzip (out x , out y))
  where order (x,y)  |  x < y   =  LT
                     |  x == y  =  EQ
                     |  x > y   =  GT
        op EQ y  =  y
        op x  y  =  x
```

The classes Show and Read

The classes Show and Read contains functions for printing values of a datatype. To be able to define the functions in these classes as polytypic functions, we have to introduce a separate built-in function that gives access to constructor names. This function is called fconstructor_name, and it is used in function constructor_name.

```
fconstructor_name  ::  f a b -> String

constructor_name  :: d a -> String
constructor_name  =  fconstructor_name . out
```

For example, constructor_name (Cons 1 Nil) equals "Cons".

Using constructor_name a polytypic show function can be written as a catamorphism, but we have not worked out all details yet.

```
show :: (Regular d, Show a) => d a -> String
show = cata fshow . pmap show

polytypic fshow :: f String String -> String
   ...
```

3.4 Polytypic term rewriting

Rewriting systems is an area in which polytypic functions are useful. A rewriting system is an algebra together with a set of rewriting rules. In a functional language, the algebra is represented by a datatype, and the rewriting rules can be represented as a list of pairs of values of the datatype extended with variables.

In this section we will define a function rewrite which takes a set of rewrite rules of some datatype extended with variables, and a value of the datatype without variables, and rewrites this value by means of the rewriting rules using the parallel-innermost strategy, until a normal form is reached. We use the parallel-innermost strategy [53] because it is relatively easy to implement function rewrite as an efficient function when using this strategy. Function rewrite does not check if the rewriting rules in its first argument are normalising, so it will not terminate for certain inputs.

The other main function defined in this section is a function that determines whether a set of rewriting rules is normalising. This function is based on a well-known method of recursive path orderings, as developed by Dershowitz on the basis of a theorem of Kruskal, see [53]. The results in this section are for a large part based on results from Berglund [8], in which more applications of polytypic functions in rewriting systems can be found.

We will assume there is a polytypic unification function `unify` that takes a pair a terms and gives a substitution (see chapter 4 for an implementation), and a function `appSubst` for applying this substitution to a term.

The code in this section was developed using a predecessor of PolyP with slightly different notation and has not been translated to the current notation. The differences mainly appear in the types[15] and they are not essential for the algorithms.

3.4.1 A function for rewriting terms

Function `rewrite` takes two arguments with different but related types: a set of rewrite rules of a datatype extended with variables, and a value of the datatype without variables. To express this relation between the types of the arguments we have to make the presence of variables visible in the type. Let `Mu f a` be an arbitrary datatype. Then we can extend this datatype with variables (represented by integers) by adding an extra component to the sum represented by `f`: `Mu (Const Int + f) a`. Thus we obtain the following type for `rewrite`:

```
type MuVar f a  =  Mu (Const Int + f) a
type Rule f a   =  (MuVar f a , MuVar f a)

rewrite  ::  [Rule f a] -> Mu f a -> Mu f a
```

Later we will convert values of `Mu f a` to values of type `MuVar f a` and vice versa. Functions `toMuVar` and `fromMuVar` take care of these type conversions. Function `toMuvar` injects values of an arbitrary datatype into values of the datatype extended with variables. The resulting value does not contain variables. Function `fromMuVar` translates a variable-free value of the datatype extended with variables to the datatype without variables. This function fails when it is applied to a value that does contain variables.

```
toMuVar  :: Mu f a -> MuVar f a
toMuVar  =  cata ftoMuVar

polytypic ftoMuVar :: f a (MuVar f a) -> MuVar f a
  = case f of
      _  ->  \x -> inn (Right x)
```

[15]The main difference is that types of polytypic functions are written `pf :: ... Mu f a ...` instead of PolyPs `pf :: Regular d => ... d a ...`.

```
fromMuVar  :: MuVar f a -> Mu f a
fromMuVar  =  cata ffromMuVar

polytypic ffromMuVar :: (Const Int + f) a (Mu f a) -> Mu f a
  = case f of
      _  ->  \(Right x) -> inn x
```

We will define function `rewrite` in a number of stages. The first definition is a simple, clearly correct but very inefficient implementation of `rewrite`. This definition will subsequently be refined to a function with better performance.

A first definition of function `rewrite`

Function `rewrite` rewrites its second argument with the rules from its first argument until it reaches a normal form. So function `rewrite` is the fixed-point of a function that performs a single parallel-innermost rewrite step, function `rewrite_step`. The fixed-point computation is surrounded by type conversions in order to be able to apply the functions for unification given in chapter 4.

```
rewrite rs  =  fromMuVar . rewrite' rs . toMuVar

rewrite' rs  =  fp (rewrite_step rs)

fp f x  |  fx !==! x  =  x
        |  otherwise  =  fp f fx
  where fx = f x

rewrite_step  ::  [Rule f a] -> MuVar f a -> MuVar f a
```

Function `rewrite_step` is the main rewriting engine. Given a set of rules and a value `x`, it tries to rewrite all innermost redeces of `x`. This is achieved by applying `rewrite_step` recursively to `x`, and only rewriting the innermost redeces. At each recursive application function `rewrite_step` applies a function `innermost`. Function `innermost` determines if any of the children have been rewritten. Only if no child has been rewritten, it tries to reduce its argument. To determine whether or not one of the children has been rewritten, function `innermost` compares its argument with the original argument of function `rewrite_step`. The recursive structure of function `rewrite_step` is that of a `cata`, but it needs access to the original argument too. Such functions are called *paramorphisms* [63].

```
para      :: (f a b -> Mu f a -> b) -> Mu f a -> b
para h x  =  h (fmap id (para h) (out x)) x
```

```
rewrite_step rs  =  para (innermost rs)

innermost :: Eq a => [Rule f a] ->
                     (Const Int + f) a (MuVar f a) ->
                     MuVar f a -> MuVar f a
innermost rs x' x =
  if (inn x') !/=! x then inn x' else reduce rs x
```

Function unify is defined in chapter 4.

```
reduce :: Eq a => [Rule f a] -> MuVar f a -> MuVar f a
reduce []             t  =  t
reduce ((lhs,rhs):rs) t  =  case unify (lhs,t) of
                               Just s   ->  appSubst s rhs
                               Nothing  ->  reduce rs t
```

Function `rewrite` is extremely inefficient. For example, if we represent natural numbers with `Succ` and `Zero`, and we use the rewriting rules for `Zero`, `Succ`, `:+:`, and `:*:` given in the introduction, it takes hundreds of millions of (Gofer) reductions to rewrite the representation of 2^8 to the representation of 256. One reason why `rewrite` is inefficient is that in each application of function `rewrite_step` the argument is traversed top-down to find the innermost redeces. Another reason is that function `rewrite_step` performs a lot of expensive comparisons.

If we want to change function `rewrite` to rewrite terms using another innermost rewriting strategy, the only function that has to be rewritten is function `innermost`. All the variants of `innermost` are simple polytypic functions.

Avoiding unnecessary top-down traversals and comparisons

We want to obtain a function that rewrites a term in time proportional to the number of steps needed to rewrite the term. As a first step towards such a function, we replace the fixed-point computation by a double recursion. The double recursion avoids the unnecessary top-down traversals in search for the innermost redeces. The idea is to first recursively rewrite the children of the argument to normal form, and only then rewrite the argument itself.

For confluent and normalising term rewriting systems we have that first applying `rewrite'` to the subterms of the argument, and subsequently to the argument itself, gives the same result as applying function `rewrite'` to the argument itself.

```
rewrite' rs (inn x)  =  rewrite' rs (inn (fmap id (rewrite' rs) x))
```

It follows that function `rewrite'` can be written as a catamorphism, which uses function `rewrite'` in the recursive step. This version of function `rewrite` is called `rewritec`.

```
rewritec rs = cata frewrite
  where frewrite x = rewrite' rs (inn x)
```

Observe that in the recursive step, all subexpressions are in normal form. It follows that the only possible term that can be rewritten is the argument `inn x`. If `inn x` is a redex, then it is rewritten, and we proceed with rewriting the result. If `inn x` is not a redex, then `inn x` is in normal form. We adjust function `reduce` such that it returns `Nothing` if it does not succeed in rewriting its argument, and `Just x` if it does succeed with `x`.

```
rewritec rs  =  cata frewrite
  where
  frewrite x = maybe (inn x) (rewritec rs) (reduce rs (inn x))

reduce :: Eq a => [Rule f a] -> MuVar f a -> Maybe (MuVar f a)
reduce [] t             = Nothing
reduce ((lhs,rhs):rs) t = case unify (lhs,t) of
                            Just s   ->  Just (appSubst s rhs)
                            Nothing  ->  reduce rs t
```

This function rewrites 2^8 much faster than the first definition of function `rewrite`, but it is still far from linear in the number of rewrite steps.

Efficient rewriting

A source of inefficiency in function `rewritec` is the call to function `rewritec` in `frewrite`. If `reduce rs (inn x)` returns some expression `Just e`, `rewritec rs` is applied to `e`. When evaluating the expression `rewritec rs e` the whole expression `e` is traversed to find the innermost redeces, including all subterms which are known to be in normal form. For example, consider the expression `100 :*: 2`, where `2` and `100` abbreviate their equivalents written with `Succ` and `Zero`. Applying the second rule for `:*:`, this term is reduced to `(100 :*: 1) :+: 100`. Now, `rewritec rs` will traverse both subexpressions `100`, and find that they are in normal form, which we already knew. To avoid these unnecessary traversals, function `rewritec` is rewritten as follows. Instead of applying `rewritec rs` recursively to the reduced expression, we apply a similar function recursively to the right-hand side of the rule with which the expression is reduced. This avoids recursing over the expressions substituted for the variables in this rule, which are known to be in normal form. To define this function we use the polytypic version of function `zipWith`, called `pzipWith`. Function `pzipWith` is used in the definition of `frewrite` to zip the right-hand side of a rule with the expression obtained by substituting the appropriate expressions for the variables in this rule. This means that in case `pzipWith` encounters two arguments with a different outermost constructor, the left argument is a variable, and the right argument is an expression in normal form substituted for the variable. In that case we return the second argument. In case `pzipWith` encounters two arguments with the same outermost constructor, it tries to rewrite the zipped expression.

```
rewritec rs  =  cata frewrite
  where frewrite x = maybe (inn x) just (reduce rs (inn x))
        just = pzipWith frewrite fst snd

reduce :: Eq a => [Rule f a] -> MuVar f a ->
                  Maybe (MuVar f a , MuVar f a)
reduce [] t            = Nothing
reduce ((lhs,rhs):rs) t = case unify (lhs,t) of
                            Just s   ->  Just (rhs,
                                               appSubst s rhs)
                            Nothing  ->  reduce rs t
```

The resulting rewrite function is linear in the number of reduction steps needed to rewrite a term to normal form. It rewrites the representation of 2^8 into the representation of 256 with the rules given for `Zero`, `Succ`, `:+:`, and `:*:` in the introduction about 500 times faster than the original specification of function `rewrite`. This function can be further optimised by partially evaluating with respect to the rules; we omit these optimisations.

3.4.2 Normalising sets of rewriting rules

Termination of function `rewrite` can only be guaranteed if its argument rules are normalising. A set of rules is normalising if all terms are rewritten to normal form (i.e. cannot be rewritten anymore) in a finite number of steps. It is undecidable whether or not a set of rewriting rules is normalising (unless all rules do not contain variables), but there exist several techniques that manage to prove the normalising property for a large class of normalising rewriting rules. A technique that works in many cases is the method based on a well-known method of recursive path orderings, as developed by Dershowitz on the basis of a theorem of Kruskal, see [53]. In this section we will define a function `normalise` based on this technique.

```
normalise  ::  Eq a => [Rule f a] -> Bool
```

Note that if function `normalise` returns `False` for a given set of rules this does not necessarily mean that the rules are not normalising, it only means that function `normalise` did not succeed in constructing a witness for the normalising property of the rules.

The recursive path orderings technique

The recursive path orderings technique for proving the normalising property is rather complicated; it is based on a deep theorem from Kruskal. In this section we will see the technique in action; see [53] for the theory behind this technique.

A set of rules of type `[Rule f a]` is normalising according to the recursive path orderings technique if we can find an ordering on the constructors of the datatype `MuVar f a` such that each left-hand side of a rule can be rewritten into its right-hand side using a set of four special rules. These rules will be illustrated with the rewriting rules for `Zero`, `Succ`, `Add` and `Mul` given in the introduction:

```
Var 1 :++: VZero          ->  Var 1
Var 1 :++: VSucc (Var 2)  ->  VSucc (Var 1 :++: Var 2)
Var 1 :**: VZero          ->  VZero
Var 1 :**: VSucc (Var 2)  ->  (Var 1 :**: Var 2) :++: Var 1
```

We assume that the constructors of the datatype `VNumber` are ordered by `Var < VZero < VSucc < :++: < :**:`. The four rewriting rules with which left-hand sides have to be rewritten into right-hand sides are the following:

- Place a mark (denoted by an exclamation mark `!`) on top of a term.
- A marked value `x` with outermost constructor `c` may be replaced by a value with outermost constructor `c' < c`, and with marked `x`'s occurring at the recursive child positions of `c'`. For example, suppose `y` equals `!(Var 1 :++: VSucc (Var 2))`, then `y -> VSucc y`, since `VSucc < :++:`.
- A mark on a value `x` may be passed on to zero or more children of `x`. For example, the mark on `y` in the above example may be passed on to the subexpression `VSucc (Var 2)`, so `!(Var 1 :++: VSucc (Var 2)) -> Var 1 :++: !(VSucc (Var 2))`.
- A marked value may be replaced by one of its children occurring at the recursive positions. For example, `!(VSucc (Var 2)) -> Var 2`.

Each of the right-hand sides of the rules for rewriting numbers can be rewritten to its left-hand side using these rules. For example,

```
  Var 1 :**: VSucc (Var 2)
-> { Rule 1 }
  !(Var 1 :**: VSucc (Var 2))
-> { Rule 2 }
  !(Var 1 :**: VSucc (Var 2)) :++: !(Var 1 :**: VSucc (Var 2))
```

```
-> { Rule 4 }
  !(Var 1 :**: VSucc (Var 2)) :++: Var 1
-> { Rule 3 }
  (Var 1 :**: !(VSucc (Var 2))) :++: Var 1
-> { Rule 4 }
  (Var 1 :**: Var 2) :++: Var 1
```

It follows that the set of rules for rewriting numbers is normalising.

Function `normalise`

A naive implementation of a function `normalise` that implements the recursive path orderings technique computes all possible orderings on the constructors, and tests for each ordering whether or not each left-hand side can be rewritten to its corresponding left-hand side using the four special rules. If it succeeds with one of the orderings, the set of rewriting rules is normalising. Since the four special rules of the recursive path ordering technique themselves are not normalising this test might fail to terminate. To obtain a terminating function `normalise`, we implement a restricted version of the four special rules. Thus, function `normalise` does not fully implement the recursive path orderings technique, but it still manages to prove the normalising property for a large class of sets of rewriting rules.

```
normalise rules = or [all (l_to_r ord) rules | ord <- allords]

allords :: [Mu f a -> Int]
l_to_r  :: Eq a => (Mu f a -> Int) -> (Mu f a, Mu f a) -> Bool
```

Function `allords` generates all orderings, where an ordering is a function that given a value of the datatype returns an integer. Function `l_to_r` implements a restricted version of the four special rewrite rules.

Function `allords` is defined by means of two functions: function `perms`, which computes all permutations of a list, and function `fconstructors` which returns a representation of the list of all constructors of a datatype. The definition of function `perms` is omitted.

```
polytypic fconstructors :: [f a b]
  = case f of
      g + h  ->  [Left x  | x <- fconstructors] ++
                 [Right y | y <- fconstructors]
      _      ->  [undefined]

allords  =  map make_ord (perms fconstructors)
```

```
where make_ord l x = index (fcnumber (out x)) (map fcnumber l)
      index n (m:ms)  |  n == m     =  0
                      |  otherwise  =  1 + index n ms
      index n []      =  error "no index in list"
```

A straightforward optimisation of function `normalise` can be obtained by only generating those orderings that do not immediately fail given the argument `rules`. For example, any ordering on `VNumber` with `:**: < :++:` will immediately fail on account of the fourth rewriting rule, which requires `:++: < :**:`. We will not implement this optimisation here.

Finally, we have to implement function `l_to_r`. Given an ordering and a rewriting rule `(l,r)`, function `l_to_r` tries to rewrite `l` into `r`. Distinguish the following three cases:

- The outermost constructor of the right-hand side, `ocr`, is larger than the outermost constructor of the left-hand side, `ocl`, under the given ordering. In this case it is impossible to rewrite `l` into `r`, and function `l_to_r` returns `False`.
- `ocr` is smaller than `ocl` under the given ordering. In this case, function `l_to_r` computes the recursive components of the right-hand side. If there are no such, it checks that the right-hand side itself is a subexpression of the left-hand side. If there are recursive components, function `l_to_r` checks that all of these are subexpressions of the left-hand side. For this purpose we define function `subexpr`, which takes two arguments, and determines whether or not the second argument is a subexpression of the first argument. A subexpression of `x` does not have to be a consecutive part of `x`, for example, the tree `Bin (Leaf 3) (Leaf 2)` is a subexpression of the tree `Bin (Bin (Leaf 3) (Leaf 4)) (Leaf 2)`. On lists, subexpressions are usually called subsequences.

  ```
  subexpr :: Eq a => Mu f a -> Mu f a -> Bool
  subexpr l r  =
    pzipWith (and . fl_all)
             (uncurry (==))
             (\(x,y)-> (any (`subexpr` y) . fl_rec . out) x)
             (l,r)
  ```

- The outermost constructors are equal under the given ordering. In this case, function `l_to_r` fzips the children of the left-hand side and the right-hand side. It checks that all pairs of values appearing at the parameter position consist of equal values, and it checks that there exists at least one recursive position pair. Furthermore, for each pair of values `(l,r)` appearing at a recursive position, `l_to_r ord (l,r)` has to hold.

We obtain the following definition of function `l_to_r`.

```
l_to_r ord (l,r)
  | ocl < ocr  = False
  | ocl > ocr  = let x = fl_rec (out r) in
     if null x then subexpr l r else all (subexpr l) x
  | ocl == ocr =
     maybe undefined
           (\x -> parEq x && all' (l_to_r ord) (fl_rec x))
           (fzip (out l , out r))
  where ocl = ord l
        ocr = ord r

all' p []  =  False
all' p xs  =  all p xs

parEq  :: Eq a => f (a,a) b -> Bool
parEq  =  all (uncurry (==)) . fl_par
```

3.5 Conclusions and future work

This paper introduces polytypic programming: programming with polytypic functions. Polytypic functions are useful in applications where programs are datatype independent in nature. Typical example applications of this kind are unification (see chapter 4) and the rewriting system examples discussed in this paper. Polytypic functions are also useful in the evolutionary process of developing complex software. Here, one important feature of polytypic functions is the fact that they adapt automatically to changing structure.

The code generated for programs containing polytypic functions is usually only slightly less efficient than datatype-specific code. In fact, polytypic programming encourages writing libraries of generally applicable applications, which is an incentive to write efficient code, see for example our library of rewriting functions.

The polytypic programming system PolyP is still under development. In the future PolyP will be able to handle mutual recursive datatypes, datatypes with function spaces, and datatypes with multiple arguments.

Polytypic programming has many more applications than we have described in this paper. A whole range of applications can be found in adaptive object-oriented programming. Adaptive object-oriented programming is a kind of polytypic programming, in which constructor names play an important role. For example, Palsberg *et al.* [73] give a program that for an arbitrary datatype that contains the constructor names `Bind` and `Use`, checks that no variable is used before it is bound. This program is easily translated into a polytypic function, but we have yet to investigate the precise relation between polytypic programming and adaptive object-oriented programming.

Acknowledgements

During the two years we worked on polytypic programming we benefited from stimulating discussions and comments from Patrik Berglund, Graham Hutton, Mark Jones, Erik Meijer, Oege de Moor, Rinus Plasmeijer, Fritz Ruehr, Tim Sheard, Doaitse Swierstra, Phil Wadler, and Arjan van IJzendoorn. Oege de Moor, Graham Hutton and Arjan van IJzendoorn helped implementing predecessors of PolyP.

Chapter 4

Polytypic unification

Polytypic unification[1]

Patrik Jansson and Johan Jeuring
Chalmers University of Technology and University of Göteborg
S-412 96 Göteborg, Sweden.
E-mail: {patrikj,johanj}@cs.chalmers.se

Abstract

Unification, or two-way pattern matching, is the process of solving an equation involving two first-order terms with variables. Unification is used in type inference in many programming languages and in the execution of logic programs. This means that unification algorithms have to be written over and over again for different term types.
Many other functions also make sense for a large class of datatypes; examples are pretty printers, equality checks, maps etc. They can be defined by induction on the structure of user-defined datatypes. Implementations of these functions for different datatypes are closely related to the structure of the datatypes. We call such functions polytypic.
This paper describes a unification algorithm parametrised on the type of the terms and shows how to use polytypism to obtain a unification algorithm that works for all regular term types.

[1]Submitted for publication, 1997

4.1 Introduction

In simple pattern matching, a pattern (a string containing wild cards) is matched with a normal string to determine if the string is an instance of the pattern. This can be generalised in at least two directions; we can allow the second string to contain wild cards too, thus making the matching symmetric, or we can allow more complicated terms than strings. By combining these two generalisations we obtain unification. A unification algorithm tries to find a *most general unifier* (*mgu*) of two terms. The most general unifier of two terms is the smallest substitution of terms for variables such that the substituted terms become equal.[1] Use of unification is widespread; it is used in type inference algorithms, rewriting systems, compilers, etc [54].

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 specialised version of the algorithm for this term type. This paper describes a functional unification program that works for all regular term types. The program is an example of a *polytypic function* [44].

Function `length :: List a -> Int`, which counts the number of occurrences of `a`'s in a list, and the similar function `numOfNodes :: Tree a -> Int`, which counts the number of occurrences of `a`'s in a tree are both instances of a more general function `size :: d a -> Int`. Function `size` is not only polymorphic in `a`, but also in the type constructor `d`. In the same way we can generalise the function `map :: (a -> b) -> List a -> List b` into a function `map :: (a -> b) -> d a -> d b`, so that it too works for trees and other similar datatypes. We call such functions polytypic functions. For an introduction to the basic ideas of polytypic functions see [46] and for a more theoretical treatment of polytypism [9] and [71].

In this paper we show that

- by parametrising the unification algorithm by the datatype for terms, we can separate the core of the algorithm from the parts depending on the specific datatype, and
- by abstracting away from the type constructor dependence in the datatype dependent part we obtain a polytypic program.

Thus we have *one* implementation of unification that works for many different term types leaving the specialisation to the computer.

The core of the unification algorithm is written in Haskell (using the Gofer interpreter [50]) and the polytypic part in PolyP [39]. The full code is available from `http://www.cs.chalmers.se/~patrikj/unify/`.

[1] If two first order terms are unifiable, their *mgu* is unique [75].

```
class Children t   where  children  ::  t → [t]
class VarCheck t   where  varCheck  ::  t → Maybe Var
class TopEq t      where  topEq     ::  t → t → Bool

class (VarCheck t, TopEq t, Children t) ⇒ Term t

class Subst s t    where  idSubst   ::  s t
                          modBind   ::  (Var,t) → s t → s t
                          lookup    ::  Var → s t → Maybe t
```

Figure 4.1: Terms and substitutions

4.2 Unification

In this section we will specify and implement a functional unification algorithm. 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 constants. Since 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 substitution $\sigma = \{x \mapsto g(y, a), y \mapsto f(a, b)\}$, the original pair of terms is also unified by applying the substitution σ, yielding the unified term $f(g(f(a, b), a), f(a, b))$.

4.2.1 Terms

In the unification literature, a term is usually defined as either a variable or an application of a constructor to zero or more terms. (*Var* is a set of variables and *Con* is a set of constructor constants.)

$$T \ ::= \ v \mid c(T_1, \ldots, T_{arity(c)}), \quad v \in \mathit{Var}, \quad c \in \mathit{Con}$$

We instead focus on the three properties of the type of terms we need to define unification. We need to know

- whether or not a term is a variable, and if it is, which variable;
- the immediate subterms of a term;
- when two terms are top-level equal. (Usually 'top level equal' means 'equal outermost constructors'.)

We define one type class for each of these properties and define the class of terms to be the intersection of these three classes (see figure 4.1). As an example, the instances for the type T above are shown in figure 4.2.

```
data T = V Var | App Con [T]

instance Children T where   children (V v)       = []
                            children (App c ts)  = ts

instance VarCheck T where   varCheck (V v) = Just v
                            varCheck  _    = Nothing

instance TopEq T    where   topEq (App c l) (App c' l') = c == c'
                            topEq (V v)     (V w)       = v == w
                            topEq  _         _          = False
instance Term T
```

Figure 4.2: T is an instance of `Term`.

4.2.2 Substitutions

A substitution is a mapping from variables to terms leaving all but a finite number of variables unchanged. We define a class of substitutions parametrised on the type of terms[2] by the three functions `lookup`, `modBind` and `idSubst`. The call `lookup v s` looks up the variable `v` in the substitution `s` (giving `Nothing` if the variable is unchanged by `s`), `modBind (v,t) s` modifies the substitution `s` to bind `v` to `t` (leaving the bindings for other variables unchanged) and `idSubst` is the identity substitution

A unifier of two terms is a substitution that makes the terms equal. A substitution `s` is at least as general as a substitution `s'` if and only if `s'` can be factored by `s`, i.e. if there exists a substitution `r` such that `s'` $=$ `r . s`, where we treat substitutions as functions. We want to define a function that given two terms finds the most general substitution that unifies the terms or, if the terms are not unifiable, reports this.

4.2.3 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 given as an extra argument. The unification algorithm starts with the identity substitution, traverses the terms and tries to update the substitution (as little as possible) while solving the constraints found. If this succeeds 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.

[2]This parametrisation is not essential for the results in the paper, we could just as well have used some specific type for substitutions instead.

```
unify  ::  (Term t,Subst s t) => t -> t -> Maybe (s t)
unify' ::  (Term t,Subst s t) => t -> t -> s t -> Maybe (s t)

unify  tx ty = unify' tx ty idSubst
unify' tx ty = uni (varCheck tx,varCheck ty) where
  uni (Nothing,Nothing) | topEq tx ty = uniTerms tx ty
                        | otherwise   = fail
  uni (Just i, Just j ) | i == j      = ok
  uni (Just i, _      )               = i |-> ty
  uni ( _,     Just j )               = j |-> tx

  uniTerms x y = threadList (zipWith unify' (children x) (children y))

(|->) ::  (Term t, Subst s t) => Var -> t -> s t -> Maybe (s t)
(i |-> t) s = if occursCheck i s t then fail s
              else case lookup i s of
                     Nothing -> ok (modBind (i,t) s)
                     Just t' -> mapMaybe (modBind (i,t)) (unify' t t' s)
```

Figure 4.3: The core of the unification algorithm

- If neither term is a variable we have two sub-cases; either the constructors of the terms are different (that is the terms are not top level equal) and the unification fails, or the constructors are equal and we unify all the children pairwise.
- If both terms are variables and the variables are equal we succeed without changing the substitution. (If the variables are not equal the case below matches.)
- If one of the terms is a variable we try to add to the substitution the binding of this variable to the other term. 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 4.3 using the auxiliary functions in figure 4.4. We use the following functions and types from Haskell 1.3: functions `return` and `(>>=)` for monad operations, the type `Maybe a` and its catamorphism `maybe` and the type `Either` with catamorphism `either`.

To use this unification algorithm on some term type `T` we must make `T` an instance of the class `Term` by defining the three functions `children`, `varCheck` and `topEq`. Traditionally these instances would be handwritten for the type `T` and when we need unification on a different type we would need new instances.

```
occursCheck :: (VarCheck t,Children t,Subst s t)
                => Var -> s t -> t -> Bool
occursCheck i s t = i `elem` reachlist s (vars t)
  where
    reachlist s l = l ++ concat (map (reachable s) l)
    reachable s v = reachlist s (maybe [] vars (lookup v s))

vars   :: (Children t,VarCheck t) => t -> [Var]
vars t =  [ v | Just v <- map varCheck (subTerms t)]

subTerms :: Children t => t -> [t]
subTerms t = t : concat (map subTerms (children t))

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

mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f = maybe Nothing (Just.f)

ok, fail  :: a -> Maybe a
ok        = Just
fail      = const Nothing
```

Figure 4.4: Auxiliary functions in the unification algorithm

But we can do better than that; by making these functions polytypic we get one description that works for all term types.

4.3 Polytypic unification

A polytypic function is a function parametrised on type constructors. Polytypic functions are defined either by induction on the structure of user-defined datatypes or defined in terms of other polytypic (and possibly non-polytypic) functions. To define polytypic functions we use the Haskell extension PolyP [39].

To make the unification algorithm polytypic we define `children`, `topEq` and `varCheck` for all term types `d a`, i.e. we express them as polytypic functions. In the following subsections we will describe the polytypic functions we need for unification and how they are expressed in PolyP.

4.3.1 Polytypic notation

To abstract away from the specific type constructors we view all datatypes as fix-points of functors, and extend the type language to include functor building blocks. Functors are built up from the constants `Par` for the parameter, `Rec` for the recursive component and `Const t` for constant types (`Int`, `Bool` etc.), `Empty` and the combinator `*` for products and `+` for alternatives. See figure 4.5 for some examples. With a recursive datatype as a fix-point, `inn` and `out` are the fold and unfold isomorphisms showing $d\ a \cong F_d\ a\ (d\ a)$.

```
inn  ::  d a ← (FunctorOf d) a (d a)
out  ::  d a → (FunctorOf d) a (d a)
```

To construct values of type `d a` we use `inn`, which effectively combines all the constructors of the datatype in one function, and conversely, to deconstruct values of type `d a` we use `out` instead of pattern matching. PolyP defines `inn` and `out` for all regular[3] datatypes.

4.3.2 Function `children`

Function `children :: Children t => t -> [t]` returns the immediate subterms of a term. We find these subterms by unfolding the recursive type one level, using `out`, and listing the recursive components with `fl_rec`:

```
instance Regular d => Children (d a) where
  children = fl_rec . out
```

Function `fl_rec :: f a b -> [b]` takes a value `v` of type `f a b`, and returns the list containing all elements of type `b` occurring at the top level in `v`. The

[3]A datatype `d a` is regular if it contains no function spaces, and if the arguments of the type constructor `d` are the same on the left- and right-hand side of its definition.

```
data Tree a    = Null      | Node (Tree a)   a   (Tree a)
-- FunctorOf Tree = Empty +         Rec  * Par *  Rec

data Exp a     = Val a   | Neg (Exp a) | Add (Exp a) (Exp a)
-- FunctorOf Exp  = Par +      Rec    +       Rec  *  Rec

data Type a    = TyVar Var     | BaseType a  | Apply (Type a) (Type a)
-- FunctorOf Type = Const Var +         Par +          Rec  *  Rec
```

Figure 4.5: Datatypes and functors.

polytypic definition of `fl_rec` is given in figure 4.6. (The subscripts indicating the type are included for readability and are not part of the definition.)

```
polytypic fl_rec_f :: f a b -> [b]
  = case f of
      g + h    ->  either fl_rec_g fl_rec_h
      g * h    ->  \(x,y) -> fl_rec_g x ++ fl_rec_h y
      Empty    ->  nil
      Par      ->  nil
      Rec      ->  singleton
      Const t  ->  nil

nil ::  a -> [b]
nil x = []

singleton ::  a -> [a]
singleton x = [x]
```

Figure 4.6: The polytypic `fl_rec` function.

A variant of function `fl_rec` is the function `fl_par :: f a b -> [a]`, which given a value `v` of type `f a b` returns the list of elements of type `a` in `v`. The definition of `fl_par` is similar to that of `fl_rec` and is omitted.

4.3.3 Function `topEq`

Function `topEq :: TopEq t => t -> t -> Bool` compares the top level of two terms for equality. This equality check is done in two steps; first the top level structures of the terms are compared, and then the top level data values are compared for equality.

```
instance (Regular d, Eq a) => TopEq (d a) where
  topEq t t' = case fzip (out t,out t') of
```

```
                    Nothing -> False
                    Just p  -> all (uncurry (==)) (fl_par p)
```

The structure comparison is performed by the polytypic function `fzip :: (f a b, f c d) -> Maybe (f (a,c) (b,d))`, a generalisation of the Haskell `zip` function. Function `fzip` takes a pair of structures to `Just` a structure of pairs if the structures have the same shape and `Nothing` otherwise. The definition of `fzip` is omitted.

4.3.4 Function `varCheck`

Function `varCheck :: VarCheck t => t -> Maybe Var` checks if a term is a variable. A polytypic `varCheck` must recognise 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`.

```
instance Regular d => VarCheck (d a) where
  varCheck = fvarCheck . out

polytypic fvarCheck :: f a b -> Maybe Var
  = case f of
      (Const Var) + g   ->   either ok fail
      f                 ->   fail
```

We have now made all regular datatypes instances of the class `Term`. Thus, by combining the unification algorithm from section 4.2 with the polytypic instance declarations from this section, we get a unification algorithm that works for all regular term types.

Acknowledgements

Richard Bird, Doaitse Swierstra, Patrik Berglund and Oege de Moor commented on previous versions of the paper.

Chapter 5

Implementing PolyP

This chapter discusses the implementation of a compiler for the language extension PolyP. As a short summary of what is described at length in the other chapters of this thesis we can say that PolyP is a language extension that allows the definition of polytypic functions by induction on the structure of user defined datatypes. A polytypic Haskell program is compiled (translated) to a Haskell program by expanding out the instances of the polytypic functions that are used. The polytypic definitions are type checked to ensure that the generated instances are type-correct. We will use the name PolyP for both the compiler and the language extension.

To illustrate the usefulness of polytypic programming it would have been instructive to have an implementation of PolyP written in PolyP. Unfortunately the implementation is still experimental and does not handle enough of the Haskell language to make it possible to run it on itself.[1] Therefore the implementation of PolyP is a normal Haskell 1.3 program and all instances of polytypic functions used in it are hand written. We hope that the definitions and typings of these polytypic instances, in addition to showing how PolyP is implemented, can give some idea of the possible uses of polytypism.

In this chapter we will describe the algorithms used and show some samples of the code. A more detailed description is included in the source distribution of PolyP available from:

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

In the distribution most modules are written as literate scripts with comments as LaTeX text so the whole implementation can be fed into LaTeX and printed.

Much of the non-polytypic code is borrowed from the Elmo project (an experimental implementation of Gofer in Gofer) at Utrecht University. We are indebted to J. Fokker for sending us the source code for the Elmo project and

[1] The day PolyP can be bootstrapped it will certainly handle most of Haskell — the code already uses so much of Haskell that it required a new minor release of `hbc` [1] to get going!

A. IJzendoorn for helping us understand and modify it in the beginning of the development of PolyP.

5.1 Program structure

The implementation of PolyP is a Haskell 1.3 program divided into a number of modules. The information flow inside PolyP is as follows: (pointers to more information in the following sections are parenthesised)

- The parser takes an input file to a list of equations expressed in the abstract syntax (section 5.2).
- Dependency analysis splits these equations into datatype declarations and mutually recursive groups of function definitions.
- For each regular datatype the corresponding functor is calculated. (section 5.5.1)
- The equation groups are labelled with type information and evidence values using a type inference algorithm (section 5.7) similar to Jones' translation and dictionary insertion algorithm [48, symbol $\leadsto$].
- The labelled 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 (section 5.8).
- The final equation list is pretty printed using Hughes' combinators [36].

The following sections will describe the different parts and show some (often simplified) parts of the code.

5.2 Abstract syntax

The abstract syntax of PolyP represents a subset of Haskell extended with the possibility to define polytypic functions. As expressions can contain `let`-expressions, and `let`-expressions contains equations which in turn contain expressions we use two mutually recursive types, `Expr` and `Eqn`, to represent expressions and equations. Expressions consists of variables, constructors, applications, lambda abstractions, literals, wild-cards, case expressions, let expressions and explicitly typed expressions:

```
data Expr
    = Var VarID
    | Con ConID
    | Expr :@: Expr
    | Lambda Pat Expr
    | Literal Literal
    | WildCard
    | Case Expr [(Pat, Expr)]
    | Letrec [[Eqn]] Expr
    | Typed Expr Type

type Pat = Expr
data Literal = BoolLit Bool | IntLit Int | ...  deriving Eq
```

The first argument of the `Letrec` constructor is a list of mutually recursive groups of equations.

The datatype for equations contain normal variable bindings but also polytypic bindings:

```
data Eqn =
     ...
    | Polytypic VarID Type     Func  [(Func, Expr)]
--   polytypic   p :: t  = case f of  {fi -> ei}
```

Here `VarID`, `ConID` and `Func` are the types for variable identifiers, constructor identifiers and functors respectively.

Simple types, kinds and functors are all represented by the type `Type` which contains type variables, type constructors and type application:

```
data Type = TVar VarID  | TCon ConID  | Type :@@: Type
type Kind = Type
type Func = Type
```

Function types are represented as an application of the constructor `->` to two arguments. As an example, the type `a -> b` is represented by `TCon "->" :@@: TVar "a" :@@:  TVar "b"`.

Kind is a *sort* containing types in the terminology of Barendregt [5]. Kinds are associated with types in much the same way as types are associated to expressions. For example: `Int :: *` and `List :: * -> *` where `*` is the kind of types.

5.2.1 Parsing

The parser is a simple combinator parser in the style of [24] without a separate lexical analysis phase and it is probably very inefficient. We have deliberately spent little time on this part as we probably will replace it by some complete

Haskell parser in the future (perhaps the parser from `nhc` [76]). The parser does some translations of the input; `if` expressions are translated to `case` expressions and multiple parameter λ-abstractions (`\x y z-> e`) are translated to nested one parameter abstractions (`\x-> \y-> \z-> e`) to fit into the abstract syntax.

5.3 Folding

As we need to traverse and transform expressions, equations and types a number of times in the program, we define catamorphisms over these types. As mentioned in the beginning of this chapter these definitions are hand-written but as soon as PolyP is self-applicable they will be automatically generated. The function `cataExpr` folds an expression to a value by replacing the constructors by the supplied functions. As expressions and equations are mutually recursive `cataExpr` takes as input not only the functions that are to replace the constructors of the type `Expr`, but also the replacements for the constructors in the type `Eqn`. To abbreviate the types of the functions that are used as arguments to the catamorphisms we define two type synonyms, `GExpr` and `GEqn`, in the style of Sheard [79]. These type synonyms resemble[2] the functors of the datatypes `Expr` and `Eqn`.

```
cataExpr :: ( GExpr x q , GEqn x q ) -> Expr -> x
cataEqn  :: ( GExpr x q , GEqn x q ) -> Eqn  -> q

type GExpr x q =
  ( VarID         -> x  -- variable
  , ConID         -> x  -- constructor
  , x -> x        -> x  -- application
  , x -> x        -> x  -- lambda abstraction
  , Literal       -> x  -- literal
  ,                  x  -- wild-card
  , x -> [(x, x)] -> x  -- case
  , [[q]] -> x    -> x  -- letrec
  , x -> Type     -> x  -- explicitly typed expression
  )

type GEqn x q =
  ( ...
  , VarID -> Type -> Func -> [(Func,x)] -> q -- polytypic def.
  )
```

Function `cataType` is the catamorphism for types, and this definition is so small that we can include it as an illustration of a typical catamorphism. From the

[2]They are in fact isomorphic to variants of the functors, e.g. $((FunctorOf\ Eqn)\ x\ q \to q) \cong$ (`GEqn` $x\ q$) where the isomorphism is of the kind $((a+b) \to c) \cong (a \to c) \times (b \to c)$.

definition of the local help function `cT` in function `cataType` below it is evident that the constructors of the type are replaced by the corresponding function arguments in a recursive traversal of the expression. (We repeat the definition of `Type` here for clarity.)

```
data Type  = TVar VarID      | TCon ConID      | Type:@@:Type
type FType a b = (VarID -> b,      ConID -> b,   a  ->  a  -> b)

cataType :: FType a a -> Type -> a

cataType (var,con,app) = cT
  where cT (TVar v)   = var v
        cT (TCon c)   = con c
        cT (f :@@: x) = (cT f) `app` (cT x)

mcataType :: Monad m => FType a (m a) -> Type -> m a
mcataType (var,con,app) = cataType (var,con,mapp)
  where mapp mf mx = mf >>= \f -> mx >>= \x -> app f x
```

Using the catamorphism we also define a *monadic catamorphism* [28, 67] that threads a monad through all applications. This monadic catamorphism is used in the translation of types from the abstract syntax to the graph representation used in the type inference phase.

5.4 Representing types

Types can be seen as expressions in a rudimentary functional language built up from variables, constants and applications. These type expressions can be viewed as expression trees with applications as branching points and constants and variables at the leaves. By generalising the trees to directed acyclic graphs (DAGs) we can express sharing of subexpressions. During type inference we represent types as DAGs using mutable variables encapsulated in the `ST s`-monad [20]. We chose this design to learn about and to experiment with 'stateful programming' using monads in a functional language and with the hope that the resulting implementation would gain efficiency from the ability to do update in place. When it comes to learning and experimenting this was a definite success, but it does not seem to improve the efficiency that much.

The `Hp` in the types below is an abbreviation for heap and it indicates that the mutable variables can be thought of as pointers into memory locations on the heap. We will call types containing mutable variables *heap types*.

During type inference types are represented by pointers to elements of the datatype `HpNode s`. The type `HpNode s` has the constructors `HpVar` for type variables, `HpCon` for type constructors and `HpApp` for type applications. In the representation of variables we use a 'trick' to simplify the substitution of a type for a type variable. The case `HpVar` is used to represent a type variable (if it points to itself) or an *indirection node* (if it points somewhere else). An indirection node is simply a node containing a pointer to the rest of the type. In this way, substitution can be performed simply by changing the pointer.

```
data HpNode s = HpVar (NodePtr s)
              | HpCon ConID
              | HpApp (NodePtr s) (NodePtr s)

type NodePtr s = MutVar s (HpNode s)
type HpType s  = NodePtr s
type HpKind s  = HpType s
```

Expressions in this type are built using the pseudo constructors `mkVar`, `mkCon` and `mkApp`. As in the abstract syntax for types, there is no constructor for function types as they can be built by applying the constructor `->` to two arguments, but we provide a pseudo constructor `mkFun` also for that one:

```
mkVar   ::                            ST s (NodePtr s)
mkCon   ::              ConID      -> ST s (NodePtr s)
mkApp   :: NodePtr s -> NodePtr s -> ST s (NodePtr s)
mkFun   :: NodePtr s -> NodePtr s -> ST s (NodePtr s)


mkVar =  newVar undefined     >>= \v -> -- create variable
         writeVar v (HpVar v) >>        -- tie a knot
         return v                       -- return the pointer


mkCon conID = newVar (HpCon conID)
mkApp f g   = newVar (HpApp f g)


mkFun a b   = mkCon "->"   >>= \to ->
              mkApp to  a  >>= \ato->
              mkApp ato b
```

The primitives for mutable variables in the `ST s` monad are:

```
newVar    ::  a -> ST s (MutVar s a)
writeVar  ::  MutVar s a -> a -> ST s ()
readVar   ::  MutVar s a -> ST s a
eqVar     ::  MutVar s a -> MutVar s a -> Bool
```

5.4.1 Catamorphisms for heap types

The catamorphism for `HpType` gives its result in the `ST s`-monad as it uses data on the heap. The catamorphism takes three functions to replace the three constructors with. A combination of `NodePtr s` and `s` (the state) can be seen as the real type of an element of a heap type as its value depends on a location on the heap. We use the type `NodePtr s -> ST s a` for the function that replaces `HpVar` to allow this function to access the value the node pointer points to.

```
type GHpType s a = ( NodePtr s -> ST s a  -- HpVar
                   , ConID     ->      a  -- HpCon
                   , a -> a    ->      a  -- HpApp
                   )

cataHpType :: GHpType s a -> HpType s -> ST s a
```

This catamorphism is used when translating the DAG representation to the abstract syntax for types. In the opposite direction, translating from the abstract syntax for types to the DAG representation[3], an environment is used to make sure that the variables are shared. This translation is expressed using the monadic catamorphism for `Type`; `mcataType`. A future extension of the code to share constructors or even common subexpressions can be obtained with minor changes of the code.

5.5 Functors and type evaluation

5.5.1 From datatypes to functors

For every regular datatype `D` we need the functor `fD` that represents its recursive structure: `Mu fD` = `D` or equivalently `fD` = `FunctorOf D`. The function `functorOf :: Eqn -> Func` transforms the abstract syntax representing a datatype definition to a functor. Note that `functorOf` (with a lower case `f`) is an implementation of the `FunctorOf` from the type system. Function `functorOf` recursively replaces alternatives with $+$, tuples by $\times$ and *Empty*, parameters by *Par*, recursive occurrences with *Rec*, constant types with *Const* and type applications with @. The current version of `functorOf` requires exactly one type parameter and does not handle nested datatype applications on the right hand side.

5.5.2 Type evaluation

In the typing rule for the `polytypic` construct (see figure 2.7) we substitute functor expressions for functor variables. These substituted functor expressions

[3]As the abstract syntax for types is used in parsing the input and in pretty printing the output of PolyP while the type inference algorithm in the middle uses DAGs we need to translate the types in both directions.

$$\frac{}{v \sim v} \qquad \frac{}{c \sim c} \qquad \frac{v \notin t, \quad v := t}{v \sim t} \qquad \frac{f \sim g \quad a \sim b}{f\ a \sim g\ b}$$

Figure 5.1: The unification rules

are treated as type synonyms of kind `* -> * -> *`. This means that if we see type expressions as expression in a functional language, the substituted functor expressions are functions taking two arguments. Calculating the types by expanding out the type synonyms is equivalent to evaluating the type expression using β-reduction. As we have a graph based representation of types the type evaluation uses graph reduction [74, chapter 12]. The type graph reduction function `typeEval` takes a (pointer to the) type expression to be reduced and overwrites it with the result as a 'side effect' in the `ST s` monad:

```
typeEval :: HpType s -> ST s ()
```

5.6 Unification

The implemented unification algorithm is not the algorithm described in section 2.3.3 but hopefully a fair approximation of it. (The implementation seems to lag behind the theory by at least half a year. :-) In this section we first describe the unification algorithm for the simple type language without any polytypic additions, and then the additional machinery that is used to approximate[4]unification when functors are added.

5.6.1 Simple DAG-unification

Unification algorithms normally take two terms containing variables as input and return a substitution that, when applied, makes the two terms equal. In all cases[5] where the unification algorithm is used in type inference we are interested only in the types *after* application. Thus we can combine the generation of the substitution and its application to the terms. Furthermore, as we represent types by updatable DAGs, we can use destructive update to overwrite type variables with the type terms they should equal instead of introducing new bindings in a substitution.

The function `unify` unifies two types, the rules are shown in figure 5.1. Variables are overwritten with with the corresponding term, type constructors are checked for equality and application nodes unify their children. An occurs-check prevents the construction of cyclic types.

[4]We use the word *approximate* as we have no proof that this unification algorithm does the right thing in all cases.

[5]At least in this implementation.

```
unify :: HpType s -> HpType s -> STErr s ()
```

The `STErr s` is a combination of the `ST s`-monad with an error monad so that `unify` can report an error if the terms are impossible to unify. In the case when a type variable `v` is unified with some term `t` the variable `v` is overwritten with an indirection node that points to the term `t`. This type variable might occur in other parts of the type being unified and all these occurrences must also be changed to `t`. But as we know that the translation from the abstract syntax for types produces a DAG with the property that all variables are shared, this single overwrite automatically updates all occurrences of the type variable `v`. See van Dijk's paper [20] for the details.

5.6.2 Type environment

The type environment is the Γ in $P \mid \Gamma \vdash^w x : \tau$ (see section 2.3) and it contains information about PolyP's predefined types and functions. The most important ones are

```
Mu        :: (* -> * -> *) -> (* -> *)
FunctorOf :: (* -> * -> *) <- (* -> *)
```

```
inn :: Poly (FunctorOf d) => (FunctorOf d) a (d a) -> d a
out :: Poly (FunctorOf d) => (FunctorOf d) a (d a) <- d a
```

where `*` is the kind of types and `a <- b` is a convenient notation for `b -> a`. Note that the context part of the the types for `inn` and `out` uses `Poly (FunctorOf d)` to denote `Regular d`. This is minor difference in notation from the theory and will be removed in a later version. Currently a minimal prelude of common Haskell functions is included but this will be replaced by a complete prelude shortly.

The environment associates names with type schemes (types in which some type variables are $\forall$-quantified). We call the $\forall$-quantified type variables *generic* (type) variables. We implement the environment by splitting it into two parts. The first part is an association of names with types and the second part a list of all non-generic variables `ngs :: [HpType s]`.

If a non-generic type variable is unified with a type `t`, all of `t`'s variables must be made non-generic too [74, section 9.5]. This is automatically handled by the representation of `ngs`: it is a list of pointers into the types that are unified, with the interpretation that all variables reachable from that list are non-generic. This means that this list is automatically kept up to date without being explicitly used in `unify`. (A very useful 'side effect'!)

5.6.3 Type ordering

In the type inference algorithm for the `polytypic` construct and when expressions or equations are explicitly typed, we have to check that the specified type is an instance of the inferred type. This check is performed by the function `checkInstance` which performs a 'one-way' unification that only allows non-generic variables to be instantiated to non-generic types. The implementation is similar to that of unification and has the cases shown in figure 5.2 (where we informally denote `checkInstance` by $\leq$). In all other cases the types are either incomparable (type ordering is a *partial* order) or ordered in the opposite direction.

$$\frac{}{t \leq \forall v.v} \qquad \frac{t \text{ contains no generic variables}}{t \leq v}$$

$$\frac{}{c \leq c} \qquad \frac{f_1 \leq f_2 \quad e_1 \leq e_2}{f_1\ e_1\ \leq\ f_2\ e_2}$$

Figure 5.2: Rules for type instantiation ($\leq$).

5.6.4 Unification with functors

As PolyP extends the type language to include functors the unification algorithm must be extended correspondingly. We use a function `functorOf` that takes the name of a datatype to the functor that represents its structure. It is (a lookup in a table generated by) the function `functorOf` defined in section 5.5.1 above. The new cases are inserted where the simple unification algorithm would fail — when a constructor is unified with an application. If this application is of the form `Mu f` or `FunctorOf D` we proceed as follows:

- (`Mu f`, `D`): use the equality `D = Mu (FunctorOf D)` to the the pair (`Mu f`, `Mu (FunctorOf D)`), strip off the `Mu`'s and go on to unify (`f`,`FunctorOf D`)
- (`FunctorOf D`, `f`) where `f` is generated by `Empty`, `Const`, `Par`, `Rec`, `+`, `*`, `@` and type (functor) variables: Succeed if (`functorOf D`, `f`)[6] are unifiable and change `f` to be `FunctorOf D`, fail otherwise.

These rules are implemented and work for all the examples given in this thesis, but there are probably examples where this algorithm would give the wrong result. We are working on replacing this with an implementation of the correct algorithm, given in section 2.3.3.

[6]Note that we use `functorOf` (with the lower case `f`) from section 5.5.1 here.

5.7 Type inference

This section describes the implementation of the type inference algorithm from section 2.3.

5.7.1 Kind inference

The kind inference algorithm is used to check that all type expressions in data-type declarations are kind-correct. It is a simpler variant of the type inference algorithm and takes a kind environment and a type as input and produces a kind or an error message as output. We represent kinds with the same type as types (using mutable variables) so the result of the kind inference algorithm must be in the `ST s` monad. As the inference algorithm might fail, we use a combination of the state monad an the error monad: `STErr s`. Here is the complete algorithm:

```
(|*) :: KindBasis s -> Type -> STErr s (HpKind s)

basis |* (TVar name) = name `lookupKind` basis
basis |* (TCon name) = name `lookupKind` basis
basis |* (f :@@: x)
  = basis |* x        >>= \kX   ->
    basis |* f        >>= \kF   ->
    mkVar'            >>= \kApp ->
    mkFun' kX kApp    >>= \kF'  ->
    unify kF kF'      >>
    return kApp
```

The primed variants of `mkVar` and `mkFun` are lifted from the `ST s` to the `STErr s` monad. Function `lookupKind` looks up names of constructors and variables in the kind environment (of type `KindBasis s`). If a name is not found this is reported by producing an error message in the `STErr s` monad.

5.7.2 Type labelling

The type inference algorithm is an extension of Damas-Milner's algorithm W [17] similar to that in Mark Jones' thesis work on qualified types [48]. The main type labelling function is `(|->)` that takes a type basis (containing types of previously defined functions) and an expression and produces a type labelled expression and the top level type. To infer the types in a group of mutually recursive definitions we need to:

- Assume new type variables for the variable bindings.
- Extend the environment with the explicitly given types for the polytypic bindings.

- Label the value bindings with types:
  - Make the type variables temporarily non-generic. (All uses of a function within its definition group must have the same type.)
  - Label all equations.
- Label the polytypic definitions.
- Return a list with the types of the value bindings and the labelled equations.

The fact that the variable bindings are temporarily given non-generic types means that we don't allow polymorphic recursion. The explicitly given types in the polytypic declarations are treated as containing only generic variables (just like any other explicit type).

To check a polytypic definition we first infer the types of the case alternatives one by one giving t_i. We also calculate the types τ_i that the alternatives *should* have by substituting the different functor alternatives for the functor in the explicitly given type and evaluating the resulting type using `typeEval`. Finally we check that the inferred types are more general than the calculated types (i.e. that $t_i \geq \tau_i$) using `checkInstance` defined in section 5.6.3.

5.8 Code generation

This section deals with creating specific instances of polytypic functions. If we see a polytypic function as taking (a representation of) a type as its first argument we can say that the code generation phase partially evaluates this function with respect to the type argument. This type argument is inserted by the type inference algorithm.

The overall structure is as follows: replace polytypic identifiers with (names of) instances and generate a list of requests for instance declarations. Every request in the list is then handled in much the same way: generate an instance declaration and (possibly) new requests. If the request is for a `polytypic` definition the corresponding functor case is looked up and inserted. If the request is for `inn` or `out` its corresponding instance is generated. And if the request is for a normal function definition, this definition is traversed recursively. The resulting program will have the same structure as the old but with lots of instance declarations added. See appendix A for an example of the result of the translation for a simple polytypic program.

5.8.1 Generating requests

We generate all requests by traversing the expression tree representing the whole program starting at the `main` definition and emitting requests for the traversal of the free variables (names of functions) that we encounter. Each request contains the name of the function to be instantiated and the type it should be instantiated to. The traversal function is a monadic catamorphism over the abstract syntax for equations and expressions. We use a combination of a reader monad, for keeping track of the types of all free variables, and a writer monad, that collects the list of requests.

5.8.2 Generating `polytypic` instances

If the traversal function issues a request for an instance of type `t'` of a function `p` defined by the `polytypic` construct:

```
polytypic p :: t = case f of { fi -> ei }
```

Then the type `t'` is matched with `t` and the functor part `f'` corresponding to the functor variable `f` in the functor case is extracted. The functor `f'` is then matched with the `fi` patterns in the case branches and the first matching case (`fj -> ej`) is selected. The equation `p_f' = ej` is used as the instance for function `p` after it has been traversed and instantiated recursively. See section 2.2.2 for an example.

5.8.3 Generating $\mathtt{inn}_D$ and $\mathtt{out}_D$

Instances of the polytypic functions $\mathtt{inn}_D$ and $\mathtt{out}_D$ for a datatype $D\ a$ are generated from the following pattern:

$$\mathtt{inn}_D = \mathtt{uncurry}_{k_1}\ \mathtt{C}_1\ \text{`}\mathtt{either}\text{`}\ \cdots\ \text{`}\mathtt{either}\text{`}\ \mathtt{uncurry}_{k_n}\ \mathtt{C}_n \tag{5.1}$$

where $\mathtt{C}_i$ are the n constructors of the datatype $D\ a$ and $k_i = \mathit{arity}(\mathtt{C}_i)$. The functions $\mathtt{uncurry}_k$ are generalisations to arbitrary arity k of the binary (i.e. $k = 2$) `uncurry` in Haskell.

$$\begin{array}{l}
\mathtt{out}_D\ x = \mathtt{case}\ x\ \mathtt{of} \\
\quad \begin{array}{llllll}
\mathtt{C}_1 & \mathtt{a}_{1,1} & \cdots & \mathtt{a}_{1,k_1} & \rightarrow & \mathtt{Left}\ (\mathtt{a}_{1,1}, (\cdots, \mathtt{a}_{1,k_1})\cdots) \\
\mathtt{C}_2 & \mathtt{a}_{2,1} & \cdots & \mathtt{a}_{2,k_2} & \rightarrow & \mathtt{Right}\ (\mathtt{Left}\ (\mathtt{a}_{2,1}, (\cdots, \mathtt{a}_{2,k_2})\cdots)) \\
\vdots & \vdots & & \vdots & \vdots & \vdots \\
\mathtt{C}_n & \mathtt{a}_{n,1} & \cdots & \mathtt{a}_{n,k_n} & \rightarrow & \mathtt{Right}^{n-1}\ (\mathtt{Left}\ (\mathtt{a}_{n,1}, (\cdots, \mathtt{a}_{n,k_n})\cdots))
\end{array}
\end{array}$$

5.9 Conclusions and future work

The implementation of PolyP is still experimental. It suffers from many restrictions in the expressiveness of the language that we hope to cure soon. Yet it has been used to implement a number of polytypic functions: `pmap`, `cata`, `pzip`, unification [38, 40], rewriting [8, 46], parsing [12], genetic programming [84].

There are several improvements on our wish list:

- Support for full Haskell syntax and the full Haskell type system.
- Polytypic functions for multi parameter datatypes.
- Polytypic functions for mutually recursive datatypes.
- Plugging PolyP into some existing Haskell compiler.
- Produce code that uses some kind of dictionaries that would create instances lazily at run time.
- ...

Chapter 6

Related work

In 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.

6.1 The theory of datatypes

The basic idea behind polytypic programming is the idea of modelling 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 [56], Manes and Arbib [60], and Hagino [31]. Böhm and Berarducci [13] have a more algebraic approach to modelling 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. Fokkinga extends the theory of datatypes to include 'Datatype Laws without Signatures' [25] enabling abstract datatypes like stacks to be defined in a category theoretic setting.

6.2 BMF $\cong$ Squiggol

Polytypism has its roots in the branch of constructive algorithmics that was named the Bird-Meertens Formalism (BMF) [10, 62] by Backhouse [2]. 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, 43] 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[1] community, see [3, 9, 27, 59, 63, 64, 65, 67], where the list based calculus is generalised and

[1] The term "Squiggol" comes from the use of more and more "squiggly" symbols to abbreviate common functions.

extended to other datatypes[2] and polytypic versions of many list functions are defined: `cata`, `map`, `zip`, `sum` etc. Together with the functions, also the theorems and the transformation techniques developed in the theory of lists were generalised. This leads us to the subject of polytypic calculation.

6.3 Polytypic calculation

Malcolm [59] and Fokkinga [26, 25, 27] develop categorical techniques for calculating and transforming programs. The most well know polytypic transformation is the Fusion law, first described by Malcolm [58, 59].[3]

$$
\begin{array}{ll}
 & h \cdot \mathit{cata}_D\ f \ = \ \mathit{cata}_D\ g \\
\Leftarrow & \quad (\text{Fusion}) \\
 & h \cdot f \ = \ g \cdot \mathit{fmap}_D\ \mathit{id}\ h
\end{array}
$$

The Fusion law (for applications, see section 3.2.7) gives the conditions under which the composition of a function with a catamorphism can be fused to just one catamorphism. Takano and Meijer [83] use another polytypic law, the acid-rain theorem, to apply deforestation [85] transformations and Hu [35] uses a number of polytypic laws to eliminate multiple traversals of data by combining functions that recurse over the same structure.

Both the the fusion law and the acid-rain theorem are examples of *free theorems* [86].[4] A free theorem can be derived automatically from the polymorphic type of a function. Fegaras and Sheard [23, appendix A.1] (in more detail: [22]) give a function that given a type constructs its free theorem.

More examples of polytypic calculation of programs can be found in de Moor [72] and Meertens [64].

6.4 Relational polytypism

Backhouse *et al.* [3, 4] argues very convincingly that the basis of the theory of polytypism is best described in a relational setting. Bird, de Moor and Hoogendijk [9] use this setting to generalise the theory of segments of lists to all datatypes.

6.5 Regular datatypes and beyond

Polytypic functions are normally defined for regular datatypes. Regular datatypes are initial fix-points of regular functors or, in the relational setting, regular relators [4].

[2]Datatypes that can be defined by means of a regular functor.

[3]Malcolm calls it the promotion theorem following Bird's terminology for lists [10].

[4]For example, the Fusion law is the free theorem of function `cata`.

Jay [42] has developed an alternative theory for polytypic functions (in his terminology: *shapely* functions), in which values are represented by their structure and their contents. He uses an category theoretic formulation of polytypism based on the notion of strong functors [69, 68].

The class of datatypes on which polytypic functions can be defined can be extended (with some effort) to include datatypes with function spaces. Freyd [29] 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, the recursive definition of the catamorphism uses its own (right) inverse. Meijer and Hutton [66] 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 [23] points 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 could lead to trouble.

6.6 Type systems

Type systems for languages which allow the use of polytypic functions have been developed by several people:

- Ruehr [77] 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 [48, 51] is based on qualified types and higher-order polymorphism. The type system is implemented in the functional language Gofer [50] (a Haskell variant). Gofer has no construction for writing polytypic functions by induction on user defined datatypes but can be used to simulate and type check polytypic functions. For an example, see the code in appendix C.

- Jay *et al.* [7, 41] 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 has no recursive `let` and no fixpoint operator so all recursive definitions must be expressed using `fold`. The language can deal with multiple parameter datatypes, but not mutual recursive datatypes and there is no means of introducing new polytypic function defined by induction over datatypes.

- Sheard and Nelson [80] gives a type system for a restricted version of Compile-time Reflexive ML (CRML [34]). 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.

- Our type system (described in chapter 2 and [39]) extends Jones' system [48, 51] with the possibility to introduce and type check polytypic functions defined by induction on the structure of user defined datatypes.

6.7 Implementations

In previous chapters of this thesis we have argued that a polytypic programming system should

- type check the polytypic code,
- allow definitions of new polytypic functions, and
- generate instances of these polytypic functions for regular datatypes.

The categorical language `Charity` [15] and the functional language **P2** [41] do not satisfy the second requirement. A predecessor of PolyP, Hollum [45] does not satisfy the first requirement. Our system PolyP, described here and in chapter 2 satisfies these requirements and we know of only one other such system: that of Sheard [78] 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 [34]) extended with a type system using some dependent types. We did not want to move that far away from the Haskell (type) system. But the compile time reflexive setting could very well be used to *implement* future versions of PolyP.

6.8 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 [13, 21, 38, 39, 41, 71, 81, 78, 79]. Catamorphisms were early generated by Böhm and Berarducci [13] (in the Λ-calculus) and Sheard [81] (in an ML-like language). Sheard also gave programs to automatically generate other kinds of traversal functions like accumulations and equality functions.

The paramorphism[5], a more general recursion operator than the catamorphism, was introduced by Meertens [63] and many other recursion operators are defined in [70, 71]. Catamorphisms can also be generalised to *monadic catamorphism* [28, 67] that thread a monad through all applications.

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.* [9] and Jeuring [44].[6] Many other algorithms have also been expressed polytypically: unification [38, 40], pattern matching [44], data compression [39], parsing [37, 12], rewriting [46, 8], genetic programming [84], etc.

All the polytypic functions above are parametrised on *one* datatype. There is however no theoretical problem with defining multiply parametrised polytypic functions. One example is the doubly parametric function `transpose` (also called `zip`) defined by Ruehr [77] and Hoogendijk and Backhouse [33]. In PolyP it could have the type:

```
transpose :: (Regular d, Regular e) => d (e a) -> e (d a)
```

It is a generalisation of the transpose operation on matrices.

6.9 Adaptive Object-Oriented Programming

In object-oriented programming polytypic programming appears under the names 'design patterns' [30], and 'adaptive object-oriented programming' [57, 73]. Adaptive OOP is a programming style similar to polytypic programming. In adaptive OOP methods (corresponding to our polytypic functions) are attached to groups of classes (types) that usually satisfy certain constraints (such as being regular).

Lieberherr *et al.* [57] describes a system that allows the programmer to write template programs containing a number of methods with associated 'propagation patterns'. The template programs are parametrised on (the structure of) a group of related classes and the system automatically instantiates these templates for different class dependence 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 dependence graph that method should be used on) and a code part (to be executed for all matching paths).

As an example they give a program that sums the salaries in a conglomerate of companies. The template has one method and this method has the pattern `from Conglomerate to Salary` where `Conglomerate` is the root node of the class graph and `Salary` is the name of the container object used for the salary of an employee. The signature of the method says that the method updates an integer variable `totalSalary` (passed in as a reference), and the code part just increases `totalSalary` with the local salary.

[5]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.

[6]The first published use of the term *polytypic function* was by Jeuring in [44].

Appendix A

Translating PolyP-code to Haskell

A.1 A simple PolyP program

Combining the definitions from figure 2.1 (`flatten`) with the definition of `fmap` in figure 2.2 and the code below we get a small polytypic program testing the function `separate`. We assume a prelude containing composition and definitions of the functions `const`, `concat`.

```
main = (separate l,separate r)

l = Cons 1 (Cons 2 Nil)
r = Fork 1 (Cons (Fork 2 Nil) Nil)

data List a = Nil | Cons a (List a)
data Rose a = Fork a (List (Rose a))

separate x = (pmap (const ()) x,flatten x)

pmap f = inn . fmap f  (pmap f) . out
cata h =  h  . fmap id (cata h) . out
```

A.2 The generated code

The code generated by PolyP looks as follows. We have edited the generated code slightly.

```
uncurry0 f p = f
uncurry2 f p = f (fst p) (snd p)
```

```
data List a = Nil | Cons a (List a)
data Rose a = Fork a (List (Rose a))

main = (separate_f4List l, separate_f4Rose r)
l = Cons 1 (Cons 2 Nil)
r = Fork 1 (Cons (Fork 2 Nil) Nil)
separate_f4List x= (pmap_f4List (const ()) x, flatten_f4List x)
separate_f4Rose x= (pmap_f4Rose (const ()) x, flatten_f4Rose x)
pmap_f4List f = inn_f4List . fmap_f4List f (pmap_f4List f)
                           . out_f4List
pmap_f4Rose f = inn_f4Rose . fmap_f4Rose f (pmap_f4Rose f)
                           . out_f4Rose
flatten_f4List = cata_f4List fl_f4List
flatten_f4Rose = cata_f4Rose fl_f4Rose
inn_f4List = either (uncurry0 Nil) (uncurry2 Cons)
fmap_f4List = \p r -> (fmap_e p r) -+- (fmap_Ppr p r)
out_f4List x = case x of
                 Nil -> Left ()
                 (Cons a b) -> Right (a, b)
cata_f4List h = h . fmap_f4List id (cata_f4List h)
                  . out_f4List
fl_f4List = either fl_e fl_Ppr
inn_f4Rose = uncurry2 Fork
fmap_f4Rose = \p r -> (fmap_p p r) -*- (fmap_A4Listr p r)
out_f4Rose x = case x of
                 (Fork a b) -> (a, b)
cata_f4Rose h = h . fmap_f4Rose id (cata_f4Rose h)
                  . out_f4Rose
fl_f4Rose = \(x, y) -> fl_p x ++ fl_A4Listr y
f -+- g = either (Left . f) (Right . g)
fmap_e = \p r -> id
fmap_Ppr = \p r -> fmap_p p r -*- fmap_r p r
fl_e = \x -> ([])
fl_Ppr = \(x, y) -> (fl_p x) ++ (fl_r y)
(f -*- g)  (x, y) = (f x, g y)
fmap_p = \p r -> p
fmap_A4Listr = \p r-> pmap_f4List (fmap_r p r)
fl_p = \x -> x : ([])
fl_A4Listr = concat . flatten_f4List . (pmap_f4List fl_r)
fmap_r = \p r -> r
fl_r = \x -> x
```

Appendix B

A small polytypic function library

This appendix contains a small polytypic function library. Many of the functions are also described in section 3.3. The following polytypic functions are defined:

1. **Base:** `fmap`, `pmap`, `cata`, `ana`, `hylo`
2. **Sum:** `sum`, `size`
3. **Flatten:** `flatten`, `fl`, `fl_par`, `fl_rec`, `fl_all`
4. **Crush:** `crush`, `fcrush`, `sum`, `conc`, `size`, `flatten`
5. **Propagate:** `propagate`, `fprop`
6. **Thread:** `thread`, `fthread`
7. **Zip:** `pzip`, `pzipWith`, `pzip'`, `fzip`
8. **ConstructorName:** `constructorName`, `fconstructorName`
9. **EqOrd:** `peq`, `pord`, `cnumber`, `fcnumber`

B.1 Base

```
----------------------------------------------------------------
-- Basic polytypic functions
--   970508 Patrik Jansson
-- module Base where
----------------------------------------------------------------
polytypic fmap :: (a -> c) -> (b -> d) -> f a b -> f c d
  = \p r -> case f of
             g + h     ->  (fmap p r) -+- (fmap p r)
```

```
             g * h     ->  (fmap p r) -*- (fmap p r)
             Empty     ->  id
             Par       ->  p
             Rec       ->  r
             d @ g     ->  pmap (fmap p r)
             Const t   ->  id


-- pmap :: Regular d => (a -> b) -> d a -> d b
pmap f   = inn . fmap f  (pmap f)   . out

-- cata :: Regular d => (FunctorOf d a b -> b) -> (d a -> b)
-- ana  :: Regular d => (FunctorOf d a b <- b) -> (d a <- b)
-- hylo :: Bifunctor f => (f a b -> d)-> (c -> f a b) -> c -> d
cata i   = i   . fmap id (cata i  ) . out
ana    o = inn . fmap id (ana    o) . o
hylo i o = i   . fmap id (hylo i o) . o

----------------------------------------------------------------
-- Help functions
--(-*-) :: (a -> c) -> (b -> d) -> (a,b) -> (c,d)
--(-+-) :: (a -> c) -> (b -> d) -> Either a b -> Either c d

(f -*- g) (a,b)  =  (f a,g b)
(f -+- g) = either (Left.f) (Right.g)
```

B.2 Sum

```
----------------------------------------------------------------
-- Sum and size
--   970424 Patrik Jansson
-- module Sum where
-- import Base(cata,pmap)
----------------------------------------------------------------
-- sum :: Regular d => d Int -> Int
sum  =  cata fsum

-- Bifunctor f => ...
polytypic fsum :: f Int Int -> Int
  = case f of
      g + h ->  fsum `either` fsum
      g * h ->  \(x,y) -> fsum x + fsum y
```

```
      Empty ->  \x -> 0
      Par ->  id
      Rec ->  id
      d @ g ->  sum . pmap fsum
      Const t ->  \x -> 0

-- size :: Regular d => d a -> Int
size = sum . pmap (\_->1)
```

B.3 Flatten

```
----------------------------------------------------------------
-- Flatten
--   970424 Patrik Jansson
-- module Flatten where
-- import Base(cata,pmap,fmap)
----------------------------------------------------------------
-- flatten :: Regular d => d a -> [a]
flatten  =  cata fflatten

-- Bifunctor f => ...
polytypic fflatten :: f a [a] -> [a]
  = case f of
      g + h     ->  fflatten `either` fflatten
      g * h     ->  \(x,y) -> fflatten x ++ fflatten y
      Empty     ->  nil
      Par       ->  singleton
      Rec       ->  id
      d @ g     ->  concat . flatten . pmap fflatten
      Const t   ->  nil

-- fl_par :: Bifunctor f => f a b -> [a]
-- fl_rec :: Bifunctor f => f a b -> [b]
-- fl_all :: Bifunctor f => f a a -> [a]
fl_par =          fflatten . fmap id  nil
fl_rec = concat . fflatten . fmap nil (singleton.singleton)
fl_all =          fflatten . fmap id  singleton

----------------------------------------------------------------
-- Help functions for lists

singleton x = [x]
```

```
nil x = []
```

B.4 Crush

```
----------------------------------------------------------------
-- Crush
--   970428 Patrik Jansson
-- module Crush where
-- import Base(cata,pmap)
-- main = size "Patrik"
----------------------------------------------------------------
-- crush :: Regular d => (a->a->a) -> a -> d a -> a
crush op e =  cata (fcrush op e)

-- Bifunctor f => ...
polytypic fcrush :: (a->a->a) -> a -> f a a -> a
  = \op e -> case f of
               g + h   ->  fcrush op e `either` fcrush op e
               g * h   ->  \(x,y) -> op (fcrush op e x)
                                        (fcrush op e y)
               Empty   ->  \x -> e
               Par     ->  id
               Rec     ->  id
               d @ g   ->  crush op e . pmap (fcrush op e)
               Const t ->  \x -> e

-- sum  :: Regular d => d Int -> Int
-- conc :: Regular d => d [a] -> [a]
sum  = crush (+)  0
conc = crush (++) []

-- size    :: Regular d => d a -> Int
-- flatten :: Regular d => d a -> [a]
size    = sum  . pmap (\_->1)
flatten = conc . pmap (\x->[x])
```

B.5 Propagate

```
----------------------------------------------------------------
-- Propagate Maybe to the top
--   970424 Patrik Jansson
-- module Propagate where
```

```
-- import Base(cata,inn,pmap)
-- (Can be generalised to thread any monad, see Thread)
----------------------------------------------------------------
-- propagate  :: Regular d => d (Maybe a) -> Maybe (d a)
propagate  =  cata (mapMaybe inn . fprop)

-- Bifunctor f => ...
polytypic fprop :: f (Maybe a) (Maybe b) -> Maybe (f a b)
  = case f of
      g + h     ->  sumprop   . (fprop -+- fprop)
      g * h     ->  prodprop  . (fprop -*- fprop)
      Empty     ->  Just
      Par       ->  id
      Rec       ->  id
      d @ g     ->  propagate . (pmap fprop)
      Const t   ->  Just

--sumprop  :: Either (Maybe a) (Maybe b) -> Maybe (Either a b)
sumprop  =  mapMaybe Left `either` mapMaybe Right

--prodprop :: (Maybe a,Maybe b) -> Maybe (a,b)
prodprop p = case p of
               (Just x,Just y) ->  Just (x,y)
               _               ->  Nothing

----------------------------------------------------------------
--  Maybe functions

mapMaybe f = maybe Nothing (Just.f)
```

B.6 Thread

```
----------------------------------------------------------------
-- Threads a monad through a value
--   970424 Patrik Jansson
-- module Thread where
-- import Base(cata,inn,pmap)
----------------------------------------------------------------

-- (Regular d, Monad m) => d (m a) -> m (d a)
thread  =  cata (mapm inn . fthread)
```

```
-- Bifunctor f => ...
polytypic fthread :: f (m a) (m b) -> m (f a b)
  = case f of
      g + h     ->  sumthread  . (fthread -+- fthread)
      g * h     ->  prodthread . (fthread -*- fthread)
      Empty     ->  return
      Par       ->  id
      Rec       ->  id
      d @ g     ->  thread     . (pmap fthread)
      Const t   ->  return

--sumthread  :: Either (m a) (m b) -> m (Either a b)
sumthread  =  mapm Left `either` mapm Right

--prodthread :: (m a,m b) -> m (a,b)
prodthread (mx,my) = mx >>= \x-> my >>= \y-> return (x,y)

----------------------------------------------------------------
-- Monad operations (that are not in PolyPs prelude)

mapm f mx = mx >>= \x -> return (f x)
```

B.7 Zip

```
---------------------------------------------------------------
-- Zip functions
---  970424 Patrik Jansson
-- module Zip where
-- import Base(inn,out, fmap,(-+-),(-*-))
-- import Propagate(fprop,sumprop,prodprop,propagate)
---------------------------------------------------------------
-- Regular d => (d a,d b) -> Maybe (d (a,b))
pzip  =  ( (resultM.inn) @@ (fprop.fmap resultM pzip) @@ fzip )
      .  (out -*- out)

-- Regular d => (d a, d b) -> Maybe (d (a, b))
pzip2 = pzipWith (mapMaybe inn.fprop) resultM (const Nothing)

-- Regular d => (d a, d b) -> (d (a, b),Bool)
pzip3 p = ( pzipWith  inn         id           (const bottom) p
          , pzipWith (and.fl_all) (const True) (const False)  p
```

```
          )

-- Regular d => (FunctorOf d c e -> e) -> ((a,b)->c) ->
--                ((d a,d b)->e) -> (d a,d b) -> e
pzipWith ins op fail (x,y)  =
  maybe (fail (x,y)) (ins.fmap op (pzipWith ins op fail))
        (fzip (out x,out y))

-- Bifunctor f => ...
polytypic fzip :: (f a b,f c d) -> Maybe (f (a,c) (b,d))
  = case f of
      g + h     ->  (sumprop   . (fzip -+- fzip)) @@ sumzip
      g * h     ->  (prodprop  . (fzip -*- fzip)) @@ prodzip
      Empty     ->  const (resultM ())
      Par       ->  resultM
      Rec       ->  resultM
      d @ g     ->  (propagate . (pmap fzip)    ) @@ pzip
      Const t   ->  constzip


--sumzip ::(Either a b,Either c d)-> Maybe (Either (a,c) (b,d))
sumzip (x,y)  =  case (x,y) of
                   (Left s ,Left t ) -> resultM (Left (s,t))
                   (Right s,Right t) -> resultM (Right (s,t))
                   _                 -> Nothing

--prodzip :: ((a,b),(c,d)) -> Maybe ((a,c),(b,d))
prodzip ((x,y),(s,t))  =  resultM ((x,s),(y,t))

-- should be
--   constzip :: Eq t => (t,t) -> Maybe t
--   constzip (x,y) = if x==y then resultM x else Nothing
-- but this requires an inexpressible type of fzip
constzip (x,y) = resultM x

{-
  Maybe monad functions
-}
--resultM  :: a -> Maybe a
resultM x  =  Just x

--bindM    :: Maybe a -> (a -> Maybe b) -> Maybe b
bindM x f  =  maybe Nothing f x
```

```
--(@@)     :: (a -> Maybe b) -> (c -> Maybe a) -> c -> Maybe b
(g @@ f) a =  f a `bindM` g

bottom = bottom
```

B.8 ConstructorName

```
----------------------------------------------------------------
-- ConstructorName
--   970424 Patrik Jansson
-- module ConstructorName where
-- import Base(inn,out)
----------------------------------------------------------------
constructors  :: [d a]
constructors  =  map inn cs

constructorName = fconstructorName . out

polytypic cs :: [f a b]
  = case f of
      g + h     ->  map Left cs ++ map Right cs
      g         ->  [undef]

undef = undef
```

B.9 EqOrd

```
----------------------------------------------------------------
-- Equality and ordering
--   970424 Patrik Jansson
-- module EqOrd where
-- import Base
-- import Flatten
-- import Zip
----------------------------------------------------------------
data Order = Less | Same | More deriving Show

peq x y = maybe False (all (uncurry (==)) . flatten) (pzip (x,y))

peq' l r = pzipWith (and . fl_all)
```

```
    (uncurry (==))
    (\_ -> False)
    (l,r)

pord (x,y) = maybe (compar (cnumber x,cnumber y))
                   (foldr ordop Same .fl_all.fmap compar pord)
                   (fzip (out x, out y))

ordop x y = case x of
              Same -> y
              _    -> x
compar = compar
----------------------------------------------------------------
cnumber = fcnumber . out

polytypic fcnumber :: f a b -> Int
  = case f of
      g + h -> fcnumber `either` (inc . fcnumber)
      g     -> \_ -> 0

inc n = n+1
----------------------------------------------------------------
all p = and . map p
```

Appendix C

Simulating polytypism in Gofer

This appendix shows some examples of how polytypic functions can be simulated in Gofer using higher order polymorphism and constructor classes. The idea and some of the code comes from Mark Jones [49]. `Mu` is a higher-order polymorphic type constructor: its argument `f` is a type constructor that takes two types and constructs a type.

```
----------------------------------------------------------------
-- Representing datatypes

data Mu f a = In (f a (Mu f a))

inn  :: f a (Mu f a) -> Mu f a
inn  =  In

out  :: Mu f a -> f a (Mu f a)
out (In x)  =  x

data Sum  f g  a b  =  InL (f a b) | InR (g a b)
data Empty     a b  =  Empty
data Prod f g  a b  =  Prod (f a b) (g a b)
data Comp f g  a b  =  Comp (Mu f (g a b))
data Par       a b  =  Par a
data Rec       a b  =  Rec b

----------------------------------------------------------------
-- Sums

infix 6 -+-

(-+-) :: (f a b -> f' c d) -> (g a b -> g' c d) ->
         Sum f g a b -> Sum f' g' c d
```

```
(f -+- g) (InL x) = InL (f x)
(f -+- g) (InR x) = InR (g x)

mapsum :: (f a b -> f c d) -> (g a b -> g c d) ->
          Sum f g a b -> Sum f g c d
(f `mapsum` g) x  =  case x of
                       InL y -> InL (f y)
                       InR z -> InR (g z)

junc :: (f a b -> c) -> (g a b -> c) -> Sum f g a b -> c
junc f g (InL x) = f x
junc f g (InR x) = g x

data S a b = Inl a | Inr b

instance (Eq a,Eq b) => Eq (S a b) where
  Inl x == Inl y = x == y
  Inr x == Inr y = x == y
  _ == _ = False

----------------------------------------------------------------
-- Function fmap

class Fun f where
   fmap :: (a -> c) -> (b -> d) -> f a b -> f c d

instance (Fun f, Fun g) => Fun (Sum f g) where
   fmap f g = fmap f g -+- fmap f g

instance (Fun f, Fun g) => Fun (Prod f g) where
   fmap f g (Prod a b) = Prod (fmap f g a) (fmap f g b)

instance Fun Empty where
   fmap f g Empty = Empty

instance Fun Par where
   fmap f g (Par a) = Par (f a)

instance Fun Rec where
   fmap f g (Rec b) = Rec (g b)

instance (Fun f, Fun g) => Fun (Comp f g) where
   fmap f g (Comp x) = Comp (pmap (fmap f g) x)
```

```
-----------------------------------------------------------------
-- Catamorphism and map

pmap :: Fun f => (a -> b) -> Mu f a -> Mu f b
pmap f = inn . fmap f (pmap f) . out

--pmap f = cata (inn . fmap f id)

cata :: Fun f => (f a b -> b) -> Mu f a -> b
cata phi = phi . fmap id (cata phi) . out

ana :: Fun f => (b -> f a b) -> b -> Mu f a
ana  psi = inn . fmap id (ana  psi) . psi

-----------------------------------------------------------------
-- Function flatten flattens a value of an arbitrary datatype.

flatten :: (Fun f, Fl f) => Mu f a -> [a]
flatten = cata fl

class Fl f where
  fl :: f a [a] -> [a]

instance (Fl f,Fl g) => Fl (Sum f g) where
  fl = fl `junc` fl

instance Fl Empty where
  fl Empty = []

instance (Fl f,Fl g) => Fl (Prod f g) where
  fl (Prod x y) = fl x ++ fl y

instance (Fun f,Fl f,Fl g) => Fl (Comp f g) where
  fl (Comp x) = (concat . flatten . pmap fl) x

instance Fl Par where
  fl (Par x) = [x]

instance Fl Rec where
  fl (Rec x) = x

-----------------------------------------------------------------
```

```
-- Functions flr and flp return the elements in the recursive
--   and parameter positions, respectively.

class Flr f where
  flr :: f a b -> [b]
-- definition omitted

class Flp f where
  flp :: f a b -> [a]
-- definition omitted
```

Bibliography

[1] L. Augustsson. *HBC User's Manual.* Programming Methodology Group, Department of Computer Sciences, Chalmers, S–412 96 Göteborg, Sweden, 1993. Distributed with the HBC compiler.

[2] 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.

[3] 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.

[4] R.C. Backhouse, P.J. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J.C.S.P. van der Woude. Polynomial relators. In *Proceedings second Conference on Algebraic Methodology and Software Technology*, 1991.

[5] H. P. Barendregt. Introduction to Generalised Type Systems. *J. Functional Programming*, 1(2):125–154, April 1991.

[6] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression.* Prentice Hall, 1990.

[7] G. Bellè, C.B. Jay, and E. Moggi. Functorial ML. In *PLILP '96*. Springer-Verlag, 1996. LNCS.

[8] Patrik Berglund. A polytypic rewriting system. Master's thesis, Chalmers University of Technology, 1996. Forthcoming.

[9] Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

[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] Staffan Björk. Polytypic parsing. Master's thesis, University of Göteborg, 1997. Forthcoming.

[13] C. Böhm and A. Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[14] Robert D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, 1988.

[15] R. Cockett and T. Fukushima. About `charity`. Unpublished article, see `http://www.cpsc.ucalgary.ca/projects/charity/home.html`, 1992.

[16] J.F. Contla. Compact coding of syntactically correct source programs. *Software — Practice and Experience*, 15(7):625–636, 1985.

[17] L. Damas and R. Milner. Principal type-schemes for functional programs. In *9th Symposium on Priciples of Programming Languages, POPL '82*, pages 207–212, 1982.

[18] Rod M. Davies and Ian H. Witten. Compressing computer programs. Technical Report 93/7, Department of ComputerScience, University of Waikato, Hamilton, New Zealand, 1993.

[19] N. Dershowitz. A note on simplification orderings. *Information Processing Letters*, 9(5):212–215, 1979.

[20] J. van Dijk. Monadic type inference and experimental type systems. In Thomas Johnsson, editor, *Proceedings of The Workshop on the Implementation of Functional Languages '95*, pages 281–293, 1995.

[21] J.H. Fasel, P. Hudak, S. Peyton Jones, and P. Wadler. Sigplan Notices Special Issue on the Functional Programming Language Haskell. *ACM SIGPLAN notices*, 27(5), 1992.

[22] Leonidas Fegaras. Fusion for free! Technical Report CSE-96-001, Department of Computer Science, Oregon Graduate Institute, 1996. Available by `ftp` from `cse.ogi.ed` in `/pub/tech-reports/1996/96-001.ps.gz`.

[23] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions. In *Proceedings Principles of Programming Languages, POPL '96*, 1996.

[24] J. Fokker. Functional parsers. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[25] M. M. Fokkinga. Datatype laws without signatures. *Mathematical Structures in Computer Science*, 6:1–32, 1996.

[26] M.M. Fokkinga. Calculate categorically! In J. van Leeuwen, editor, *Proceedings SION Computing Science in the Netherlands*, pages 211–230, 1991.

[27] M.M. Fokkinga. *Law and order in algorithmics*. PhD thesis, Twente University, 1992.

[28] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, June 1994.

[29] P. Freyd. Recursive types reduced to inductive types. In *Proceedings Logic in Computer Science, LICS '90*, pages 498–507, 1990.

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[31] T. Hagino. *Category Theoretic Approach to Data Types*. PhD thesis, University of Edinburgh, 1987.

[32] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Symposium on Priciples of Programming Languages, POPL '95*, pages 130–141, 1995.

[33] Paul Hoogendijk and Roland Backhouse. When do datatypes commute? Unpublished manuscript, Eindhoven University of Technology, 1997.

[34] James Hook and Tim Sheard. A semantics of compile-time reflection. Oregon Graduate Institute of Science and Technology, Beaverton, OR, USA, 1993.

[35] 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.

[36] J. Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[37] Marieke Huisman. The calculation of a polytypic parser. Master's thesis, Utrecht University, 1996. INF/SRC-96-19.

[38] P. Jansson. Polytypism and polytypic unification. Master's thesis, Computing Science, Chalmers University of Technology, 1995. Available from `http://www.cs.chalmers.se/~patrikj/papers/masters/masters.html`.

[39] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[40] P. Jansson and J. Jeuring. Polytypic unification. Submitted for publication. Available from `http://www.cs.chalmers.se/~patrikj/poly/unify2.ps`, 1997.

[41] C. Barry Jay. Polynomial polymorphism. In *Proceedings of the Eighteenth Australasian Computer Science Conference*, pages 237–243, 1995.

[42] C. Barry Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[43] J. Jeuring. Algorithms from theorems. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, pages 247–266. North-Holland, 1990.

[44] J. Jeuring. Polytypic pattern matching. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 238–248, 1995.

[45] J. Jeuring, G. Hutton, and O. de Moor. Hollum – a polytypic programming extension to gofer. Code obtainable by `ftp` from `ftp.cs.chalmers.se` in the directory `pub/users/johanj/`, file `hollum.tar.Z`, 1994.

[46] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming, Second International School*, pages 68–114. Springer-Verlag, 1996. LNCS 1129.

[47] 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.

[48] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[49] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Lecture Notes in Computer Science 925, pages 97–136. Springer-Verlag, 1995.

[50] Mark P. Jones. Gofer. Available via ftp on `ftp.cs.nott.ac.uk`, 1995.

[51] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, pages 1–35, 1995.

[52] J. Katajainen, M. Penttonen, and J. Teuhola. Syntax-directed compression of program files. *Software–Practice and Experience*, 16(3):269–276, 1986.

[53] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science*, pages 1–116. Oxford University Press, 1992.

[54] K. Knight. Unification: A multidisciplinary survey. *Computing Surveys*, 21(1):93–124, 1989.

[55] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[56] D.J. Lehmann and M.B. Smyth. Algebraic specification of data types: A synthetic approach. *Math. Systems Theory*, 14:97–139, 1981.

[57] 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.

[58] G. Malcolm. Homomorphisms and promotability. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, pages 335–347. Springer-Verlag, 1989. LNCS 375.

[59] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[60] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics.* Text and Monographs in Computer Science. Springer Verlag, 1986.

[61] A. Martelli, C. Moiso, and C.F. Rossi. An algorithm for unification in equational theories. In *Proc. Symposium on Logic Programming*, pages 180–186, 1986.

[62] L. Meertens. Algorithmics — towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North–Holland, 1986.

[63] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

[64] L. Meertens. Calculate polytypically! In H. Kuchen and S.D. Swierstra, editors, *Proceedings of the Eigth International Symposium PLILP '96 Programming Languages: Implementations, Logics and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, 1996.

[65] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA '91*, pages 124–144, 1991.

[66] 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.

[67] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Lecture Notes in Computer Science 925, pages 228–266. Springer-Verlag, 1995.

[68] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89)*, pages 14–23, 1989.

[69] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[70] O. de Moor. *Categories, relations and dynamic programming*. PhD thesis, Oxford University, 1992. Technical Monograph PRG-98.

[71] O. de Moor. Categories, relations and dynamic programming. *Mathematical Structures in Computer Science*, 4:33–69, 1994.

[72] O. de Moor. A generic program for sequential decision processes. In *Programming Languages: Implementations, Logics, and Programs, PLILP '95*, volume 982 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 1995.

[73] J. Palsberg, C. Xiao, and K. Lieberherr. Efficient implementation of adaptive software. *TOPLAS*, 1995.

[74] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[75] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[76] Niklas Röjemo. Highlights from nhc – a space-efficient Haskell compiler. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*. ACM Press, 1995.

[77] Fritz Ruehr. *Analytical and Structural Polymorphism Expressed Using Patterns Over Types*. PhD thesis, University of Michigan, 1992.

[78] T. Sheard. Type parametric programming. Unpublished manuscript, Oregon Graduate Institute of Science and Technology, Portland, OR, USA, 1993.

[79] 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.

[80] 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.

[81] Tim Sheard. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, 1991.

[82] R.G. Stone. On the choice of grammar and parser for the compact analytical encoding of programs. *The Computer Journal*, 29(4):307–314, 1986.

[83] 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.

[84] Måns Vestin. Genetic algorithms in haskell with polytypic programming. Master's thesis, Göteborg University, Gothenburg, Sweden, 1997. Available from `http://www.cs.chalmers.se/~johanj/polytypism/genetic.ps`.

[85] P. Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings European Sytmposium on Programming, ESOP88*, pages 344–358. Springer-Verlag, 1988. LNCS 300.

[86] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, FPCA '89*, pages 347–359. ACM Press, 1989.

[87] T.A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[88] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.