Thesis for the Degree of Licentiate of Engineering

Implementing Functional Generic Programming

Ulf Norell

CHALMERS | GÖTEBORG UNIVERSITY

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

Göteborg, October 2004

Implementing Functional Generic Programming
Ulf Norell

Technical Report no. 39L
School of Computer Science and Engineering

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

Printed at Chalmers, Göteborg, 2004

Abstract

Functional generic programming extends functional programming with the ability to parameterize functions on the structure of a datatype. This allows a programmer to implement certain algorithms once and for all, instead of re-implementing them for each datatype they apply to. Examples of such algorithms include simple traversals and pretty printing as well as more complex XML processing tools.

The topic of this dissertation is the implementation of functional generic programming. More precisely we address two particular questions: how can we reduce the amount of work required to implement generic programming languages, and how can we embed generic programming in an existing functional language.

To answer the first question we show how meta-programming can be used to quickly prototype generic programming languages. In particular we describe prototype implementations of two generic programming languages: PolyP [15] and Generic Haskell [4]. The prototypes are extremely light-weight while still retaining most of the functionality of the original languages. One thing that is missing, though, is a way of adding type systems to the prototypes.

In answer to the second question we show how generic programming can be embedded in Haskell by exploiting the class system. We describe a new version of PolyP (version 2) together with a compiler that compiles PolyP 2 code into Haskell. By compiling the polytypic library, PolyLib [16], with our compiler we get a library of generic functions for Haskell.

Acknowledgments

First of all I would like to thank my supervisor Patrik Jansson without whom I would never had started working on generic programming in the first place. He has been a constant source of good ideas and helpful suggestions guiding me through the writing of this dissertation and the work that lies behind it.

Secondly I would like to thank my fellow PhD students for creating a very nice working environment and for our many interesting and diverse coffee break discussions.

Finally I would like to thank my fiancée for being the wonderful person she is and for not allowing me to get too wrapped up in my work.

Contents

Chapter 1

Introduction

1.1 Generic Programming

Depending on who you ask, generic programming can mean very different things. For a Java programmer, generic programming means parametric polymorphism [3], for a C++ programmer, generic programming is done with *concepts*, a disciplined form of overloading that places requirements on the running time of the operations. The most widely used library for generic programming in C++ is the Standard Template Library (STL) [33]. For a Haskell programmer, and thus in this dissertation, generic programming means parameterizing over the structure of types.

Common for all these different views is that generic programming is done by abstracting over types. The Java and C++ versions of generic programming use two forms of type abstraction: parametric polymorphism, where type parameters are treated as black boxes, and overloading, where there is a separate definition for each type an overloaded function can be applied at. A generic function in these settings is a parametrically polymorphic function that might require some overloaded functions to be defined for its type parameters, and in the case of C++, it also specifies the running time of these functions.

There is a trade-off between the number of types a function applies to and the amount of information it has about its type parameters. A parametrically polymorphic function can be applied at any type, but it cannot do anything interesting with its type parameters since they cannot be inspected. On the other hand, an overloaded function knows precisely at which type it is applied and can do completely different things for different types, but it can only be applied at the types for which it has been defined. Combining the two, unfortunately, does not give us the best of the two worlds. A generic function in the Java and C++ sense can only be applied at types for which the overloaded functions are defined and the only information they have about their type parameters is that the overloaded functions indeed are defined for them.

Instead functional generic programming introduces a different form of type abstraction, namely abstraction over type structures. In this setting a generic function gets access to the structure of its type parameters and can thus do a lot more with them than a parametrically polymorphic function. It can also be applied at a large—and most importantly, open—set of types, since it does not need a separate definition for each type. This is made possible

by the uniformity of algebraic datatypes—every datatype is a sum of products, a number of constructors with zero or more arguments.

The most obvious application for generic programming is the implementation of algorithms that applies to many different datatypes, such as simple traversal and data collection functions. Another case is when the datatype an algorithm works on changes during development, for example, the abstract syntax is likely to evolve during the implementation of a domain specific language. But even if a function is only ever applied to a single never-changing type, there can be benefits from implementing it generically. In many cases the generic implementation is shorter, since it abstracts away from unnecessary details, and because of this there are fewer places where bugs can occur.

Different systems for functional generic programming provide varying levels of access to the datatype structure. In systems like PolyP [15], Generic Haskell [4] and Generic Clean [1] generic functions are defined by pattern matching on the type structure directly, whereas systems like Strafunski [23] and GHC's Data.Generics library [21, 22] provide generic combinators and facilities for overriding their behavior at particular types. In this dissertation we focus on the former systems and describe how they can be implemented.

1.2 Haskell

The topic of this dissertation is the implementation of functional generic programming, but before we can implement functional *generic* programming we need ordinary functional programming. To this end we use the functional language Haskell [30]. We assume a basic knowledge of Haskell and the rest of this section gives an introduction only to the more advanced language concepts that are used in the dissertation.

1.2.1 Laziness

Haskell is a non-strict or lazy language. This means that things are not evaluated until they are actually needed. This makes it possible to create infinite structures like the infinite list of ones defined by

```
ones :: [Int]
ones = 1 : ones
```

We do not make use of laziness in any essential ways in the dissertation but it can be helpful to keep in mind that Haskell is a lazy language when reading some of the code.

1.2.2 Kinds

Kinds are the types of types and a kind can be either $\star$ (pronounced star) or $\kappa \to \nu$ (kappa to nu), for two kinds κ and ν. Haskell expressions always have types of kind $\star$, for instance

```
Int  :: ⋆
Bool :: ⋆
```

$$\begin{array}{ll} [\mathsf{Int}] & :: \star \\ \mathsf{Either\ Bool\ Int} & :: \star \end{array}$$

Type constructors have higher kinds:

$$\begin{array}{l} [] \quad :: \star \to \star \\ \mathsf{Either} :: \star \to \star \to \star \\ \mathsf{Fix} \quad :: (\star \to \star) \to \star \\ \mathbf{data}\ \mathsf{Fix}\, f \ = \ \mathsf{In}\ (f\ (\mathsf{Fix}\, f)) \end{array}$$

When we get to generic programming kinds will play an important rôle. In some cases we want to limit generic programming to datatypes of a single kind, and in other cases we want functions that are generic over datatypes of any kind.

1.2.3 Newtypes

In Haskell you can use the keyword **newtype** in place of **data** to introduce datatypes with a single constructor of only one argument. The advantage of using **newtype** is that an element of a **newtype** has the same representation as the underlying type, i.e. the constructor does not have a physical representation in memory. For instance:

$$\begin{array}{ll} \mathbf{newtype}\ \mathsf{A} = & \mathsf{A\ Int} \\ \mathbf{data} \quad\ \ \mathsf{B} = & \mathsf{B\ Int} \end{array}$$

In this example $\mathsf{A}\ 5$ have the same representation in memory as 5, whereas the representation of $\mathsf{B}\ 5$ would include a representation of the constructor B. As a consequence the constructor A is strict and the B constructor is lazy.

1.2.4 Records

Haskell provides a syntax for defining records with named fields. For instance we can define

$$\mathbf{data}\ \mathsf{Person} \ = \ \mathsf{P}\ \{\ \mathit{name} \ ::\ \mathsf{String}\ ,\ \mathit{age} \ ::\ \mathsf{Int}\ ,\ \mathit{male} \ ::\ \mathsf{Bool}\ \}$$

A record datatype can be treated in the same way as a normal datatype—the type of the constructor is $\mathsf{P} :: \mathsf{String} \to \mathsf{Int} \to \mathsf{Bool} \to \mathsf{Person}$ and we can ignore the records when pattern matching on a Person—but we get a few extra features: record projection functions, a facility for updating a record and special syntax for pattern matching on a record. In this dissertation we will use records just to get the projection functions and otherwise treat record datatypes as if they were normal datatypes. In the example above we would get the projection functions

$$\begin{array}{ll} \mathit{name} & :: \mathsf{Person} \to \mathsf{String} \\ \mathit{age} & :: \mathsf{Person} \to \mathsf{Int} \\ \mathit{male} & :: \mathsf{Person} \to \mathsf{Bool} \end{array}$$

1.2.5 Higher-rank polymorphism

A polymorphic type is a type that contains universally quantified type variables. Examples of functions with polymorphic types are

$$
\begin{aligned}
&\mathit{id} &&:: \forall a.\ a \to a\\
&\mathit{length} &&:: \forall a.\ [a] \to \mathsf{Int}\\
&\mathit{map} &&:: \forall a\ b.\ (a \to b) \to [a] \to [b]
\end{aligned}
$$

Haskell 98 supports rank-1 polymorphism, that is, universal quantifiers are only allowed on the top level of a type, as in the examples above. In these cases the quantifiers are not given explicitly, instead all free type variables are implicitly universally quantified at the top level[1]. Some Haskell implementations, most notably GHC, supports arbitrary rank polymorphism, where quantifiers are allowed anywhere. An example where this can be useful involves generalized rose trees:

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{GRose}\ f\ a\ =\ \mathsf{Branch}\ a\ (f\ (\mathsf{GRose}\ f\ a))\\
&\\
&\mathit{mapGRose}\ ::\ \forall f\ a\ b.\ (\forall s\ t.\ (s \to t) \to f\ s \to f\ t) \to\\
&\qquad\qquad (a \to b) \to \mathsf{GRose}\ f\ a \to \mathsf{GRose}\ f\ b\\
&\mathit{mapGRose}\ h\ f\ (\mathsf{Branch}\ x\ t)\ =\ \mathsf{Branch}\ (f\ x)\ (h\ (\mathit{mapGRose}\ h\ f)\ t)
\end{aligned}
$$

The map function for a generalized rose tree takes two arguments; a map function for its type argument f and the function that should be mapped over the rose tree. For $f = []$ we have

$$\mathit{mapGRose}\ \mathit{map}\ ::\ \forall\ a\ b.\ (a \to b) \to \mathsf{GRose}\ []\ a \to \mathsf{GRose}\ []\ b$$

We need higher rank polymorphism when defining generic functions for datatypes of higher kinds.

1.2.6 Sums and products

When writing generic programs we often view a datatype as a sum of products. One natural way of representing sums and products is to use the sum and product types from the Haskell Prelude.

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{Either}\ a\ b\ =\ \mathsf{Left}\ a\ |\ \mathsf{Right}\ b\\
&\mathbf{data}\ (a, b)\ \ \ \ \ =\ (a, b)
\end{aligned}
$$

Note that the pair type is a built-in type and the definition above is not legal Haskell. To manipulate elements of these types we extend what the Prelude defines with two mapping functions:

$$
\begin{aligned}
&(-\!\!+\!\!-)\ ::\ (a \to c) \to (b \to d) \to \mathsf{Either}\ a\ b \to \mathsf{Either}\ c\ d\\
&(f\ -\!\!+\!\!-\ g)\ (\mathsf{Left}\ x)\ \ =\ \mathsf{Left}\ (f\ x)\\
&(f\ -\!\!+\!\!-\ g)\ (\mathsf{Right}\ y)\ =\ \mathsf{Right}\ (g\ y)\\
&(-\!\!*\!\!-)\ ::\ (a \to c) \to (b \to d) \to (a, b) \to (c, d)\\
&(f\ -\!\!*\!\!-\ g)\ (x, y)\ =\ (f\ x, g\ y)
\end{aligned}
$$

[1] This is sometimes referred to as let-polymorphism.

1.2.7 Infix type constructors

In Haskell you can define your own infix operators and data constructors as we did in Section 1.2.6. What you cannot do in Haskell 98 is define infix type constructors. GHC, however, allows this. We can, for instance, define an infix product type constructor as follows:

```
data a :*: b = a :*: b
```

Here the type constructor and the data constructor have the same name (:*:). As with infix data constructors, infix type constructors have to start with a colon.

1.2.8 Multi-parameter type classes

Haskell's type classes allows for a powerful and structured form of overloading. A simple example is the monoid class defined in the Data.Monoid module of the standard Haskell libraries.

```
class Monoid m where
    mempty  :: m
    mappend :: m → m → m
instance Monoid [a] where
    mempty  = []
    mappend = (++)
```

The class declaration states that a type m is a Monoid if there are two functions *mempty* and *mappend* of the appropriate types. Note that there exist no facility for stating the laws that we expect these operations to satisfy. In the instance declaration we provide these functions for the list datatype declaring it to be a Monoid. In Haskell 98 type classes can only have a single type argument, so we can only describe predicates over types, not relations between two or more types. Most Haskell implementations, however, allows type classes with multiple arguments. For instance we can define a class for collections as follows.

```
class Collection c e | c → e where
    empty  :: c
    add    :: e → c → c
    member :: e → c → Bool
instance Eq a ⇒ Collection [a] a where
    empty  = []
    add    = (:)
    member = elem
```

This states that c is a collection of es if there are functions *empty*, *add* and *member* of the above types. The "$| c \to e$" is a *functional dependency* [20] which declares that e is uniquely determined by c. In other words, a type c cannot be a collection for more than one element type. Functional dependencies provide important, and sometimes necessary,

information for resolving type ambiguities. An example of this is the *empty* function. The element type e does not occur in the type of *empty*, so if there could be several instances for the same collection type but different element types, we would have no way of choosing between them.

1.3 Overview

This dissertation consists of two papers outlined below. Both papers are co-authored by Patrik Jansson who has written some of the less technical parts and been very helpful in giving comments and suggestions for the rest. The technical contributions of the papers, however, are my own.

1.3.1 Prototyping Generic Programming

The first papers is a revised version of *Prototyping Generic Programming using Template Haskell* [29]. This paper identifies the need for a light-weight approach to implementing generic programming languages and proposes Template Haskell [32] as a means of meeting this need. Template Haskell is an extension to Haskell that makes it possible to write functions that generate Haskell code. These functions are evaluated at compile-time and the result becomes part of the program. Using this mechanism a generic function can be modeled as a function that given the structure of a datatype produces the specialized Haskell code for that particular datatype. To demonstrate these ideas we develop prototype implementations of PolyP [15] and Generic Haskell [4]—two generic programming extensions to Haskell.

1.3.2 Polytypic Programming in Haskell

The second paper is a revised and extended version of *Polytypic Programming in Haskell* [28]. This paper describes how Haskell's class system can be used to achieve generic programming in Haskell. We show how to express the generic functions from the polytypic library [16] as well as a few new generic functions not previously definable. A new version of PolyP (version 2) is described together with a compiler that produces Haskell code using the class system to implement genericity.

Chapter 2

Prototyping Generic Programming

Generic Programming deals with the construction of programs that can be applied to many different datatypes. This is achieved by parameterizing the generic programs by the structure of the datatypes on which they are to be applied. Programs that can be defined generically range from simple map functions through pretty printers to complex XML tools.

The design space of generic programming languages is largely unexplored, partly due to the time and effort required to implement such a language. In this paper we show how to write flexible prototype implementations of two existing generic programming languages, PolyP and Generic Haskell, using Template Haskell, an extension to Haskell that enables compile-time meta-programming. In doing this we also gain a better understanding of the differences and similarities between the two languages.

2.1 Introduction

Generic functional programming [13] aims to ease the burden of the programmer by allowing common functions to be defined once and for all, instead of once for each datatype. Classic examples are small functions like maps and folds [12], but also more complex functions, like parsers and pretty printers [18] and tools for editing and compressing XML documents [9], can be defined generically. There are a number of languages for writing generic functional programs [1, 5, 10, 11, 15, 19, 21], each of which has its strengths and weaknesses, and researchers in generic programming are still searching for *The Right Way*. Implementing a generic programming language is no small task, which makes it cumbersome to experiment with new designs.

In this paper we show how to use Template Haskell [32] to implement two generic programming extensions to Haskell: PolyP [15] and Generic Haskell [4, 10]. With this approach, generic functions are written in Haskell (with the Template Haskell extension), so there is no need for an external tool. Furthermore the support for code generation and manipulation in Template Haskell greatly simplifies the compilation of generic functions, thus making the implementations very lightweight and easy to experiment with. A disadvantage of this approach is that we do not get the nice syntax we can get with a custom made parser. Another problem is that it is not at all clear how to implement a generic

type system in this setting, so type checking is deferred to instantiation time.

The rest of this section gives brief introductions to Template Haskell, PolyP and Generic Haskell (GH). Section 2.2 compares PolyP and GH. Section 2.3 introduces the concepts involved in implementing generic programming using Template Haskell. Sections 2.4 and 2.5 outline our prototype implementations of PolyP and GH and Section 2.6 points to possible future work.

2.1.1 Template Haskell

Template Haskell [32] is a language extension implemented in the Glasgow Haskell Compiler that enables compile-time meta-programming. This means that we can define code generating functions that are run at compile-time. In short you can *splice* abstract syntax into your program using the $(...) notation and *lift* an expression to the abstract syntax level using the quasi-quotes [| ... |]. Splices and quasi-quotes can be nested arbitrarily deep. For example, it is possible to define the *printf* function with the following type:

$$printf \;::\; \mathsf{String} \to \mathsf{Q}\;\mathsf{Exp}$$

Here *printf* takes the format string as an argument and produces the abstract syntax for the *printf* function specialized to that particular format string. To use this function we can write, for instance

```
Main⟩ $(printf "x%d=%s") 3 "foo"
"x3=foo"
```

Template Haskell comes with libraries for manipulating the abstract syntax of Haskell. The result type Q Exp of the *printf* function models the abstract syntax of an expression. The type constructor Q is the quotation monad, that takes care of, for instance, fresh name generation and the Exp type is a normal Haskell datatype modeling Haskell expressions. Similar types exist for declarations (Dec) and types (Type).

It is interesting to note that lifting is done on type correct Haskell expressions, whereas splicing might produce ill-typed code. In other words $[| e |] :: \mathsf{Q}\;\mathsf{Exp}$ if and only if $e :: \tau$ for some τ, and $\$(e)$ is well-defined (but not necessarily well-typed) whenever $e :: \mathsf{Q}\;\mathsf{Exp}$. So for any expression $e :: \tau$, we have $\$([| e |]) = e$, but $[| \$(t) |] = t$, for $t :: \mathsf{Q}\;\mathsf{Exp}$, only when $\$(t)$ is well-typed. Since lifting creates fresh names for all bound variables the above equalities are modulo alpha-renaming.

The definition of *printf* might look a bit complicated with all the lifts and splices, but ignoring those we have precisely what we would have written in an untyped language.

```
printf :: String → Q Exp
printf fmt = prAcc fmt [| "" |]
  where
```

```
prAcc :: String → Q Exp → Q Exp
prAcc fmt r =
  case fmt of
    '%' : 'd' : f → [| λ n → $(prAcc f [| $r ++ show n |]) |]
    '%' : 's' : f → [| λ s → $(prAcc f [| $r ++ s |]) |]
    c : f         → prAcc f [| $r ++ [c] |]
    ""            → r
```

The *prAcc* function uses an accumulating parameter r containing (the abstract syntax of) an expression representing the string created so far. Every time we see a % code we add a lambda at the top level and update the parameter with the argument.

We can step through the example from the beginning of the section to get a better understanding of what is going on.

```
printf "x%d=%s"
= prAcc "x%d=%s" [| "" |]
= prAcc "%d=%s" [| $([| "" |]) ++ "x" |]
= prAcc "%d=%s" [| "x" |]
= [| λ n → $(prAcc "=%s" [| "x" ++ show n |]) |]
= [| λ n → $(prAcc "%s" [| "x" ++ show n ++ "=" |]) |]
= [| λ n → $([| λ s → $(prAcc "" [| "x" ++ show n ++ "=" ++ s |]) |]) |]
= [| λ n → λ s → $(prAcc "" [| "x" ++ show n ++ "=" ++ s |]) |]
= [| λ n → λ s → $([| "x" ++ show n ++ "=" ++ s |]) |]
= [| λ n → λ s → "x" ++ show n ++ "=" ++ s |]
```

The keen observer will note that this definition of *printf* is quadratic in the length of the format string. This is easy to fix but for the sake of brevity we chose the inefficient version, which is slightly shorter.

Template Haskell supports program reflection or reification, which means that it is possible to get hold of the type of a named function or the declaration that defines a particular entity. For example:

```
reifyType id    :: Q Type
reifyDecl Maybe :: Q Dec
```

We can use this feature to find the definitions of the datatypes that a generic function is applied to.

2.1.2 PolyP

PolyP [15, 28] is a language extension to Haskell for generic programming, that allows generic functions over unary regular datatypes. A regular datatype is a datatype with no function spaces, no mutual recursion and no nested recursion[1]. Examples of unary regular datatypes are [], Maybe and Rose:

[1] The recursive calls must have the same form as the left hand side of the definition.

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

Figure 2.1: Pattern functors

$$\textbf{data}\ \mathsf{Rose}\ a\ =\ \mathsf{Fork}\ a\ [\mathsf{Rose}\ a]$$

Note that in this chapter we implement a prototype version of PolyP version 1 [15] and not of PolyP version 2 described in Chapter 3 and in [28]. A reason for this is that PolyP 2 relies on a generic type system to infer the appropriate class constraints for the generic functions (see Section 3.3.5) and it is not clear how to implement type systems for our generic languages in this setting.

Generic programming in PolyP is based on the notion of *pattern functors*. Each datatype is associated with a pattern functor that describes the structure of the datatype. The different pattern functor combinators are shown in Figure 2.1. The $g :+: h$ pattern functor is used to model multiple constructors, $g :*: h$ and Unit model the list of arguments to the constructors, Par is a reference to the parameter type, Rec represents a recursive call, $d :@: g$ models an application of a regular datatype d and $\mathsf{Const}\ t$ is used for a constant type t. The pattern functors of the datatypes mentioned above are (the comments show the expanded definitions applied to two type variables p and r):

$$
\begin{array}{lcll}
\textbf{type}\ \mathsf{ListF} & = & \mathsf{Unit} :+: (\mathsf{Par} :*: \mathsf{Rec}) & \text{— } \mathsf{Either}\ ()\ (p, r) \\
\textbf{type}\ \mathsf{MaybeF} & = & \mathsf{Unit} :+: \mathsf{Par} & \text{— } \mathsf{Either}\ ()\ p \\
\textbf{type}\ \mathsf{RoseF} & = & \mathsf{Par} :*: ([] :@: \mathsf{Rec}) & \text{— } (p, [r])
\end{array}
$$

PolyP provides two functions *inn* and *out* to fold and unfold the top-level structure of a datatype. Informally, for any regular datatype D with pattern functor F, *inn* and *out* have the following types:

$$
\begin{array}{l}
inn\ ::\ F\ a\ (D\ a) \to D\ a \\
out\ ::\ D\ a \to F\ a\ (D\ a)
\end{array}
$$

Note that only the top-level structure is folded/unfolded.

A special construct, **polytypic**, is used to define generic functions over pattern functors by pattern matching on the functor structure. As an example, the definition of *fmap*2, a generic map function over pattern functors, is shown in Figure 2.2. Together with *inn* and *out* these polytypic functions can be used to define generic functions over regular datatypes. For instance:

$$
\begin{array}{l}
pmap\ ::\ (a \to b) \to D\ a \to D\ b \\
pmap\ f\ =\ inn \circ fmap2\ f\ (pmap\ f) \circ out
\end{array}
$$

```
polytypic fmap2 :: (a → c) → (b → d) → f a b → f c d
  = λ p r → case f of
              g :+: h → fmap2 p r —+— fmap2 p r
              g :*: h → fmap2 p r —*— fmap2 p r
              Unit    → const ()
              Par     → p
              Rec     → r
              d :@: g → pmap (fmap2 p r)
              Const t → id
```

Figure 2.2: The definition of *fmap*2 in PolyP version 1

The same polytypic function can be used to create several different generic functions. We can, for instance, use *fmap*2 to define generic cata- and anamorphisms (generalized folds and unfolds):

```
cata :: (F a b → b) → D a → b
cata φ = φ ∘ fmap2 id (cata φ) ∘ out

ana :: (b → F a b) → b → D a
ana ψ = inn ∘ fmap2 id (ana ψ) ∘ ψ
```

2.1.3 Generic Haskell

Generic Haskell [4, 10]. is an extension to Haskell that allows generic functions over datatypes of arbitrary kinds. Hinze [7] observed that the type of a generic function depends on the kind of the datatype it is applied to, hence each generic function in Generic Haskell comes with a generic (kind indexed) type. The kind indexed type associated with the generic map function is defined as follows:

```
type Map {[⋆]}       s t = s → t
type Map {[κ → ν]} s t = ∀ a b. Map {[κ]} a b → Map {[ν]} (s a) (t b)
```

For a kind κ, the kind of Map $\{[\kappa]\}$ is $\kappa \to \kappa \to \star$. Generic Haskell uses the $\{[$*kind brackets*$]\}$ to enclose kind arguments. The type of the generic map function *gmap* applied to a type t of kind κ can be expressed as

```
gmap {|t :: κ|} :: Map {[κ]} t t
```

The $\{|$*type brackets*$|\}$ encloses type arguments. The kind argument κ is often omitted, since it can be inferred from the type t. Following are the types of *gmap* for some standard datatypes.

$$
\begin{array}{lll}
gmap\ \{|t\ ::\ \kappa|\}\ ::\ Map\ \{[\kappa]\}\ t\ t & & \\
gmap\ \{|\ :+:\ |\}\ gmapA\ gmapB\ (\mathsf{Inl}\ a) & = & \mathsf{Inl}\ (gmapA\ a) \\
gmap\ \{|\ :+:\ |\}\ gmapA\ gmapB\ (\mathsf{Inr}\ b) & = & \mathsf{Inr}\ (gmapB\ b) \\
gmap\ \{|\ :*:\ |\}\ gmapA\ gmapB\ (a\ :*:\ b) & = & gmapA\ a\ :*:\ gmapB\ b \\
gmap\ \{|\mathsf{Unit}|\}\ \mathsf{Unit} & = & \mathsf{Unit} \\
gmap\ \{|\mathsf{Con}\ c|\}\ gmapA\ (\mathsf{Con}\ a) & = & \mathsf{Con}\ (gmapA\ a) \\
gmap\ \{|\mathsf{Label}\ l|\}\ gmapA\ (\mathsf{Label}\ a) & = & \mathsf{Label}\ (gmapA\ a) \\
gmap\ \{|\mathsf{Int}|\}\ n & = & n
\end{array}
$$

Figure 2.3: A generic map function in Generic Haskell

$$
\begin{array}{lll}
\textbf{data}\ a\ :+:\ b & = & \mathsf{Inl}\ a \mid \mathsf{Inr}\ b \\
\textbf{data}\ a\ :*:\ b & = & a\ :*:\ b \\
\textbf{data}\ \mathsf{Unit} & = & \mathsf{Unit} \\
\textbf{data}\ \mathsf{Con}\ a & = & \mathsf{Con}\ a \\
\textbf{data}\ \mathsf{Label}\ a & = & \mathsf{Label}\ a
\end{array}
$$

Figure 2.4: Structure types in Generic Haskell

$$
\begin{array}{ll}
gmap\ \{|\ \mathsf{Int}\ |\} & ::\ \mathsf{Int} \to \mathsf{Int} \\
gmap\ \{|\ []\ |\} & ::\ \forall\ a\ b.\ (a \to b) \to ([a] \to [b]) \\
gmap\ \{|\ \mathsf{Either}\ |\} & ::\ \forall\ a\ b.\ (a \to b) \to \\
 & \quad \forall\ c\ d.\ (c \to d) \to (\mathsf{Either}\ a\ c \to \mathsf{Either}\ b\ d)
\end{array}
$$

The kind indexed types follow the same pattern for all generic functions. A generic function applied to a type of kind $\kappa \to \nu$ is a function that takes a generic function for types of kind κ and produces a generic function for the target type of kind ν.

The generic functions in Generic Haskell are defined by pattern matching on the top-level structure of the type argument. Figure 2.3 shows the definition of the generic map function *gmap*, generalizing Haskell's *fmap* and PolyP's *fmap*2. The structure combinators are similar to those in PolyP. Sums and products are encoded by :+: and :*: and the empty product is called Unit. A difference from PolyP is that constructors and record labels are represented by the structure combinators Con c and Label l. The arguments (c and l) contain information such as the name and fixity of the constructor or label. A generic function must also contain cases for primitive types such as Int. The type of each clause is the type of the generic function instantiated with the structure type on the left. The definitions of the structure types are shown in Figure 2.4. Note that the arguments to Con and Label containing the name and fixity information are only visible in pattern matching and not in the actual types.

Generic Haskell contains many features that we do not cover here, such as type indexed types, generic abstraction and constructor cases.

2.2 Comparing PolyP and Generic Haskell

The most notable difference between PolyP and Generic Haskell is the set of datatypes available for generic programmers. In PolyP generic functions can only be defined over unary regular datatypes, while Generic Haskell allows generic functions over (potentially non-regular) datatypes of arbitrary kinds.

Examples of datatypes that are supported by Generic Haskell but not by PolyP are

```
  — Kind ⋆ → ⋆ → ⋆
data Tree′ a b          = Leaf a | Node b (Tree′ a b) (Tree′ a b)

  — Mutually recursive
data Exp a              = Var a |
                          App (Exp a) (Exp a) |
                          Let (Dec a) (Exp a)
data Dec a              = Fun a (Exp a)

  — Nested
data Bin a              = One a | Fork (Bin (a, a))
```

There is a trade-off here, in that more datatypes means fewer generic functions. In PolyP it is possible to define generic folds and unfolds such as *cata* and *ana* (see Section 2.1.2) that cannot be defined in Generic Haskell.

Even if PolyP and Generic Haskell may seem very different, their approaches to generic programming are very similar. In both languages generic functions are defined, not over the datatypes themselves, but over a structure type acquired by unfolding the top-level structure of the datatype. The structure types in PolyP and Generic Haskell are very similar. The differences are that in PolyP constructors and labels are not recorded explicitly in the structure type and the structure type is parameterized over recursive occurrences of the datatype. This is made possible by only allowing regular datatypes. For instance, the structure of the list datatype in the two languages is (with Generic Haskell's sums and products translated into Either, $(,)$ and $()$):

```
type ListF a r = Either       ()       (a, r)          — PolyP
type ListS a   = Either (Con ()) (Con(a, [a]))         — Generic Haskell
```

To transform a generic function over a structure type into a generic function over the actual datatype, conversion functions between the datatype and the structure type are needed. In PolyP they are called *inn* and *out* (described in Section 2.1.2) and they are primitives in the language. In Generic Haskell this conversion is done by the compiler and the conversion functions are not available to the programmer.

As mentioned above, generic functions in both languages are primarily defined over the structure types. This is done by pattern matching on a type code, representing the structure of the datatype. The type codes differ between the languages, because they model different sets of datatypes, but the generic functions are defined in very much the same way. The most significant difference is that in Generic Haskell the translations of

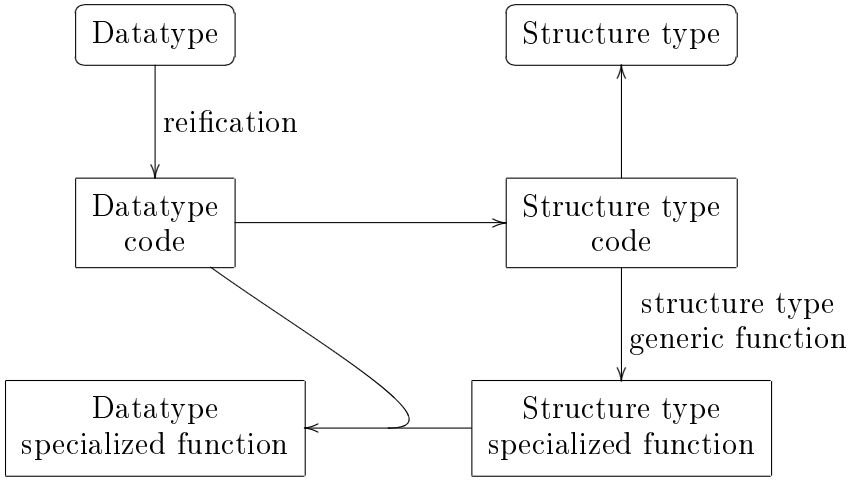

Figure 2.5: The implementation of a generic function.

type abstraction, type application and type variables are fixed and cannot be changed by the programmer, whereas in PolyP on the other hand the pattern functors Par and Rec gives the programmer access to the two abstracted type variables and the pattern functor d :@: g models type application.

Given a generic function over a structure type it should be possible to construct a generic function over the corresponding datatype. In Generic Haskell this process is fully automated and hidden from the programmer. In PolyP, however, it is the programmer's responsibility to take care of this. One reason for this is that the structure types are more flexible in PolyP, since they are parameterized over the recursive occurrences of the datatype. This means that there is not a unique datatype generic function for each structure type generic function. For instance the structure type generic function *fmap*2 from Figure 2.2 can be used not only to define the generic map function, *pmap*, but also the generic cata- and anamorphisms, *cata* and *ana*.

2.3 Guidelines for Implementing Generic Programming

Generic functions in both PolyP and Generic Haskell are defined by pattern matching over the code for a datatype. Such a generic function can be viewed as an algorithm for constructing a Haskell function given a datatype code. For instance, given the type code for the list datatype a generic map function can generate the definition of a map function over lists. Program constructing algorithms like this can be implemented nicely in Template Haskell; a generic function is simply a function from a type code to the abstract syntax for the function specialized to the corresponding type. When embedding a generic programming language like PolyP or Generic Haskell in Template Haskell there are a few things to consider:

- **Datatypes**

  The first thing to decide on is the set of datatypes over which we want to define

generic functions. For Generic Haskell this would be the full set of Haskell datatypes, whereas in PolyP the set is restricted to unary regular datatypes.

A large set of datatypes means more generically applicable functions, but often a more restricted set of definable generic functions.

- **Structure types**

  To be able to define functions generically over a set of datatypes it is necessary to have a uniform view of these datatypes. In PolyP and Generic Haskell this is achieved by assigning to each datatype a structure type capturing the top-level structure of that datatype. Now all generic manipulation of data can be done at the structure type level by translating back and forth between datatypes and structure types.

- **Type Codes**

  Regardless of how the generic programming languages is implemented the set of datatypes (and structure types) to be used has to be decided upon. The choice of datatype codes, on the other hand, is particular to the Template Haskell implementation of generic programming. As mentioned before, a generic function is implemented as a function from a type code (for a datatype) to abstract syntax (for the function specialized to that datatype). Thus the choice of codes can greatly influence how generic functions are defined.

  To be able to design the type codes we have to take a closer look at how they are used. Figure 2.5 sketches how generic functions are implemented in our framework. First of all we need two set of codes, one for datatypes and one for structure types. The structure type codes are used when defining generic functions over the structure types. For example, the Generic Haskell function *gmap* from Figure 2.3 and the PolyP function *fmap*2 from Figure 2.2 are defined in this way. The structure type codes are also used to generate the actual structure types when they need to be explicit [2].

  The structure type codes should be derivable from the datatype codes (just as structure types are derived from datatypes), so the datatype codes must contain at least the same information as the structure type codes. The datatype codes are also used when lifting functions from the structure type level to the datatype level.

  From this brief analysis we can conclude a few things about the type codes. Since structure type generic functions are defined by pattern matching on structure type codes we want to keep these codes as simple as possible. When we implement an existing generic programming language it is also desirable to make generic function definitions in the Template Haskell implementation resemble the definitions from the original language. When it comes to the datatype codes, they should contain the corresponding structure type code, but also enough information to generate conversion functions between datatype and structure type levels. The minimal information we need to do this is the names and arities of the datatype constructors.

[2] For instance, if explicit type signatures are required in the generated code.

- **Structure type generic function definitions**

  The structure type generic functions are where the real work is happening. Provided with a structure type code a structure type generic function generates the specialized version of the function to the corresponding structure type. Things to consider here include how to handle calls to other generic functions (or the same function on a different type). Consider, for example, the definition of the PolyP function *fmap*2 in Figure 2.2. What should be generated for the call to the generic function *pmap* in the d :@: g case? One choice is to inline the call, i.e. generate the body of *pmap* specialized to d and use that directly in the generated code. The other choice is to generate a call to a named function, *pmapd* say, that is defined as the specialization of *pmap* to d.

  In our PolyP implementation we choose the former and in the Generic Haskell implementation the latter.

- **Lifting functions on structure types to functions on datatypes**

  As can be seen in Figure 2.5 we want to transform the structure type specialization generated by the structure type generic function into a datatype specialization. This can be done in various ways and, as described in Section 2.2, PolyP and Generic Haskell take two different approaches to constructing these specializations. In PolyP it is the responsibility of the user whereas in Generic Haskell, it is done by the compiler. In any case we need to convert between an element of a datatype and an element of the corresponding structure type.

  In the approach taken by PolyP, the conversion functions (*inn* and *out*) are all the compiler needs to define. The programmer of a generic function will then use these to lift her function from the structure type level to the datatype level. Implementing the Generic Haskell approach on the other hand requires some more machinery. For each generic function, the compiler must convert the specialization for a structure type into a function that operates on the corresponding datatype (see Section 2.5).

- **Instantiation**

  Both the PolyP and Generic Haskell compilers do selective specialization, that is, generic functions are only specialized to the datatypes on which they are actually used in the program. This requires traversing the entire program to look for uses of generic functions. When embedding generic programming in Template Haskell we cannot analyze the entire program to find out which specializations to construct. One solution is to inline the body of the specialized generic function every time it is used. This makes the use of the generic functions easy, but special care has to be taken to avoid that recursive generic functions give rise to infinite specializations. This is the approach we use when embedding PolyP in Template Haskell (Section 2.4). Another approach is to require the user to decide which functions to specialize on which datatypes. This makes it harder on the user, but a little easier for the implementor of the generic programming language. Since our focus is on fast prototyping of generic languages, we use this approach when implementing Generic Haskell (Section 2.5).

2.4 PolyP in Template Haskell

Following the guidelines described in Section 2.3 we can start to implement our first generic programming language, PolyP.

2.4.1 Datatypes and Structure Types

Since we are implementing an existing language the choices of datatypes and structure types have already been made for us. PolyP allows generic functions over unary regular datatypes and the structure types are built up by the pattern functors (see Section 2.1.2). Worth noting here is that the datatypes and the structure types have different kinds ($\star \to \star$ and $\star \to \star \to \star$ respectively).

2.4.2 Type Codes

What we do have to decide on is how to design the codes for datatypes and structure types. Starting with the structure type codes we choose a representation matching PolyP's pattern functors as closely as possible.

```
data Code = Code :+: Code | Code :*: Code | Unit |
            Par | Rec | Regular :@: Code | Const Type
```

This coding corresponds perfectly to the definition of the pattern functors in Figure 2.1, we just have to decide what Type and Regular mean. The Template Haskell libraries define the abstract syntax for Haskell types in a datatype called Type so this is a natural choice to model types. The type Regular is nothing less than the type of datatype codes.

The datatype codes should contain enough information to generate the *inn* and *out* functions as discussed in Section 2.3, as well as the corresponding structure type code. So we simply choose to implement Regular as a pair of the constructor names and arities and a structure type code.

```
type Regular = ([(ConName, Int)], Code)

functorOf :: Regular → Code
functorOf = snd
```

To make it easy to get hold of the code for a datatype, we want to define a function that converts from the (abstract syntax of a) datatype definition to Regular. A problem with this is that one regular datatype might depend on another regular datatype, in which case we have to look at the definition of the second datatype as well. So instead of just taking the definition of the datatype in question our conversion function takes a list of all definitions that might be needed together with the name of the type to be coded.

```
regular :: [Q Dec] → TypeName → Regular
```

If a required datatype definition is missing, or the datatype is not regular, the call to *regular* will fail. But because the function is by the Template Haskell system at compile time, this

```
fmap2 :: Code → Q Exp
fmap2 f =
  [| λ p r → $(
    case f of
      g :+: h → [| $(fmap2 g) p r -+- $(fmap2 h) p r |]
      g :*: h → [| $(fmap2 g) p r -*- $(fmap2 h) p r |]
      Unit    → [| const () |]
      Par     → [| p |]
      Rec     → [| r |]
      d :@: g → [| $(pmap d) ($(fmap2 g) p r) |]
      Const t → [| id |]
  )
  |]
```

Figure 2.6: *fmap*2 in Template Haskell

will result in a compile-time error rather than a run-time error. Note that *regular* has to escape from the quotation monad somehow and since the quotation monad is built on top of the IO monad there is only one way of doing that: *unsafePerformIO*. This is perfectly safe here because the only IO computation performed is fresh name generation and we do not rely on names being fresh.

As an example, combining the function *regular* with Template Haskell's reification mechanism we obtain the code for the Rose datatype defined in Section 2.1.2.

```
roseD = regular [reifyDecl Rose, reifyDecl []] "Rose"
```

2.4.3 Structure Type Generic Functions

Generic functions over pattern functors are implemented as functions from structure type codes to (abstract) Haskell code. For example, the function *fmap*2 from Figure 2.2 in Section 2.1.2 is implemented as shown in Figure 2.6. The two definitions are strikingly similar, but there are a few important differences, the most obvious one being the splices and quasi-quote brackets introduced in the Template Haskell definition. Another difference is in the type signature. PolyP has its own type system capable of expressing the types of generic functions, but in Template Haskell everything inside quasi-quotes has type Q Exp, and thus the type of *fmap*2 is lost. The third difference is that in Template Haskell we have to pass the type codes explicitly to the recursive calls of *fmap*2.

The (:@:)-case in the definition of *fmap*2 calls the datatype level function *pmap* described in Section 2.4.4 to map over the regular datatype d. As mentioned in Section 2.3, there is a choice here of how to handle the call to *pmap*. One way of doing it would be to call it by name, i.e. assume that the specialization of *pmap* to the datatype d is available as a function of a particular name and just call that function. The other way, which we choose here, is to inline the specialization of *pmap*. This has the advantage of being

simpler—we do not need to bother with names of specializations—but it might lead to a code blowup. In PolyP mutually recursive datatypes are not allowed so the code blowup is not a big problem.

2.4.4 Datatype Generic Functions

In the previous subsection we saw how to define the structure type generic functions in the Template Haskell implementation of PolyP. Now we have to turn these into datatype generic functions. In PolyP datatype generic functions are defined in terms of the structure type generic functions and the functions *inn* and *out*, that convert between a datatype and its structure type. To generate these functions we need to know the constructor names and arities of the datatype as well as the corresponding structure type. This is precisely what (for good reason) the datatype codes Regular contain. We define

```
inn, out :: Regular → Q Exp
```

Applied to the code for the list datatype *inn* and *out* produce the following:

```
listD = regular [reifyDecl []] "[]"

$(inn listD) :: Either () (a, [a]) → [a]
$(inn listD) = λxs → case xs of
                       Left ()       → []
                       Right (x, xs) → x : xs
$(out listD) :: [a] → Either () (a, [a])
$(out listD) = λxs → case xs of
                       []     → Left ()
                       x : xs → Right (x, xs)
```

Basically we have to generate a case expression with one branch for each constructor. In the case of *out* we match on the constructor and construct a value of the structure type whereas *inn* matches on the structure type and creates a value of the datatype. Note that the arguments to the constructors (here x and xs) are left untouched, in particular the tail of the list is not unfolded.

With *inn* and *out* at our disposal we define the generic map function over a regular datatype, *pmap*. The definition is shown in Figure 2.7 together with the same definition in PolyP. In PolyP, *pmap* is a recursive function and following our decision to inline calls to generic functions we might be tempted to define it recursively in Template Haskell as well. This is not what we want, because it would make the generated code infinite. Instead we have to adopt the second approach and name the specialization of *pmap* to d, effectively generating a recursive function.

2.4.5 Instantiation

The generic functions defined in this style are very easy to use. To map a function f over a rose tree *tree* we simply write

```
  — PolyP definition
pmap :: Regular d ⇒ (a → b) → d a → d b
pmap f = inn ∘ fmap2 f (pmap f) ∘ out

  — Template Haskell definition
pmap :: Regular → Q Exp
pmap d = [| let pmapd f = $(inn d)
                          ∘ $(fmap2 (functorOf d)) f (pmapd f)
                          ∘ $(out d)
            in pmapd
          |]
```

Figure 2.7: The *pmap* function in PolyP and Template Haskell

$(*pmap roseD*) *f tree*

where *roseD* is the representation of the rose tree datatype defined in Section 2.4.2.

In summary, the prototype implementation of PolyP consists of the following

- Definitions of the type codes Code and Regular.
- A function *regular* that computes the code of a particular datatype, and the function *functorOf* extracting the structure type code from a datatype code.
- The generic functions *inn* and *out*, taking a Regular and generating conversion functions between the corresponding datatype and structure type.

Using this machinery generic functions can be defined almost as smoothly as in the full-blown implementation. And here we can easily experiment with extensions, such as optimizations of the generated code.

2.5 Generic Haskell in Template Haskell

In Section 2.4 we outlined an embedding of PolyP into Template Haskell. In this section we do the same for the core part of Generic Haskell. Features of Generic Haskell that we do not consider include constructor cases, type indexed types and generic abstraction. Type indexed types and generic abstraction should be possible to add without much difficulty; constructor cases might require some more work, though.

2.5.1 Datatypes and Structure Types

The set of datatypes over which we can define generic functions in Generic Haskell is the full set of arbitrarily kinded Haskell datatypes. Similarly to PolyP, Generic Haskell uses binary sums and products to model datatype structures (see Figure 2.4). We diverge from

Generic Haskell in this implementation in that we use the Haskell Prelude types Either and $(,)$ for sums and products instead of using the (isomorphic) versions from Generic Haskell. Another difference between our Template Haskell implementation and standard Generic Haskell is that constructors and labels are not modeled in the structure type. Compare, for example, the structure types of the list datatype in standard Generic Haskell (first) to our implementation (second):

```
type ListS a = Con Unit :+: Con(a :*: [a])      — standard Generic Haskell
type ListS a = Either () (a, [a])               — prototype
```

Since we are not implementing all features of Generic Haskell, we can allow ourselves this simplification.

2.5.2 Type Codes

To be consistent with how generic functions are defined in Generic Haskell we choose the following datatype for structure type codes:

```
data Code = Sum | Prod | Unit |
            Con ConDescr | Label LabelDescr |
            Fun | TypeCon TypeName |
            App Code Code | Lam VarName Code | Var VarName
```

The first seven constructors should be familiar to users of Generic Haskell, although you do not see the TypeCon constructor when matching on a specific datatype in Generic Haskell. The last three constructors App, Lam and Var you never see in Generic Haskell. The reason why they are not visible is that the interpretation of these type codes is hard-wired into Generic Haskell and cannot be changed by the programmer. By making them explicit we get the opportunity to experiment with this default interpretation.

The types ConDescr and LabelDescr describe the properties of constructors and labels. In our implementation this is just the name, but it could also include information such as fixity and strictness.

In our implementation of PolyP the datatype codes extended the structure type codes with constructor names and arities. Now this information is encoded already in the structure type codes, so the question is do the datatype codes need anything else. As it turns out, the answer to this question is yes. We need to know the name and kind of the datatype when generating the datatype specializations. The name, since we want to name the specialization, and the kind, because the type of the specialization depends on the kind of the datatype. Hence we define

```
data Kind     = ⋆ | Kind :→: Kind
type Datatype = (TypeName, Kind, Code)
```

If we define the infix application of App to be left associative, we can write the type code for the list datatype as follows:

```
listD :: Datatype
listD = ( "[]"
        , ⋆ :↠: ⋆
        , Lam "a" $
            Sum `App` (Con "[]" `App` Unit)
                `App` (Con " : " `App` (Prod `App` Var "a"
                                             `App` (TypeCon "[]" `App` Var "a")
                                       )
                      )
        )
```

This is not something we want to write by hand for every new datatype, even though it can be made much nicer by some suitable helper functions. Instead we define a function that produces a datatype code given the abstract syntax of a datatype declaration.

```
datatype :: Kind → Q Dec → Datatype
structureCode :: Datatype → Code
structureCode (_, _, c) = c
```

Thus, to get the above code for the list datatype we just write

```
listD = datatype (⋆ :↠: ⋆) (reifyDecl [])
```

Unfortunately Template Haskell cannot tell us the kind of a datatype. In principle we could infer it, but for simplicity we choose to have the user provide it explicitly.

2.5.3 Generic Function Types

The type of a generic function in Generic Haskell depends on the kind of the datatype the function is applied to, a fact first observed by Hinze [7]. At a glance it would seem like we could ignore the types of generic functions, because Template Haskell does not have any support for typing anyway. It turns out, however, that we need the type when generating the datatype specializations. The reason for this is that the types of the specializations use higher rank polymorphism and thus we need to generate explicit type signatures.

In the previous section we borrowed the representation of types from Template Haskell. In this case, however, we want to have explicit type abstractions, something that is missing from the Template Haskell representation[3]. Our datatype for types is shown in Figure 2.8. Now we can define the kind indexed type, Map from Section 2.1.3 as

```
typeMap ⋆           = LamT ["s", "t"] (VarT "s" ⟶ VarT "t")
typeMap (κ :↠: ν)   = LamT ["s", "t"] (ForallT ["a", "b"]
                        (   typeMap κ @ a @ b ⟶
                            typeMap ν @ (s @ a) @ (t @ b)
                        )
                      )
```

[3]Explicit type abstractions are not strictly necessary, but they make things much easier.

```
data Type = ForallT [VarName] Context Type |
            VarT VarName | ConT ConName |
            TupleT Int | ListT | ArrowT |
            AppT Type Type |
            LamT [VarName] Type
infixl 9 @
a @ b     = AppT a b
a ⟶ b  = ArrowT @ a @ b
```

Figure 2.8: A datatype for types

```
  where [s, t, a, b] = map VarT ["s", "t", "a", "b"]
```

This is slightly more clumsy than the Generic Haskell syntax, but we can make things a lot easier by observing that all type indexed types follow the same pattern. The only thing we need to know is the number of generic and non-generic arguments and the type for kind $\star$. With this information we define the function *kindIndexedType*:

```
kindIndexedType :: Int          — number of generic arguments
                → Int           — number of non−generic arguments
                → Type          — type for kind ⋆
                → Kind → Type
```

Using this function we can define the kind indexed type *typeMap* as

```
typeMap = kindIndexedType 2 0 (LamT ["s", "t"] (VarT "s" ⟶ VarT "t"))
```

Recall from Section 2.1.3 that the type of a generic function applied to a datatype t of kind κ is given on the form $T\ \{[\kappa]\}\ t\ \ldots\ t$, where T is a kind indexed type. Inspired by this we define a GenericType to be a function from a kind and a type to a type.

```
type GenericType = Kind → Type → Type
```

Now, the type of the generic map function from *gmap* can be defined as

```
gmapType :: GenericType
gmapType κ t = typeMap κ @ t @ t
```

2.5.4 Structure Type Generic Functions

A structure type generic function should take a structure type code and produce abstract Haskell syntax, so we define

```
type GenericFun = Code → Q Exp
```

```
defaults :: VarName → GenericFun′ → GenericFun′
defaults name gfun env t =
  case t of
    Con _      → [| id |]
    Label _    → [| id |]
    TypeCon c  → varE (gName name c)
    App s t    → [| $(gfun env s) $(gfun env t) |]
    Lam x t    → [| λ gx → $(gfun (addToEnv x [| gx |] env) t) |]
    Var x      → lookupEnv x env
varE :: String → Q Exp
```

Figure 2.9: Default generic translations

However, our structure type codes contain abstractions, which means that we have to deal with free variables. The standard way of doing this is to pass around an environment in which we can store what to do with variables.

```
type GEnv        = Env VarName (Q Exp)
type GenericFun′ = GEnv → GenericFun
emptyEnv  :: Env k v
addToEnv  :: Ord k ⇒ k → v → Env k v → Env k v
lookupEnv :: Ord k ⇒ k → Env k v → v
```

With these types at our disposal we define a function *defaults*, that handles default cases that will be the same for most generic functions. The function, defined in Figure 2.9, takes the name of the generic function that is being constructed and a GenericFun′ that handles the non-standard cases and produces a new GenericFun′. The idea is that a generic function should call *defaults* on all type codes that it does not handle (see the generic map function in Figure 2.10 for an example).

Let us look at the default cases a bit closer, starting from the bottom. The last three cases handle type application, abstraction and variables.

```
App s t → [| $(gfun env s) $(gfun env t) |]
Lam x t → [| λ gx → $(gfun (addToEnv x [| gx |] env) t) |]
Var x   → lookupEnv x env
```

The default case for a type application App s t is to generate a value application. This fits well with the notion of kind indexed types: Assume that s is the code for a type of kind $\kappa \to \nu$ and that t is the code for a type of kind κ, then (ignoring the details) we have

```
$(gfun env s) :: T {[κ → ν]} ...
$(gfun env t) :: T {[κ]} ...
```

for some kind indexed type T. Now, remember that a kind indexed type applied to a higher kind, $T\ \{[\kappa \to \nu]\}\ \ldots$, is basically a function type $T\ \{[\kappa]\}\ \ldots \to T\ \{[\nu]\}\ \ldots$.

Analogously for a type abstraction Lam x t, we generate a value abstraction taking a function gx of the appropriate type and producing the specialization of the generic function to the body t where gx is used as the specialization for the variable x.

These three cases cannot be changed by the user in Generic Haskell. In our prototype implementation, on the other hand, nothing prevents a generic function to handle these cases differently.

For a type constructor, the default case is to call the, presumably generated, specialization of the generic function to that type.

```
TypeCon c → varE (gName name c)
```

The function *varE* comes with Template Haskell and turns a string into the abstract syntax of an identifier. We define the function *gName* to return the name of the specialization of a generic function to a particular type.

When implementing the PolyP prototype we chose to inline calls to generic functions. This was feasible, because PolyP does not allow mutually recursive datatypes. In Generic Haskell, on the other hand, mutually recursive datatype are allowed, in which case inlining would lead to infinite programs.

The default cases for constructors and labels are defined to be the identity function.

```
Con _   → [| id |]
Label _ → [| id |]
```

To understand why this is a reasonable choice we have to look at what the Con c and Label l structure type codes mean. Recall from Section 2.5.1 that constructors and labels are not modeled in the structure types. This means that the code Con c `App` t represents the same structure type as the code t. Now consider what the default cases would produce for this code:

```
defaults name gfun env (Con c `App` t)
 == [| $(gfun env (Con c)) $(gfun env t) |]
 == ...
 == [| $(defaults name gfun env (Con c)) $(gfun env t) |]
 == [| id $(gfun env t) |]
 == gfun env t
```

In other words the default action for constructors and labels is to ignore them, and the reason why we can do this is that they are not modeled in the structure type.

Provided that it does not need to change the default actions, a generic function only has to provide actions for Sum, Prod and Unit. The definition of the generic map function from Section 2.1.3 is shown in Figure 2.10. For Sum and Prod the generic map function returns the map functions for Either and $(,)$, and mapping over the unit type is just the identity functions. For all other type codes we call the *defaults* function to perform the default actions.

```
gmap :: GenericFun
gmap t = gmap' emptyEnv t
  where gmap' :: GenericFun'
        gmap' env t =
          case t of
            Sum  → [| (—+—) |]
            Prod → [| (—*—) |]
            Unit → [| id |]
            t    → defaults "gmap" gmap' env t
```

Figure 2.10: Generic map

2.5.5 Datatype Generic Functions

In our prototype implementation of PolyP (Section 2.4) it was the responsibility of the programmer lift structure type generic functions to datatype generic functions. The reason for this was that in PolyP, the conversion could be done in several different ways, yielding different datatype functions. In Generic Haskell, on the other hand, we have a unique datatype function in mind for every structure type function. For instance, look at the type of the function generated by applying *gmap* to the code for the list structure type:

```
gmap_ListS :: (a → b) → Either () (a, [a]) → Either () (b, [b])
gmap_ListS = $(gmap (structureCode listD))
```

From this function we want to generate the map function for lists with type

```
gmap_[] :: (a → b) → [a] → [b]
```

The first step is to generate the conversion functions between a datatype and its structure type. For this purpose we define the function *structEP*.

```
structEP :: Datatype → Q Dec
```

This function generates a declaration of the conversion functions, so for the list datatype it would generate something like the following:

```
ep_[] :: EP [a] (Either () (a, [a]))
ep_[] = EP out inn
    where out []             = Left ()
          out (x : xs)       = Right (x, xs)
          inn (Left ())      = []
          inn (Right (x, xs)) = x : xs
```

The EP type, shown in Figure 2.11, models an embedding projection pair.

$$\textbf{data}\ \mathsf{EP}\ a\ b\ =\ \mathsf{EP}\ \{\ from\ ::\ a \to b,\ to\ ::\ b \to a\ \}$$

$$ep_{id}\ ::\ \mathsf{EP}\ a\ a$$
$$ep_{id}\ =\ \mathsf{EP}\ id\ id$$

$$(\xrightarrow{ep})\ ::\ \mathsf{EP}\ a\ a' \to \mathsf{EP}\ b\ b' \to \mathsf{EP}\ (a \to b)\ (a' \to b')$$
$$eqA \xrightarrow{ep} epB\ =\ \mathsf{EP}\ (\lambda f \to\ from\ epB \circ f \circ to\quad epA)$$
$$(\lambda g \to\ to\quad epB \circ g \circ from\ epA)$$

Figure 2.11: Embedding projection pairs

Using $ep_{[]}$ we can define the map function for the list datatype as

$$gmap_{[]}\ ::\ (a \to b) \to [a] \to [b]$$
$$gmap_{[]}\ f\ =\ to\ (ep_{[]} \xrightarrow{ep} ep_{[]})\ (gmap_{\mathsf{ListS}}\ f)$$

The embedding projection pair is generated directly from the type of the generic function. In this case an embedding projection pair of type

$$\mathsf{EP}\ ([a] \to [b])\ (\mathsf{Either}\ ()\ (a, [a]) \to \mathsf{Either}\ ()\ (b, [b]))$$

should be generated. Embedding projection pairs between function types can be constructed with $(\xrightarrow{ep})$, and $ep_{[]}$ can convert between a list and an element of the list structure type. We define the function *typeEP* to generate the appropriate embedding projection pair.

$$typeEP\ ::\ \mathsf{Q\ Exp} \to \mathsf{Kind} \to \mathsf{GenericType} \to \mathsf{Q\ Exp}$$

The first argument to *typeEP* is the embedding projection pair converting between the datatype and its structure type, the second argument is the kind of the datatype and the third argument is the type of the generic function. So to get the embedding projection pair used in $gmap_{[]}$ we write

$$typeEP\ [|\ ep_{[]}\ |]\ (\star :\rightarrow: \star)\ gmapType$$

2.5.6 Instantiation

The focus of this article is on fast prototyping of generic programming languages. This means that we do not make great efforts to facilitate the use of the generic functions. In particular what we do not do is figuring out which specializations to generate. Instead we provide a function *instantiate* that generates the definition of the specialization of a generic function to a particular datatype, as well as a function *structure*, that generates the embedding projection pair definition converting between a datatype and its structure type using the function *structEP* from Section 2.5.5.

$$\textbf{type}\ \mathsf{Generic} = (\mathsf{VarName}, \mathsf{GenericType}, \mathsf{GenericFun})$$

$$\begin{aligned} &\mathit{instantiate} :: \mathsf{Generic} \to \mathsf{Datatype} \to \mathsf{Q}\ [\mathsf{Dec}] \\ &\mathit{structure} :: \mathsf{Datatype} \to \mathsf{Q}\ [\mathsf{Dec}] \end{aligned}$$

Using these functions a map function for rose trees can be generated by

$$\textbf{data}\ \mathsf{Rose}\ a = \mathsf{Fork}\ a\ [\mathsf{Rose}\ a]$$

$$\begin{aligned} &\mathit{listD} = \mathit{datatype}\ (\star :\twoheadrightarrow: \star)\ (\mathit{reifyDecl}\ []) \\ &\mathit{roseD} = \mathit{datatype}\ (\star :\twoheadrightarrow: \star)\ (\mathit{reifyDecl}\ \mathsf{Rose}) \\ &\mathit{gmapG} = (\text{“}\mathtt{gmap}\text{”}, \mathit{gmapType}, \mathit{gmap}) \\ &\$(\mathit{structure}\ \mathit{listD}) \\ &\$(\mathit{structure}\ \mathit{roseD}) \\ &\$(\mathit{instantiate}\ \mathit{gmapG}\ \mathit{listD}) \\ &\$(\mathit{instantiate}\ \mathit{gmapG}\ \mathit{roseD}) \end{aligned}$$

Since the rose trees contain lists we have to create specializations for the list datatype as well. The code generated for *gmap* specialized to rose trees will look something like the following (after some formatting and alpha renaming). Note that $\mathit{gmap}_{\mathsf{RoseS}}$ uses both $\mathit{gmap}_{[]}$ and $\mathit{gmap}_{\mathsf{Rose}}$.

$$\begin{aligned} &\mathit{gmap}_{\mathsf{RoseS}} :: (a \to b) \to (a, [\mathsf{Rose}\ a]) \to (b, [\mathsf{Rose}\ b]) \\ &\mathit{gmap}_{\mathsf{RoseS}} = \lambda f \to f \mathbin{-\!\!*\!\!-} \mathit{gmap}_{[]}\ (\mathit{gmap}_{\mathsf{Rose}}\ f) \end{aligned}$$

$$\begin{aligned} &\mathit{gmap}_{\mathsf{Rose}} :: (a \to b) \to \mathsf{Rose}\ a \to \mathsf{Rose}\ b \\ &\mathit{gmap}_{\mathsf{Rose}}\ f = \mathit{to}\ (\mathit{ep}_{\mathsf{Rose}} \xrightarrow{ep} \mathit{ep}_{\mathsf{Rose}})\ (\mathit{gmap}_{\mathsf{RoseS}}\ f) \end{aligned}$$

Our, now finished, prototype implementation of Generic Haskell contains the following:

- The type codes Datatype and Code.
- A function *datatype* that computes the code a datatype, and a function to extract the structure type code from a datatype code, *structureCode*.
- Datatypes Kind and *Typ* representing kinds and types.
- A function *kindIndexedType* that builds a kind indexed type.
- The *defaults* function that provides a default implementation for some common cases in a generic function.
- Two functions *structEP* that generates the conversion functions between a datatype and its structure type, and a function *typeEP* that generates conversion functions between a structure type generic function and a datatype generic function.
- The top-level functions *structure* and *instantiate* that uses *structEP* and *typeEP* to generate declarations that can be spliced into a Haskell program.

Compared to the implementation of PolyP from Section 2.4, the Generic Haskell prototype required more machinery. One reason for this is that Generic Haskell handles datatypes of arbitrary kinds, which adds complexity. Another reason is that in the PolyP implementation, more work was left to the programmer of the generic functions. All in all, writing generic functions in our prototype is not much more work than in the *real* language, although one should keep in mind that a few nice features from the full implementation are still missing in the prototype.

2.6 Conclusions and Future Work

Efforts to explore the design space of generic programming have been hampered by the fact that implementing a generic programming language is a daunting task. In this paper we have shown that this does not have to be the case. We have presented two prototype implementations of generic programming approximating PolyP and Generic Haskell. Thanks to the Template Haskell machinery, these prototypes could be implemented in a short period of time (each implementation consists of a few hundred lines of Haskell code). Comparing these two implementations we obtain a better understanding of the design space when it comes to implementations of generic programming languages.

There are a few different areas one might want to focus future work on:

- The idea of fast prototyping is to make it possible to experiment with different ideas in an easy way. So far, most of our work has been concentrated on how to write the prototypes and not so much on experimenting.

- One of the biggest problems with current generic programming systems is efficiency. The conversions between datatypes and structure types takes a lot of time and it would be a big win if one could remove this extra cost. We have started working on a simplifier for Haskell expressions that can do this based on [2].

- It would be interesting to see how other generic programming styles fit into this framework. In particular one could look at the `Data.Generics` libraries in GHC [21] and also at the generic traversals of adaptive OOP [24].

- The design of a generic programming language includes the design of a type system. In this paper we have ignored the issue of typing, leaving it up to the Haskell compiler to find type errors in the specialized code. Is there an easy way to build prototype type systems for our implementations?

Chapter 3

Polytypic Programming in Haskell

A polytypic (or generic) program captures a common pattern of computation over different datatypes by abstracting over the structure of the datatype. Examples of algorithms that can be defined polytypically are equality tests, mapping functions and pretty printers.

A commonly used technique to implement polytypic programming is specialization, where a specialized version of a polytypic function is generated for every datatype it is used at. In this paper we describe an alternative technique that allows polytypic functions to be defined using Haskell's class system (extended with multi-parameter type classes and functional dependencies). This technique brings the power of polytypic programming inside Haskell allowing us to define a Haskell library of polytypic functions. It also increases our flexibility, reducing the dependency on a polytypic language compiler.

3.1 Introduction

Functional programming draws great power from the ability to define polymorphic, higher order functions that can capture the structure of an algorithm while abstracting away from the details. A polymorphic function is parameterized over one or more types and thus abstracting away from the specifics of these types. The same is true for a polytypic (or generic) function, but while all instances of a polymorphic function share the same definition, the instances of a polytypic function definition also depend on a type.

By parameterizing the function definition by a type one can capture common patterns of computation over different datatypes. Examples of functions that can be defined polytypically include the map function that maps a function over the elements of a container datatype but also more complex algorithms like unification and term rewriting.

Even if an algorithm will only be used at a single datatype it may still be a good idea to implement it as a polytypic function. First of all, since a polytypic function abstracts away from the details of the datatype, we cannot make any datatype specific mistakes in the definition and secondly, if the datatype changes during algorithm development, there is no need to change the polytypic function.

A common technique to implement polytypic programming is to specialize the polytypic functions to the datatypes at which they are used. In other words the polytypic compiler generates a separate function for each polytypic function-datatype pair. Unfortunately

this implementation technique requires global access to the program using the polytypic functions. In this paper we describe an alternative technique to implement polytypic programs using the Haskell class system. The polytypic programs which can be defined are restricted to operate on regular, single parameter datatypes. That is, datatypes that are not mutually recursive and where the recursive calls all have the same form as the left hand side of the datatype definition. Note that datatypes are allowed to contain function spaces. This technique has been implemented as a Haskell library and as a modification of the PolyP [15] compiler resulting in PolyP version 2. The implementation of PolyP 2 is available from the polytypic programming home page [14]. In the following text we normally omit the version number — PolyP will stand for the improved language and its (new) compiler.

3.1.1 Overview

The rest of this paper is structured as follows. Section 3.2 describes how polytypic programs can be expressed inside Haskell. The structure of regular datatypes is captured by pattern functors (expressed using datatype combinators) and the relation between a regular datatype and its pattern functor is captured by a two parameter type class (with a functional dependency). In this setting a polytypic definition is represented by a class with instances for the different datatype combinators. Section 3.3 shows how the implementation of PolyP has been extended to translate PolyP code to Haskell classes and instances. Section 3.4 and Section 3.5 show two case studies of using PolyP and Haskell for polytypic programming. Section 3.6 describes related work and Section 3.7 concludes.

3.2 Polytypism in Haskell

In this section we show how polytypic programs can be embedded in Haskell[1]. The embedding uses datatype constructors to model the top level structure of datatypes, and the two-parameter type class FunctorOf to relate datatypes to their structures.

The embedding closely mimics the features of the language PolyP [15], an extension to (a subset of) Haskell that allows definitions of polytypic functions over regular, unary datatypes. This section gives a brief overview of the embedding and compares it to PolyP.

3.2.1 Datatypes and pattern functors

As mentioned earlier we allow definition of polytypic functions over regular datatypes of kind $\star \to \star$. A datatype is regular if it is not mutually recursive with another type and if the argument to the type constructor is the same in the left-hand side and the right-hand side of the definition. See Section 2.2 for examples of non-regular datatypes.

We describe the structure of a regular datatype by its *pattern functor*. A pattern functor is a two-argument type constructor built up using the combinators shown in Figure 3.1. Note that these are not the same definitions as the ones used in PolyP version 1 (Figure 2.1).

[1] The code for this paper works with current (October 2004) versions of the Glasgow Haskell Compiler and the `hugs`, and can be obtained from the polytypic programming home page [14].

```
data (g :+: h) p r       = InL (g p r) | InR (h p r)
data (g :*: h) p r       = g p r :*: h p r
data Unit p r            = Unit
newtype Par p r          = Par {unPar :: p}
newtype Rec p r          = Rec {unRec :: r}
newtype (d :@: g) p r    = Comp {unComp :: d (g p r)}
newtype Const t p r      = Const {unConst :: t}
newtype (g :->: h) p r   = Fun {unFun :: g p r -> h p r}
```

Figure 3.1: Pattern functor combinators

The infix combinators are right associative and their order of precedence is, from lower to higher: (:+:), (:*:), (:->:), (:@:). For the datatype List a we can use these combinators to define the pattern functor ListF as follows:

```
data List a = Nil | Cons a (List a)
type ListF  = Unit :+: Par :*: Rec
```

An element of ListF p r can take either the form InL Unit, corresponding to Nil or the form InR (Par x :*: Rec xs), corresponding to Cons x xs. A point worth noting is that the r does not have to be instantiated to List a. This is, in fact, what allows us to define polytypic fold and unfold functions as we will see in Section 3.2.3.

The pattern functor d :@: g represents the composition of the regular datatype constructor d and the pattern functor g, allowing us to describe the structure of datatypes like Rose:

```
data Rose a = Fork a (List (Rose a))
type RoseF  = Par :*: List :@: Rec
```

A constant type in a datatype definition is modeled by the pattern functor Const t. For instance, the pattern functor of a binary tree storing height information in the nodes can be expressed as

```
data HTree a = Leaf a | Branch Int (HTree a) (HTree a)
type HTreeF  = Par :+: Const Int :*: Rec :*: Rec
```

The pattern functor (:->:) is used to model datatypes with function spaces. Only a few polytypic functions are possible to define for such datatypes, see Section A.16 for an example.

In general we write Φ_D for the pattern functor of the datatype D a, so for example $\Phi_{\mathsf{List}} = \mathsf{ListF}$. To convert between a datatype and its pattern functor we use the methods *inn* and *out* in the multi-parameter type class FunctorOf:

```
class FunctorOf f d | d → f where
  inn             :: f a (d a) → d a
  out             :: d a → f a (d a)
  constructorName :: d a → String
  datatypeName    :: d a → String
```

Note that the class FunctorOf ranges over type constructors $f :: \star \to \star \to \star$ and $d :: \star \to \star$, rather than over base types of kind $\star$. The functions *inn* and *out* realize the isomorphism $d\ a \cong \Phi_d\ a\ (d\ a)$, that holds[2] for every regular datatype. (We can view a regular datatype $d\ a$ as a fixed point of the corresponding functor $\Phi_d\ a$.) The methods *constructorName* and *datatypeName* (and a few more for fixity and precedence information) are used in the definition of polytypic show and read functions.

In our list example we have

```
instance FunctorOf (Unit :+: Par :*: Rec) List where
  inn (InL Unit)                = Nil
  inn (InR (Par x :*: Rec xs))  = Cons x xs
  out Nil                       = InL Unit
  out (Cons x xs)               = InR (Par x :*: Rec xs)
  constructorName Nil           = "Nil"
  constructorName (Cons x xs)   = "Cons"
  datatypeName _                = "List"
```

Note that *inn* (*out*) only folds (unfolds) the top-level structure, leaving the substructures intact and it is therefore normally a constant time operation. The exception is for datatypes whose pattern functor contains a composition, d :@: g, like the Rose datatype above. In this case *inn* and *out* have to remove and add the pattern functor constructors inside the type d and thus have to traverse a potentially infinite structure. Fortunately this traversal can be done lazily so this is not a problem in practise.

The functional dependency $d \to f$ in the FunctorOf-class means that the set of instances defines a type level function from datatypes to their pattern functors. Several different datatypes can map to the same pattern functor if they share the same structure, but one datatype can not have more than one associated pattern functor. For an example where we use the fact the several datatypes can have the same structure see the function *coerce* in Section 3.2.6. The functional dependency allows the Haskell compiler to infer unambiguous types for most generic functions — without the dependency, disambiguating type annotations are often required.

3.2.2 Pattern functor classes

A polytypic function is a function that is parameterized by the structure of a datatype. In this setting datatype structures are modeled by pattern functors, so we expect a polytypic function to be defined by recursion over a pattern functor. Since pattern functors are types

[2]Since we are using a lifted binary product to represent the arguments to a datatype constructor we actually have $out \circ inn \sqsubseteq id$, so *inn* and *out* form an embedding-projection pair rather than an isomorphism.

rather than values we use the class system to achieve this recursion. For each polytypic definition f we define a *pattern functor class* P_f with a single method f. The pattern functor class is parameterized by a type of kind $\star \to \star \to \star$, the kind of the pattern functors and the set of instances for a class P_f corresponds to the clauses of the recursive definition of the polytypic function f.

An example is a generalization of the standard Haskell Prelude class Functor to the pattern functor class P_fmap2:

```
class Functor f where
  fmap :: (a → b) → (f a → f b)
class P_fmap2 f where
  fmap2 :: (a → c) → (b → d) → (f a b → f c d)
```

All pattern functors constructors except $(:\!\!\rightarrow\!\!:)$ are instances of the class P_fmap2, the full definition of *fmap*2 is shown in Figure 3.2, together with with the map function on regular datatypes, *pmap*. Pattern functor classes and their instances are discussed in more detail in Section 3.3, where we show how they can be generated from PolyP function definitions.

3.2.3 PolyLib in Haskell

PolyLib [16] is a library of polytypic definitions including generalized versions of well-known functions such as *map*, *zip* and *sum*, as well as powerful recursion combinators such as *cata*, *ana* and *hylo*. All these library functions have been converted to work with our new framework, so that PolyLib is now available as a normal Haskell library (see Appendix A for further details). The library functions can be used on all datatypes which are instances of the FunctorOf class. The FunctorOf-instances can be generated either by the PolyP 2 compiler, by using Template Haskell (Section A.2), or by defining them manually.

Using *fmap*2 from the P_fmap2-class and *inn* and *out* from the FunctorOf class we can already define quite a few polytypic functions from the Haskell version of PolyLib. For instance

```
pmap  :: (FunctorOf f d, P_fmap2 f) ⇒ (a → b) → (d a → d b)
cata  :: (FunctorOf f d, P_fmap2 f) ⇒ (f a b → b) → (d a → b)
ana   :: (FunctorOf f d, P_fmap2 f) ⇒ (b → f a b) → (b → d a)

pmap f = inn ∘ fmap2 f (pmap f) ∘ out
cata φ = φ ∘ fmap2 id (cata φ) ∘ out
ana ψ  = inn ∘ fmap2 id (ana ψ) ∘ ψ
```

We can use the functions above to define other polytypic functions. For instance, we can use *cata* to define a generalization of $sum :: \mathsf{Num}\ a \Rightarrow [a] \to a$ which works for all regular datatypes. Suppose we have a pattern functor class P_fsum with the method *fsum*:

```
class P_fsum f where
  fsum :: Num a ⇒ f a a → a
```

```
class P_fmap2 f where
  fmap2 :: (a → c) → (b → d) → (f a b → f c d)

instance (P_fmap2 g, P_fmap2 h) ⇒ P_fmap2 (g :+: h) where
  fmap2 p r (InL x) = InL (fmap2 p r x)
  fmap2 p r (InR y) = InR (fmap2 p r y)
instance (P_fmap2 g, P_fmap2 h) ⇒ P_fmap2 (g :*: h) where
  fmap2 p r (x :*: y) = fmap2 p r x :*: fmap2 p r y

instance P_fmap2 Unit where
  fmap2 p r Unit = Unit

instance P_fmap2 Par where
  fmap2 p r (Par x) = Par (p x)

instance P_fmap2 Rec where
  fmap2 p r (Rec x) = Rec (r x)

instance (FunctorOf f d, P_fmap2 f, P_fmap2 g) ⇒ P_fmap2 (d :@: g) where
  fmap2 p r (Comp x) = Comp (pmap (fmap2 p r) x)

instance P_fmap2 (Const t) where
  fmap2 p r (Const x) = Const x

pmap :: (FunctorOf f d, P_fmap2 f) ⇒ (a → b) → (d a → d b)
pmap f = inn ∘ fmap2 f (pmap f) ∘ out
```

Figure 3.2: Haskell definition of a polytypic map function.

(The method *fsum* takes care of summing the top-level, provided that the recursive occurrences have already been summed.) Then we can sum the elements of a regular datatype by defining

$$\begin{aligned} &psum \;::\; (\mathsf{FunctorOf}\; f\; d,\; \mathsf{P\_fmap2}\; f, \mathsf{P\_fsum}\; f, \mathsf{Num}\; a) \Rightarrow d\; a \to a \\ &psum \;=\; cata\; fsum \end{aligned}$$

We return to the function *fsum* in Section 3.3.1 when we discuss how the pattern functor classes are generated. In the type of *psum* we can see an indication of a problem that arises when combining polytypic functions without instantiating them to concrete types: we get large class constraints. Fortunately we can let the Haskell compiler infer the type for us in most cases, but our setting is certainly one which would benefit from extending Haskell type constraint syntax with wild cards, allowing us to write the type signature as $psum \;::\; \_ \Rightarrow d\; a \to a$.

3.2.4 Perfect binary trees

A benefit of using the class system to do polytypic programming is that it allows us to treat (some) non-regular datatypes as regular, thus providing a *regular view* of the datatype. For instance, take the nested datatype of perfect binary trees, defined by

$$\mathbf{data}\; \mathsf{Bin}\; a \;=\; \mathsf{Single}\; a \mid \mathsf{Fork}\; (\mathsf{Bin}\; (a, a))$$

Elements of $\mathsf{Bin}\; a$ take the form of a number of Forks followed by a Single containing all the elements in a structure of nested pairs. For example

$$\begin{aligned} &depth1 = \mathsf{Fork}\; (\mathsf{Single}\; (1,2)) \\ &depth2 = \mathsf{Fork}\; (\mathsf{Fork}\; (\mathsf{Single}\; ((1,2),(3,4)))) \\ &depth3 = \mathsf{Fork}\; (\mathsf{Fork}\; (\mathsf{Fork}\; (\mathsf{Single}\; (((1,2),(3,4)),((5,6),(7,8)))))) \end{aligned}$$

This type can be viewed as having the pattern functor Par :+: Rec :*: Rec, i.e. the same as the ordinary binary tree.

$$\mathbf{data}\; \mathsf{Tree}\; a \;=\; \mathsf{Leaf}\; a \mid \mathsf{Branch}\; (\mathsf{Tree}\; a)\; (\mathsf{Tree}\; a)$$

By defining an instance of the FunctorOf class for Bin (see Figure 3.3) we can then use all the PolyLib functions on perfect binary trees. For instance we can use an anamorphism to generate a full binary tree of a given height as follows.

$$\begin{aligned} &full \;::\; a \to \mathsf{Int} \to \mathsf{Bin}\; a \\ &full\; x \;=\; ana\; (step\; x) \\ &\quad \mathbf{where}\; step\; x\; 0 \;\;\;\;\;\;\;\;\;\; = \mathsf{InL}\; (\mathsf{Par}\; x) \\ &\qquad\qquad step\; x\; (n+1) = \mathsf{InR}\; (\mathsf{Rec}\; n)\; (\mathsf{Rec}\; n) \end{aligned}$$

By forcing the perfect binary trees into the regular framework we (naturally) loose some type information. Had we, for instance, made a mistake in the definition of *full* so that it did not generate a full tree, we would get a run-time error (pattern match failure in *join*) instead of a type error.

```
instance FunctorOf (Par :+: Rec :*: Rec) Bin where
  inn (InL (Par x))              = Single x
  inn (InR (Rec l :*: Rec r))    = Fork (join (l, r))
  out (Single x)                 = InL (Par x)
  out (Fork t)                   = InR (Rec l :*: Rec r)
      where (l, r)               = split t
  constructorName (Single _) = "Single"
  constructorName (Fork _)   = "Fork"
  datatypeName _             = "Bin"

join :: (Bin a, Bin a) → Bin (a, a)
join (Single x, Single y) = Single (x, y)
join (Fork l, Fork r)     = Fork (join (l, r))

split :: Bin (a, a) → (Bin a, Bin a)
split (Single (x, y)) = (Single x, Single y)
split (Fork t)        = (Fork l, Fork r)
    where (l, r) = split t
```

Figure 3.3: A FunctorOf instance for perfect binary trees

3.2.5 Abstract datatypes

In the previous example we provided a regular view on a non-regular datatype. We can do the same thing for (some) abstract datatypes. Suppose we have an abstract datatype Stack, with methods

```
push  :: a → Stack a → Stack a
pop   :: Stack a → Maybe (a, Stack a)
empty :: Stack a
```

By giving the following instance, we provide a view of the stack as a regular datatype with the same pattern functor as normal lists: Unit :+: Par :*: Rec.

```
instance FunctorOf (Unit :+: Par :*: Rec) Stack where
  inn (InL Unit)                = empty
  inn (InR (Par x :*: Rec s)) = push x s

  out s = case pop s of
            Nothing    → InL Unit
            Just (x, s') → InR (Par x :*: Rec s')
  constructorName s = case pop s of
                        Nothing → "empty"
                        Just _  → "push"
  datatypeName _    = "Stack"
```

As in the previous example, this instance allows us to use polytypic functions on stacks,

for instance applying the function *psum* to a stack of integers or using *pmap* to apply a function to all the elements on a stack.

3.2.6 Polytypic functions in Haskell

We have seen how to make different kinds of datatypes fit the polytypic framework, thus enabling us to use the polytypic functions from PolyLib on them, but we can also use the PolyLib functions to create new polytypic functions. One interesting function that we can define is the function *coerce*

$$\begin{aligned} &coerce \;::\; (\mathsf{FunctorOf}\; f\; d,\; \mathsf{FunctorOf}\; f\; e,\; \mathsf{P\_fmap2}\; f) \Rightarrow d\; a \to e\; a \\ &coerce \;=\; cata\; inn \end{aligned}$$

that converts between two regular datatypes with the same pattern functor. For instance we could convert a perfect binary tree from Section 3.2.4 to a normal binary tree or convert a list to an element of the abstract stack type from Section 3.2.5.

Another use of polytypic functions in Haskell is to define default instances of the standard type classes. For instance we can define

$$\begin{aligned} &\mathbf{instance}\; (\mathsf{FunctorOf}\; f\; d, \mathsf{P\_fmap2}\; f) \Rightarrow \mathsf{Functor}\; d\; \mathbf{where} \\ &\quad fmap \;=\; pmap \end{aligned}$$

Since this is an instance for a type variable (d), we need the Haskell extensions (available in `ghc` and `hugs`) for overlapping and undecidable instances.

Using the polytypic library we can also define more complex functions such as the *transpose* function that transposes two regular datatypes. For instance, converting a list of trees to a tree of lists. To define *transpose* we first define the function *listTranspose* for the special case of transposing the list type constructor with another regular type constructor. We omit the class constraints in the types for brevity.

$$\begin{aligned} &listTranspose \;::\; \_ \Rightarrow [d\; a] \to d\; [a] \\ &listTranspose\; (x : []) \;=\; pmap\; singleton\; x \\ &listTranspose\; (x : xs) = \; pzipWith\; (:)\; x\; (listTranspose\; xs) \end{aligned}$$

The function *pzipWith* (from PolyLib [16]) is the polytypic version of the Haskell prelude function *zipWith* and has type

$$pzipWith \;::\; \_ \Rightarrow (a \to b \to c) \to d\; a \to d\; b \to d\; c$$

If the structures of the arguments to *pzipWith* differ the function fails. Using *listTranspose* we can define *transpose* as follows:

$$\begin{aligned} &transpose \;::\; \_ \Rightarrow d\; (e\; a) \to e\; (d\; a) \\ &transpose\; x \;=\; pmap\; (combine\; s)\; (listTranspose\; l) \\ &\quad \mathbf{where}\; (s, l) \;=\; separate\; x \end{aligned}$$

The idea is to separate the structure and the contents of the argument to *transpose* using the function $separate \;::\; \_ \Rightarrow d\; a \to (d\; (), [a])$. The unstructured representation

```
polytypic fsum :: Num a ⇒ f a a → a
  = case f of
      g :+: h → λz → case z of
                       InL x → fsum x
                       InR y → fsum y
      g :*: h → λ(x :*: y) → fsum x + fsum y
      Unit    → const 0
      Par     → unPar
      Rec     → unRec
      d :@: g → psum ∘ pmap fsum ∘ unComp
      Const t → const 0
```

Figure 3.4: Defining *fsum* using the **polytypic** construct

is then transposed using *listTranspose* and the structure is re-applied by the function *combine* :: _ ⇒ *d* () → [*a*] → *d a*. As with *pzipWith*, *combine* fails if the length of the list does not match the number of holes in the structure. It is easy to modify *transpose* to use the Maybe monad to catch the potential failures.

The complete code for the *transpose* function can be found in Section A.17

3.3 Translating PolyP into Haskell

So far we have seen how we can use the polytypic functions defined in PolyLib directly in our Haskell program, either applying them to specific datatypes or using them to define other polytypic functions. In Section 3.3.1 below, we describe how to define polytypic functions from scratch using a slightly modified version of the PolyP language [15]. The polytypic definitions in PolyP *can* also be expressed in Haskell (as described in Section 3.2), but the syntax of the language extension is more convenient than writing the classes and the instances by hand. Sections 3.3.2 to 3.3.6 discuss how the PolyP definitions are compiled into Haskell.

3.3.1 The polytypic construct

In Section 3.2.1 we introduced the pattern functor Φ_d of a regular datatype *d a*. In PolyP we define polytypic functions by recursion over this pattern functor, using a type case construct that allows us to pattern match on pattern functors. This type case construct is translated by the compiler into a pattern functor class and instances corresponding to the branches.

In Figure 3.4 we define, using the type case construct **polytypic**, the function *fsum* from Section 3.2.3 that operates on pattern functors applied to some numeric type. This function takes an element of type *f a a* where *a* is in Num and *f* is a pattern functor. The first *a* means that the parameter positions contain numbers and the second *a* means that all the substructures have been replaced by numbers (sums of the corresponding

substructures). The result of *fsum* is the sum of the numbers in the top level structure. To sum the elements of something of a sum type we just apply *fsum* recursively regardless of if we are in the left or right summand. If we have something of a product type we sum the components and add the results together. The sum of Unit or a constant type is zero and when we get one Par or Rec they already contain a number so we just return it. If the pattern functor is a regular datatype d composed with a pattern functor g we map *fsum* over d and use the function *psum* to sum the result.

In general a **polytypic** definition has the form

$$\begin{aligned}&\textbf{polytypic}\ p\ ::\ \tau\\ &\quad =\ \lambda\, x_1\ \ldots\ x_m \rightarrow\ \textbf{case}\ f\ \textbf{of}\\ &\qquad\qquad\qquad\qquad \varphi_1 \rightarrow\ e_1\\ &\qquad\qquad\qquad\qquad\quad\ \vdots\\ &\qquad\qquad\qquad\qquad \varphi_n \rightarrow\ e_n\end{aligned}$$

where f is the pattern functor (occurring somewhere in τ) and φ_i is an arbitrary pattern matching a pattern functor. The lambda abstraction before the type case is optional and a short hand for splicing in the same abstraction in each of the branches. The type of the branch body depends on the branch pattern; more specifically we have $(\lambda\, x_1\ \ldots\ x_m \rightarrow\ e_i)\ ::\ \tau[\varphi_i/f]$.

A **polytypic** definition operates on pattern functors, but what we are really interested in are functions that operates on regular datatypes. We have already seen how to define these functions in Haskell and the only difference when defining them in PolyP is that the class constraints are simpler. Take for instance the datatype level function *psum* which can be defined as the catamorphism of *fsum*:

$$\begin{aligned}&\mathit{psum}\ ::\ (\mathsf{Regular}\ d, \mathsf{Num}\ a) \Rightarrow d\ a \rightarrow a\\ &\mathit{psum} = \mathit{cata}\ \mathit{fsum}\end{aligned}$$

The PolyP compiler translates the class constraint Regular d to a constraint FunctorOf $\Phi_d\ d$ and constraints for any suitable pattern functor classes on Φ_d.

In summary, the **polytypic** construct allows us to write polytypic functions over pattern functors by recursion over the structure of the pattern functor. We can then use these functions together with the functions *inn* and *out* to define functions that work on all regular datatypes.

3.3.2 Compilation: from PolyP to Haskell

Given a PolyP program we want to generate Haskell code that can be fed into a standard Haskell compiler. Our approach differs from the standard one in that we achieve polytypism by taking advantage of the Haskell class system, instead of specializing polytypic functions to the datatypes on which they are used. The compilation of a PolyP program consists of the three phases described in the following subsections. In the first phase, described in Section 3.3.3, the pattern functor of each regular datatype is computed and an instance of the class FunctorOf is generated, relating the datatype to its functor. The second phase (Section 3.3.4) deals with the **polytypic** definitions. For every polytypic function a type

class is generated and each branch in the type case is translated to an instance of this class. The third phase is described in Section 3.3.5 and consists of inferring the class constraints introduced by our new classes. Section 3.3.6 describes how the module interfaces are handled by the compiler.

Worth mentioning here is that we do not need to compile ordinary function definitions (i.e. functions that have not been defined using the **polytypic** keyword) even when they use polytypic functions. So for instance the definition of the function *psum* from Section 3.3.1 is the same in the generated Haskell code as in the PolyP code. The type on the other hand does change, but this is handled by phase three.

3.3.3 From datatypes to instances

When compiling a PolyP program into Haskell we have to generate an instance of the class FunctorOf for each regular datatype. How to do this is described in the rest of this section. First we observe that we can divide the pattern functor combinators into two categories: *structure* combinators that describe the datatype structure and *content* combinators that describe the contents of the datatype. The structure combinators, (:+:), (:*:) and Unit, tell you how many constructors the datatype has and their arities, while the content combinators, Par, Rec, Const, (:@:) and (:→:) represent the arguments of the constructors. For a content pattern functor g we define the *meaning* of g, denoted by $\widehat{g}$, as

$$\begin{aligned}
\widehat{\mathsf{Par}}\ p\ r &= p \\
\widehat{\mathsf{Rec}}\ p\ r &= r \\
\widehat{\mathsf{Const}\ t}\ p\ r &= t \\
\widehat{d :@: g}\ p\ r &= d\ (\widehat{g}\ p\ r) \\
\widehat{g :\!\!\rightarrow\!: h}\ p\ r &= \widehat{g}\ p\ r \rightarrow \widehat{h}\ p\ r
\end{aligned}$$

Using this notation we can write the general form of a regular datatype as

$$\begin{aligned}
\mathbf{data}\ D\ a = {} & C_1\ (\widehat{g_{11}}\ a\ (D\ a))\ \ldots\ (\widehat{g_{1m_1}}\ a\ (D\ a)) \\
& \vdots \\
\mid {} & C_n\ (\widehat{g_{n1}}\ a\ (D\ a))\ \ldots\ (\widehat{g_{nm_n}}\ a\ (D\ a))
\end{aligned}$$

The corresponding pattern functor Φ_D is

$$\Phi_D\ =\ (g_{11} :*: \cdots :*: g_{1m_1}) :+: \cdots :+: (g_{n1} :*: \cdots :*: g_{nm_n})$$

where we represent a nullary product by Unit. When defining the functions *inn* and *out* for $D\ a$ we need to convert between g_{ij} and $\widehat{g_{ij}}$. To do this we associate with each content pattern functor g two functions to_g and $from_g$ such that

$$\begin{aligned}
to_g &:: \widehat{g}\ p\ r \rightarrow g\ p\ r && \text{--- } to_g \circ from_g = id \\
from_g &:: g\ p\ r \rightarrow \widehat{g}\ p\ r && \text{--- } from_g \circ to_g = id
\end{aligned}$$

For the pattern functors Par, Rec and Const, *to* and *from* are defined simply as adding and removing the constructor. In the case of the pattern functor $g :\!\!\rightarrow\!: h$ we just have to

apply the *to* and *from* functions for g and h at the appropriate places but for $d :@: g$ we have to map the conversion function for g over the regular datatype d. This is a source of inefficiency, as mentioned in Section 3.2.1.

$$
\begin{array}{lcl|lcl}
to_{\mathsf{Par}} & = & \mathsf{Par} & from_{\mathsf{Par}} & = & unPar \\
to_{\mathsf{Rec}} & = & \mathsf{Rec} & from_{\mathsf{Rec}} & = & unRec \\
to_{\mathsf{Const}\ t} & = & \mathsf{Const} & from_{\mathsf{Const}\ t} & = & unConst
\end{array}
$$

$$
\begin{array}{lcl}
to_{d :@: g} & = & \mathsf{Comp} \circ pmap\ to_g \\
from_{d :@: g} & = & pmap\ from_g \circ unComp
\end{array}
$$

$$
\begin{array}{lcl}
to_{g :\rightarrow: h}\ f & = & \mathsf{Fun}\ (to_h \circ f \circ from_g) \\
from_{g :\rightarrow: h}\ f & = & from_h \circ unFun\ f \circ to_g
\end{array}
$$

Now define ι_m^n to be the sequence of InL and InR's corresponding to the m^{th} constructor out of n, as follows

$$
\iota_m^n\, x = \begin{cases} x & \text{if } n = m = 1 \\ \mathsf{InL}\ x & \text{if } m = 1 \wedge n > 1 \\ \mathsf{InR}\ (\iota_{m-1}^{n-1}\ x) & \text{if } m, n > 1 \end{cases}
$$

For instance the second constructor out of three is $\iota_2^3\, x\ =\ \mathsf{InR}\,(\mathsf{InL}\ x)$.

Finally an instance $\mathsf{FunctorOf}\ \Phi_D\ D$ for the general form of a regular datatype $D\ a$ can be defined as follows:

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{FunctorOf}\ \Phi_D\ D\ \textbf{where} \\
\quad inn\ (\iota_1^n\ (x_1 :*: \ldots :*: x_{m_1})) = C_1\ (from_{g_{11}}\ x_1)\ \ldots\ (from_{g_{1m_1}}\ x_{m_1}) \\
\quad \vdots \\
\quad inn\ (\iota_n^n\ (x_1 :*: \ldots :*: x_{m_n})) = C_n\ (from_{g_{n1}}\ x_1)\ \ldots\ (from_{g_{nm_n}}\ x_{m_n}) \\
\quad out\ (C_1\ x_1\ \ldots\ x_{m_1}) = \iota_1^n\ (to_{g_{11}}\ x_1 :*: \ldots :*: to_{g_{1m_1}}\ x_{m_1}) \\
\quad \vdots \\
\quad out\ (C_n\ x_1\ \ldots\ x_{m_n}) = \iota_n^n\ (to_{g_{n1}}\ x_1 :*: \ldots :*: to_{g_{nm_n}}\ x_{m_n}) \\
\quad constructorName\ (C_1\ \_\ \ldots\ \_) = \text{“}\mathtt{C_1}\text{”} \\
\quad \vdots \\
\quad constructorName\ (C_n\ \_\ \ldots\ \_) = \text{“}\mathtt{C_n}\text{”} \\
\quad datatypeName\ \_ = \text{“}\mathtt{D}\text{”}
\end{array}
$$

3.3.4 From polytypic definitions to classes

The second phase of the code generation deals with the translation of the **polytypic** construct. This translation is purely syntactic and translates each polytypic function into a pattern functor class with one method (the polytypic function) and an instance of this class for each branch in the type case. More formally, given a polytypic function definition like the left side in Figure 3.5 the translation produces the result on the right.

$$\left.\begin{array}{l}\textbf{polytypic}\ p\ ::\ \tau \\ \quad =\ \textbf{case}\ f\ \textbf{of} \\ \qquad \varphi_1 \rightarrow\ e_1 \\ \qquad\quad \vdots \\ \qquad \varphi_n \rightarrow\ e_n\end{array}\right\} \Longrightarrow \left\{\begin{array}{l}\textbf{class}\qquad\quad \mathsf{P\_p}\,f\ \ \textbf{where}\ p\ ::\ \tau \\ \textbf{instance}\,\rho_1\ \Rightarrow \mathsf{P\_p}\,\varphi_1\,\textbf{where}\ p\ =\ e_1 \\ \quad\vdots \\ \textbf{instance}\,\rho_n\ \Rightarrow \mathsf{P\_p}\,\varphi_n\textbf{where}\ p\ =\ e_n\end{array}\right.$$

Figure 3.5: Translation of a polytypic construct to a class and instances

However, the instances generated by this phase are not complete. To make them pass the Haskell type checker we have to fill in the appropriate class constraints ρ_i. For example, in the definition of *fsum* from Section 3.3.1, the instance $\mathsf{P\_fsum}\ (g :*: h)$ needs instances of $\mathsf{P\_fsum}$ for g and for h. How to infer these constraints is the topic of the next section.

3.3.5 Inferring class constraints

When we introduce a new class for every polytypic function we automatically introduce a class constraint everywhere this function is used. Ideally the Haskell compiler should be able to infer these constraints for us, allowing us to simply leave out the types in the generated Haskell code. This is indeed the case most of the time, but there are a few exceptions that require us to take a more rigorous approach. For example, class constraints must be explicitly stated in instance declarations. In other cases the Haskell compiler can infer the type of a function, but it might not be the type we want. For instance, the inferred type of the (translation of) function *pmap* is

$$pmap\ ::\ (\mathsf{FunctorOf}\ f\ d,\ \mathsf{FunctorOf}\ f\ e,\ \mathsf{P\_fmap2}\ f) \Rightarrow (a \rightarrow b) \rightarrow d\ a \rightarrow e\ b$$

which is a little too general to be practical. In the expression $psum\ (pmap\ (1+)\ [1,2,3])$, for example, the compiler would not be able to infer the return type of *pmap*. To get the type we want the inferred type is unified with the type stated in the PolyP code. When doing this we have to replace the constraint $\mathsf{Regular}\ d$ in the PolyP type, by the corresponding Haskell constraint $\mathsf{FunctorOf}\ f\ d$ for a free type variable f. Subsequently we replace all occurrences of Φ_d in the type body with f. We also add a new type constraint variable to the given type, that can be unified with the set of new constraints inferred in the type inference. In the case of *pmap* we would unify the inferred type from above with the modified version of the type stated in the PolyP code:

$$(\mathsf{FunctorOf}\ f\ d,\ \rho) \Rightarrow (a \rightarrow b) \rightarrow d\ a \rightarrow d\ b$$

Here e would be identified with d and ρ would be unified with $\{\mathsf{P\_fmap2}\ f\}$, yielding the type we want.

The instance declarations can be treated in much the same way. That is, we infer the type of the method body and unify this type with the expected type of the method. We take the definition of *fsum* in Figure 3.4 as an example. This definition is translated to a class and instance declarations for each branch:

$$\begin{array}{l}\textbf{class}\ \mathsf{P\_fsum}\ f\ \textbf{where}\\ \quad fsum\ ::\ \mathsf{Num}\ a \Rightarrow f\ a\ a \to a\\ \\ \textbf{instance}\ \rho_* \Rightarrow\ \mathsf{P\_fsum}\ (g :*: h)\ \textbf{where}\\ \quad fsum\ =\ \lambda(x :*: y) \to fsum\ x + fsum\ y\\ \vdots\end{array}$$

In the instance for the pattern functor $g :*: h$, the PolyP compiler infers the following type for *fsum*

$$(\mathsf{Num}\ a, \mathsf{P\_fsum}\ g,\ \mathsf{P\_fsum}\ h) \Rightarrow (g :*: h)\ a\ a \to a$$

This type is then unified with the type of *fsum* extended with the constraint set variable ρ_* serving as a place holder for the extra class constraints:

$$(\mathsf{Num}\ a, \rho_*) \Rightarrow f\ a\ a \to a$$

In this case the result of the unification would be

$$\begin{array}{l}f\ \ \mapsto g :*: h\\ \rho_* \mapsto\ \{\mathsf{P\_fsum}\ g, \mathsf{P\_fsum}\ h\}\end{array}$$

The part of the substitution that we are interested in is the assignment of ρ_*, i.e. the class constraints that are in the instance declaration but not in the class declaration. We obtain the following final instance of $\mathsf{P\_fsum}\ (g :*: h)$:

$$\begin{array}{l}\textbf{instance}\ (\mathsf{P\_fsum}\ g,\ \mathsf{P\_fsum}\ h) \Rightarrow \mathsf{P\_fsum}\ (g :*: h)\ \textbf{where}\\ \quad fsum\ =\ \lambda(x :*: y) \to fsum\ x + fsum\ y\end{array}$$

3.3.6 Modules: transforming the interface

The old PolyP compiler used the cut-and-paste approach to modules, treating import statements as C-style `#includes`, effectively ignoring explicit import and export lists. Since we claim that embedding polytypic programs in Haskell's class system alleviates separate compilation, we, naturally, have to do better than the cut-and-paste approach.

To be able to compile a PolyP module without knowledge of the source code of all imported modules, we generate an interface file for each module, containing the type signatures for all exported functions as well as the definitions of all exported datatypes in the module. The types of polytypic functions are given in Haskell form (that is using FunctorOf and P_name, not Regular), since we need to know the class constraints when inferring the constraints for functions in the module we are compiling.

A slightly trickier issue is the handling of explicit import and export lists in PolyP modules. Fortunately, the compilation does not change the function names, so we do not have to change which functions are imported and exported. However, we do have to import and export the generated pattern functor classes. This is done by looking at the types of the

functions in the import/export list and collecting all the pattern functor classes occurring in their constraints. So given the following PolyP module

```
module Sum (psum) where
import Base (cata)

polytypic fsum :: ...
psum = cata fsum
```

we would generate a Haskell module looking like this:

```
module Sum (psum, P_fmap2, P_fsum) where
import Base (cata, P_fmap2)

class P_fsum f where
  fsum :: ...
⟨P_fsum instances⟩
psum = cata fsum
```

The P_fmap2 in the import declaration comes from the type of *cata*, which is looked up in the interface file for the module Base, and the two exported classes come from the inferred type of *psum*. The interface files are generated by the compiler when it compiles a PolyP module. At the moment there is no automated support for generating interface files for normal Haskell modules, though this should not be difficult to add. This means that interface files for normal Haskell modules have to be written by hand.

3.4 A polytypic show function

A common example of a function that can be defined polytypically is the *show* function, that turns its argument into a string. In this section we show how to define a polytypic show function in PolyP. For simplicity we ignore infix constructors and precedences, and always generate fully parenthesized strings, so for instance, our show function will generate the string " : (1) (: (2) ([]))", instead of the more aesthetically pleasing "1 : 2 : []", when applied to the list $[1, 2]$.

We start by defining the top level function *pshow* that takes an element of a regular datatype and a function to show the elements of the datatype and produces a string.

```
pshow :: Regular d ⇒ (a → String) → d a → String
pshow showA x = constructorName x ++ fshowSum
                $ fmap2 showA (pshow showA)
                $ out x
```

The function *pshow* starts by applying *out* to the element we want to show, thus revealing its top level structure yielding an element of the type $\Phi_d\ a\ (d\ a)$. Then *fmap*2 is used with *showA* to convert the as to strings and a recursive call to *pshow* to convert the

d as to strings, resulting in something of the type Φ_d String String. Finally the function *fshowSum* is applied to the result of *fmap*2 and the name of the constructor prepended to the resulting string. The job of the function *fshowSum* is combine the strings generated from the parameters and the recursive occurrences of d into one string.

A regular datatype is normally built up as a sum of products of content pattern functors, and in many cases you find that you want to do different things for each of the three layers. In this example we do not do anything at the sum layer, in the product layer we add spaces and parentheses and the actual showing takes place in the content layer. A good technique to deal with this is to define one **polytypic** function for each layer, in our case we define the functions *fshowSum*, *fshowProd* and *fshowContent*. In this example we could actually combine *fshowSum* and *fshowProd* into a single function, but for the sake of the example we keep them separate.

```
polytypic fshowSum :: f String String → String
  = case f of
      g :+: h → λ z → case z of
                        InL x → fshowSum x
                        InR y → fshowSum y
      g       → fshowProd
```

Since the constructor name is printed already at top level the sum part of the regular data type (i.e. which constructor is used) is not interesting, so if f is a sum type we just apply *fshowSum* recursively until we get something that is not a sum type. In that case we call *fshowProd* to print the argument list.

As an aside, note that this definition requires that overlapping instances are allowed by the Haskell compiler, since we have overlapping patterns in the type case ($g :+: h$ and g). The generated code is accepted by current GHC and hugs (with suitable extensions turned on), but if more portable Haskell code is wanted the definition could be changed to explicitly match on all the remaining pattern functor cases.

The function *fshowProd* is defined in a similar style.

```
polytypic fshowProd :: f String String → String
  = case f of
      g :*: h     → λ(x :*: y) → fshowProd x ++ fshowProd y
                  → Unit
      const "" g  → λ x → " (" ++ fshowContent x ++ ")"
```

Function *fshowProd* is used to print the *spine* of the argument list that follows after the constructor printed at top level (by *pshow*). The actual arguments in the list are filled in by *fshowContent*. Finally we define the function *fshowContent* as

```
polytypic fshowContent :: f String String → String
  = case f of
                      → Par
      unPar           → Rec
      unRec d :@: g   → pshow fshowContent ∘ unComp
      Const t         → show ∘ unConst
```

If the argument is a parameter (Par) or a recursive call (Rec) we simply return the (already generated) string and if it is a constant type we apply the standard *show* function from the Haskell prelude. Note that this means that we can only apply the polytypic show function to datatypes where all the constant types are in the class Show. When the argument type is a composition, d :@: g, we call *pshow* at d using *fshowContent* to show the parameters (of type g String String).

The functions defined in this section can be compiled by the PolyP compiler into a Haskell module that can be understood by a normal Haskell compiler (that allows multi-parameter type classes and overlapping instances). If we also allow undecidable instances we can write the following neat instance in our Haskell program

```
instance (FunctorOf f d, P_fmap2 f, P_fshowSum f, Show a) =>
         Show (d a) where
  show = pshow show
```

This instance enables us to use the standard *show* function for any regular datatype for which *pshow* is defined. This another example of the problem mentioned in Section 3.2.3, namely that the class constraints quickly get out of hand when using polytypic functions. In this case it is a bit more serious though, since the Haskell compiler will not infer class constraints in instance declarations for us. A solution to this inconvenience could be to extend the compiler to handle instance declarations, allowing you to write

```
instance (Regular d, Show a) => Show (d a) where
  show = pshow show
```

in the PolyP code. The most labor intensive part of such an extension would be to make the compiler understand and type check instance declarations — the constraint inference is already implemented.

3.5 A polytypic term interface

As our second case study we define polytypic functions to handle terms with binding constructs, illustrating the use of polytypic functions in Haskell programs. Our term interface builds on work by Jansson and Jeuring [17], but adds support for dealing with bound variables generically.

We define a term to be a regular datatype containing variables, and require for simplicity that the first constructor contains the variable, i.e. that the first constructor has the form Const Var for some predefined variable type Var. We can then define a polytypic function that checks if a term is a variable.

```
polytypic fvarCheck :: f p r → Maybe Var
  = case f of
      Const Var :+: h → λz → case z of
                               InL (Const x) → Just x
                               InR _         → Nothing
```

Here we have a single branch in the type case that only matches datatypes whose first constructor only contains a variable. This effectively restricts the domain of *fvarCheck* to a particular subset of the regular datatypes. If the argument to *fvarCheck* is built up using the first constructor we simply return the variable otherwise we return Nothing. Note that *fvarCheck* operates on the pattern functor level, so we have to define another function, *pvarCheck* operating on the datatype level. We will save that one for later, though.

The function *fvarCheck* acts as an elimination function for variables, taking a term and returning a Var if the term is a single variable. We also need to be able to construct variables. To do this we define the function *fVar* as

```
polytypic fVar :: Var → f p r
  = case f of
      Const Var :+: h → InL ∘ Const
```

Again we only have a single branch in the type case stating that we can only construct an element of datatypes whose first constructor only contains a variable.

The polytypic functions defined above are enough to handle the variables occurring in a term, but we also need functions to handle variable bindings. We define that a constructor is binding if has more than one argument and the first argument is a Var, in other words if it has the structure Const Var :*: h for some pattern functor h. First let us define a function *fbindCheck* that returns the bound variable if applied to a binding construct.

```
polytypic fbindCheck :: f p r → Maybe Var
  = case f of
      g :+: h          → λ z → case z of
                                 InL x → fbindCheck x
                                 InR y → fbindCheck y
      Const Var :*: h  → λ (Const x :*: _) → Just x
      g                → const Nothing
```

Any constructor can be binding so we simply lift *fbindCheck* through the sum part of the datatype. If the argument matches the structure of a binding construct we return the variable, otherwise we return Nothing.

Since a term can have more than one binding construct (e.g. lambda abstraction and let bindings) we cannot define a constructor function for bindings as we did for variables. Something we want to be able to do, though, is to rename the bound variable in a binding, so let us define a function *frenameBound* to do this.

```
polytypic frenameBound :: Var → f p r → f p r
  = λ v →
    case f of
      g :+: h          → λ z → case z of
                                 InL x → InL (frenameBound v x)
                                 InR y → InR (frenameBound v y)
      Const Var :*: h  → λ (_ :*: y) → Const v :*: y
      g                → id
```

This function follows the same pattern as *fbindCheck* but now, when we find a binding construct we replace the bound variable with the provided variable and leave the rest of the term intact. Note that this is not a semantic preserving operation on terms—*frenameBound* only renames the variable in the actual binding, it does not rename the occurrences of the bound variable in the term.

These four functions can be compiled by the PolyP compiler into four Haskell classes and corresponding instances. The rest of this case study is ordinary Haskell code that requires no preprocessing.

As we mentioned above, the polytypic functions we have defined work at the pattern functor level but what we really want is functions the operates on regular datatypes. So let us define the datatype level functions corresponding to the polytypic functions above.

```
pvarCheck :: (FunctorOf f d, P_fVarCheck f) ⇒ d a → Maybe Var
pvarCheck t = fvarCheck (out t)

pVar :: (FunctorOf f d, P_fVar f) ⇒ Var → d a
pVar x = inn (fVar x)

pbindCheck :: (FunctorOf f d, P_fbindCheck f) ⇒ d a → Maybe Var
pbindCheck t = fbindCheck (out t)

prenameBound :: (FunctorOf f d, P_frenameBound f) ⇒
                Var → d a → d a
prenameBound x = inn ∘ frenameBound x ∘ out
```

We write down the type of these functions explicitly here, but in the following we will elide the class constraints and use the same notation as in Section 3.2.6.

Apart from these functions for manipulating variables and variable bindings we need functions that operates on the top level children of a term.

```
pchildren :: _ ⇒ d a → [d a]
pchildren t = fl_rec (out t)

pmapC :: _ ⇒ (d a → d a) → d a → d a
pmapC f = inn ∘ fmap2 id f ∘ out
```

The function *pchildren* takes an element of a regular datatype and returns a list of its immediate children, so for instance, *pchildren* $[1,2,3] = [[2,3]]$. The function *fl_rec* is a polytypic library function of type $f\ p\ r \to [r]$. The function *pmapC* applies a function to all immediate children of an element of a datatype.

We can combine the functions *pvarCheck* and *pBindCheck* into a single function as follows.

```
data VarOrBind = IsVar Var | IsBind Var | Neither
```

```
pvarOrBind :: _ ⇒ d a → VarOrBind
pvarOrBind t =
  case pvarCheck t of
    Just x → IsVar x
    _      → case pbindCheck t of
               Just x → IsBind x
               _      → Neither
```

Now let us define a function for computing the free variables of a term:

```
pfreeVars :: _ ⇒ d a → [Var]
pfreeVars t = case pvarOrBind t of
                IsVar x  → [x]
                IsBind x → delete x vs
                Neither  → vs
  where vs = foldr union [] $ map pfreeVars $ pchildren t
```

If the term is a variable we just return the variable otherwise we compute the list of free variables in the children of the term by applying *pfreeVars* recursively to all immediate children and then combining the results. If the term is binding we remove the bound variable from the computed list. Using this function we can define a function for checking if a variable is free in a term.

```
isFreeIn :: _ ⇒ Var → d a → Bool
isFreeIn x t = elem x (pfreeVars t)
```

Another interesting function that we can define is term substitution. The trickiest part of a substitution function is to avoid inadvertent variable capture. To solve this problem we need to be able to generate fresh variable names, so assume we have a function *freshVar*

```
freshVar :: [Var] → Var
```

that takes a list of variables and returns a variable that is not in the list. We can now define substitution as follows

```
subst :: _ ⇒ (Var, d a) → d a → d a
subst (y, u) t =
    case pvarOrBind t of
      IsVar x  | x == y          → u
               | otherwise       → t
      IsBind x | x == y          → t
               | isFreeIn x u    → pmapC (subst (y, u)) (renameB x (freshVar [u, t]) t)
                                 → pmapC (subst (y, u)) t
  where
    renameB x z t = pmapC (subst (x, pVar z)) (prenameBound z t)
```

The substitution function *subst* takes a variable y and two terms u and t and substitutes u for y in t. This is done by analyzing the structure of t. The first three cases are easy,

the most difficult case is when t binds a variable that occurs free in u, in which case we have to alpha rename the bound variable avoid capturing free occurrences in u. If t does not bind a variable occurring free in u we can just apply the substitution recursively to the children of t.

Now we can define a concrete datatype for terms and apply our functions to it. Take for instance lambda terms with constants and let bindings

```
data Term a =Var Var
            | Constant a
            | Term a :@ Term a
            | Lam Var (Term a)
            | Let Var (Term a) (Term a)
```

A thing that is worth mentioning is that by our definition of variable binding the let construct is automatically recursive, in other words in the term Let x u t (representing the let expression **let** $x = u$ **in** t), x is bound in u as well as in t. One could of course imagine other definitions of variable bindings that would allow both recursive and non recursive let constructs, but that would complicate things unnecessarily.

For us to be able to apply our polytypic term functions to lambda terms we have to have an instance FunctorOf f Term for some pattern functor f. This instance can be generated by the PolyP compiler, or by the Template Haskell function *deriveFunctorOf* from PolyLib (see Section A.2 in the Appendix). Having the FunctorOf instance we can apply the polytypic substitution function to our terms. To improve the readability of the examples we omit the Var constructor in the terms and write x instead of Var x. Assume that x, y and z are variables, then

$$\begin{aligned} &\mathit{subst}\ (x,\ y\ :@\ z)\ (\mathsf{Lam}\ y\ (y\ :@\ x)) &&\Longrightarrow \mathsf{Lam}\ w\ (w\ :@\ (y\ :@\ z)) \\ &\mathit{subst}\ (x,\ y)\ (\mathsf{Let}\ z\ (\mathsf{Lam}\ y\ x)\ (z\ :@\ x)) &&\Longrightarrow \mathsf{Let}\ z\ (\mathsf{Lam}\ w\ y)\ (z\ :@\ y) \end{aligned}$$

for some variable w.

This example has shown that a lot of the polytypic programming can be done inside Haskell. The only parts that had to be run through the PolyP compiler were the four **polytypic** definitions in the beginning.

3.6 Related work

A number of languages and tools for polytypic programming with Haskell have been described in the last few years:

- The old PolyP [15] allows user-defined polytypic definitions over regular datatypes. The language for defining polytypic functions is more or less the same as in our work, however, the expressiveness of old PolyP is hampered by the fact that the specialization needs access to the entire program. Neither the old nor the new PolyP compiler supports full Haskell 98, something that severely limits the usefulness of the old version, while in the new version it is merely a minor inconvenience.

- Generic Haskell [4, 10] allows polytypic definitions over Haskell datatypes of arbitrary kinds. The Generic Haskell compiler uses specialization to compile polytypic programs into Haskell, which means that it suffers from the drawbacks mentioned above, namely that we have to apply the compiler to any code that mentions polytypic functions or contains datatype definitions. This is not as serious in Generic Haskell as it is in old PolyP however, since Generic Haskell supports full Haskell 98 and has reasonably good support for separate compilation. A more significant shortcoming of Generic Haskell is that it does not allow direct access to the substructures in a datatype, so we cannot define, for instance, the function $children :: t \to [t]$ that takes an element of a datatype and returns the list of its immediate children.

  Generic Haskell requires definitions of polytypic functions to be over arbitrary kinds, even if a function is only intended for a single kind. This sometimes makes it rather difficult to come up with the right definition for a polytypic function.

- Derivable type classes [11] is an extension of the Glasgow Haskell Compiler (ghc) which allows limited polytypic definitions. The user can define polytypic default methods for a class by giving cases for sums, products and the singleton type. To make a datatype an instance of a class with polytypic default methods it suffices to give an empty instance declaration. Nevertheless this requires the user to write an empty instance declaration for each polytypic function-datatype pair while we only require a FunctorOf-instance for each datatype. Furthermore the derivable type classes extension only allows a limited form of polytypic functions over kind $\star$, as opposed to kind $\star \to \star$ in PolyP. Only allowing polytypic functions over datatypes of kind $\star$ excludes many interesting functions, such as *pmap*, and since a datatype of kind $\star$ can always be transformed into a datatype of kind $\star \to \star$ (by adding a dummy argument) we argue that our approach is preferable. A similar extension to derivable type classes, exists also for Clean [1].

- The DrIFT preprocessor for deriving non-standard Haskell classes has been used together with the Strafunski library [23] to provide generic programming in Haskell. The library defines combinators for defining generic traversal and generic queries on datatypes of kind $\star$. A generic traversal is a function of type $t \to m\ t$ for some monad m and a generic query on t has type $t \to a$. The library does not support functions of any other form, such as unfolds or polytypic equality.

  The Strafunski implementation relies on a universal term representation, and generic functions are expressed as normal Haskell functions over this representation. This means that only the Regular sublanguage has to be compiled (suitable instances to convert to and from the term representation have to be generated). This is done by the DrIFT preprocessor.

- Recently Lämmel and Peyton-Jones [21] have incorporated a version of Strafunski in ghc providing compiler support for defining generic functions. This implementation has the advantage that the appropriate instances can be derived by the compiler, only requiring the user to write a deriving-clause for each of her datatypes. Support has been added for unfolds and so called twin transformations (of type $t \to t \to m\ t$)

which enables for instance, polytypic read, equality and zip functions. Still, only datatypes of kind $\star$ is handled, so we cannot get access to the parameters of a datatype.

- Sheard [31] describes how to use two-level types to implement efficient generic unification. His ideas, to separate the structure of a datatype (the pattern functor) from the actual recursion, are quite similar to those used in PolyP, although he lacks the automated support provided by the PolyP compiler. In fact, the functions that Sheard requires over the structure of a datatype can all be defined in PolyP.

- In a recent paper [8] Hinze shows how to use Haskell 98 type classes to encode generic programming. Instead of having one type class for each generic function, as we have, Hinze uses a single class Generic for all generic functions. In his approach a generic function is defined by giving an instance of this class. He shows how generic functions over arbitrary datatypes of kind $\star$ or kind $\star \to \star$ can be defined using this technique.

- When we use type classes to implement generic programming, we are basically moving computation from the terms to the types. The pattern matching on a pattern functor in PolyP is, in the Haskell world, done by the type system looking for the the appropriate instance for the pattern functor. This technique is not limited to generic programming, for instance, McBride [25] shows how to encode dependently typed programming using the class system.

Other implementations of functional generic programming include Charity [5], FISh [19] and G'Caml [6] but in this paper we focus on the Haskell-based languages.

3.7 Conclusions

In this paper we have shown how to allow polytypic programming inside Haskell, by taking advantage of the class system. To accomplish this we introduced datatype constructors for modeling the top level structure of a datatype, together with a multi-parameter type class FunctorOf relating datatypes to their top level structure.

Using this framework we have been able to rephrase the PolyLib library [16] as a Haskell library as well as define new polytypic functions such as *coerce* that converts between two datatypes of the same shape, a substitution function on terms with bindings and the *transpose* function that commutes a composition of two datatypes, converting, for instance, a list of trees to a tree of lists.

To aid in the definition of polytypic functions we have a compiler that translates PolyP definitions to Haskell classes and instances. The same compiler can generate instances of FunctorOf for regular datatypes, but the framework also allows the programmer to provide tailor made FunctorOf instances, thus extending the applicability of the polytypic functions to datatypes that are not necessarily regular.

Areas of possible future work include extending our approach to more datatypes (partially explored in [27]), and to explore in more detail which polytypic functions are expressible in this setting. It would also be interesting to measure the efficiency of the polytypic functions in this approach compared to the specialized code of previous methods.

Appendix A

PolyLib

Using the PolyP 2 compiler described in Chapter 3, the polytypic library [16] can be compiled into a Haskell library that provides polytypic programming inside Haskell. In this appendix we describe this library, listing the polytypic functions and their types.

A.1 PolyLib.Prelude

The Prelude module contains the basic machinery needed to do polytypic programming in Haskell: the FunctorOf class and the definition of the pattern functors.

```
module PolyLib.Prelude where

class FunctorOf f d | f → d where
  inn                :: f a (d a) → d a
  out                :: d a → f a (d a)
  dataTypeName       :: d a → String
  constructorName    :: d a → String
  constructorFixity  :: d a → Fixity
    — Defaults
  constructorFixity _ :: Fixity LeftAssoc 9
data Fixity = Fixity {associativity :: Associativity
                     , precedence   :: Int
                     }
data Associativity = NonAssoc | LeftAssoc | RightAssoc

data      (f :+: g) p r = InL (f p r) | InR (g p r)
data      (f :*: g) p r = f p r :*: g p r
data      Unit      p r = Unit
newtype   Par       p r = Par { unPar  :: p }
newtype   Rec       p r = Rec { unRec  :: r }
newtype   (d :@: g) p r = Comp { unComp :: d (g p r) }
newtype   Const t   p r = Const { unConst :: t }
newtype   (f :→: g) p r = Fun { unFun :: f p r → g p r }
```

This precisely what was presented in Section 3.2.1 with the addition of the fixity information. Furthermore, the Prelude contains the *to* and *from* functions described in Section 3.3.3 and an implementation of a polytypic map function (recall that the *to* and *from* functions for (:@:) do mapping over regular datatypes).

```
class Bifunctor f where
  bimap :: (a → c) → (b → d) → f a b → f c d

gmap :: (FunctorOf f d, Bifunctor f) ⇒ (a → b) → d a → d b

class PatternFunctor f p r t | f p r → t where
  to   :: t → f p r
  from :: f p r → t
instance PatternFunctor Par p r p where ...
instance PatternFunctor Rec p r r where ...
instance (FunctorOf f d, Bifunctor f, PatternFunctor g p r t)
         ⇒ PatternFunctor (d :@: g) p r (d t) where ...
instance (PatternFunctor f p r t, PatternFunctor g p r u)
         ⇒ PatternFunctor (f :→: g) p r (t → u) where ...
```

To make them easier to use the *to* and *from* functions are wrapped up in a class PatternFunctor. The constraint PatternFunctor $f\ p\ r\ t$ states that f is a pattern functor with $f\ p\ r \cong t$.

A.2 PolyLib.FunctorOf

The FunctorOf module contains a single Template Haskell function, *deriveFunctorOf*, that generates FunctorOf instances:

```
module PolyLib.FunctorOf (deriveFunctorOf) where

deriveFunctorOf :: Q Dec → Q [Dec]
```

This function takes the abstract syntax for the definition of a regular datatype and produces an instance of FunctorOf for this datatype. So to use it in a Haskell program one would write

```
data T a = ...

$(deriveFunctorOf (reifyDecl T))
```

The techniques used to define this function are the same as the ones used when implementing the PolyP prototype from Section 2.4.

A.3 PolyLib.Base

The Base module contains the polytypic map function *pmap* and various morphisms.

```
module PolyLib.Base where

import PolyLib.Prelude

class P_ fmap2 f where
  fmap2 :: (a → c) → (b → d) → f a b → f c d
```

The P_ fmap2 class is the extension of the Functor class from the standard Prelude. All pattern functors except (:→:) are instances of this class.

```
pmap :: (FunctorOf f d, P_ fmap2 f) ⇒ (a → b) → d a → d b
cata :: (FunctorOf f d, P_ fmap2 f) ⇒ (f a b → b) → d a → b
ana  :: (FunctorOf f d, P_ fmap2 f) ⇒ (b → f a b) → b → d a
hylo :: P_ fmap2 f ⇒ (f a b → b) → (c → f a c) → c → b
para :: (FunctorOf f d, P_ fmap2 f) ⇒ (d a → f a b → b) → d a → b
```

The functions *cata*, *ana*, *hylo* and *para* implement the recursion schemes popularized by Meijer et al. [26].

A.4 PolyLib.BaseM

The BaseM module contains the monadic versions of the functions from the Base module. Every function f comes in two versions: fM, which threads the monad through left to right, and fMr, which threads the monad right to left.

```
module PolyLib.BaseM where

import PolyLib.Prelude
import PolyLib.Base

class P_ fmap2M f where
  fmap2M :: Monad m ⇒
            (a → m c) → (b → m d) → f a b → m (f c d)
class P_ fmap2Mr f where
  fmap2Mr :: Monad m ⇒
             (a → m c) → (b → m d) → f a b → m (f c d)
```

All pattern functors except (:→:) are instances of P_ fmap2M and P_ fmap2Mr. This is true for following pattern functor classes as well, unless we state otherwise.

```
pmapM  :: (FunctorOf f d, P_fmap2M f, Monad m) ⇒
          (a → m b) → d a → m (d b)
pmapMr :: (FunctorOf f d, P_fmap2Mr f, Monad m) ⇒
          (a → m b) → d a → m (d b)
cataM  :: (FunctorOf f d, P_fmap2M f, Monad m) ⇒
          (f a b → m b) → d a → m b
cataMr :: (FunctorOf f d, P_fmap2Mr f, Monad m) ⇒
          (f a b → m b) → d a → m b
anaM   :: (FunctorOf f d, P_fmap2M f, Monad m) ⇒
          (b → m (f a b)) → b → m (d a)
anaMr  :: (FunctorOf f d, P_fmap2Mr f, Monad m) ⇒
          (b → m (f a b)) → b → m (d a)
hyloM  :: (P_fmap2M f, Monad m) ⇒
          (f a b → m b) → (c → m (f a c)) → c → m b
hyloMr :: (P_fmap2Mr f, Monad m) ⇒
          (f a b → m b) → (c → m (f a c)) → c → m b
paraM  :: (FunctorOf f d, P_fmap2M f, Monad m) ⇒
          (d a → f a b → m b) → d a → m b
paraMr :: (FunctorOf f d, P_fmap2Mr f, Monad m) ⇒
          (d a → f a b → m b) → d a → m b
```

A.5 PolyLib.Crush

A common pattern of computation is to collapse or *crush* a data structure by combining all its elements using a binary operation. This pattern is captured by the *crush* function.

```
module PolyLib.Crush where
import PolyLib.Prelude
import PolyLib.Base

class P_fcrush f where
  fcrush :: (a → a → a) → a → f a a → a

crush :: (FunctorOf f d, P_fmap2 f, P_fcrush f) ⇒
         (a → a → a) → a → d a → a
```

A.6 PolyLib.CrushFuns

Many well-known functions can be expressed as a *crush*. The CrushFuns module contains a few of them.

```
module PolyLib.CrushFuns where

import PolyLib.Prelude
```

```
import PolyLib.Base
import PolyLib.Crush

psum :: (FunctorOf f d, P_fmap2 f, P_fcrush f, Num a) => d a -> a
prod :: (FunctorOf f d, P_fmap2 f, P_fcrush f, Num a) => d a -> a
pand :: (FunctorOf f d, P_fmap2 f, P_fcrush f) => d Bool -> Bool
por  :: (FunctorOf f d, P_fmap2 f, P_fcrush f) => d Bool -> Bool
conc :: (FunctorOf f d, P_fmap2 f, P_fcrush f) => d [a] -> [a]
comp :: (FunctorOf f d, P_fmap2 f, P_fcrush f) =>
        d (a -> a) -> (a -> a)
```

The first four functions generalizes the Prelude functions *sum*, *product*, *and* and *or*. The *conc* function concatenates all lists in a structure and *comp* takes a structure of functions and returns their composition. Using *crush* to do concatenation can be slightly inefficient, since there is no guarantee that *crush* uses its function argument ((++) in this case) right associatively.

```
size    :: (FunctorOf f d, P_fmap2 f, P_fcrush f) => d a -> Int
flatten :: (FunctorOf f d, P_fmap2 f, P_fcrush f) => d a -> [a]
flatten' :: (FunctorOf f d, P_fmap2 f, P_fcrush f) => d a -> [a] -> [a]
pall    :: (FunctorOf f d, P_fmap2 f, P_fcrush f) =>
           (a -> Bool) -> d a -> Bool
pany    :: (FunctorOf f d, P_fmap2 f, P_fcrush f) =>
           (a -> Bool) -> d a -> Bool
pelem   :: (FunctorOf f d, P_fmap2 f, P_fcrush f, Eq a) =>
           a -> d a -> Bool
```

The function *size* counts the number of elements in a structure and the *flatten'* function is a more efficient version of *flatten* that returns a list of all the elements in a structure. The functions *pall*, *pany* and *pelem* are generalizations of the Prelude functions *all*, *any* and *elem*.

A.7 PolyLib.Flatten

In Section A.6 we saw that *flatten* can be defined using *crush*. However, by defining a new pattern functor class we get a bit more functionality.

```
module PolyLib.Flatten where

import PolyLib.Prelude
import PolyLib.Base

class P_fflatten f where
  fflatten :: f [a] [a] -> [a]

flatten :: (FunctorOf f d, P_fmap2 f, P_fflatten f) => d a -> [a]
```

This version of the *flatten* function does the same thing as the version using *crush* from Section A.6, in particular it has the same problem with left associative use of (++).

$$\begin{aligned}
&\mathit{fl_par} :: (\mathsf{P_fmap2}\, f,\ \mathsf{P_fflatten}\, f) \Rightarrow f\ a\ b \to [a] \\
&\mathit{fl_rec} :: (\mathsf{P_fmap2}\, f,\ \mathsf{P_fflatten}\, f) \Rightarrow f\ a\ b \to [b] \\
&\mathit{fl_all} :: (\mathsf{P_fmap2}\, f,\ \mathsf{P_fflatten}\, f) \Rightarrow f\ a\ a \to [a]
\end{aligned}$$

With a special pattern functor class for flattening we can do more than just returning the elements in a structure. The above functions operates on pattern functors and returns a list of all parameters, recursive calls, or both.

$$\begin{aligned}
&\mathit{substructures} :: (\mathsf{FunctorOf}\, f\ d,\ \mathsf{P_fmap2}\, f,\ \mathsf{P_fflatten}\, f) \Rightarrow \\
&\qquad\qquad d\ a \to [d\ a]
\end{aligned}$$

Using *fl_rec* we can define a function that computes all substructures of a structure. For instance, applied to a list it would return all suffixes of the list.

A.8 PolyLib.Fold

The module Fold contains generalizations of the list functions *foldr* and *foldl* from the standard Prelude.

module PolyLib.Fold **where**

import PolyLib.Prelude
import PolyLib.Base

class P_ffoldr f **where**
  $\mathit{ffoldr} :: (a \to b \to b) \to f\ a\ (b \to b) \to b \to b$
class P_ffoldl f **where**
  $\mathit{ffoldl} :: (a \to b \to b) \to f\ a\ (b \to b) \to b \to b$

$$\begin{aligned}
&\mathit{pfoldr} :: (\mathsf{FunctorOf}\, f\ d,\ \mathsf{P_fmap2}\, f,\ \mathsf{P_ffoldr}\, f) \Rightarrow \\
&\qquad (a \to b \to b) \to b \to d\ a \to b \\
&\mathit{pfoldl} :: (\mathsf{FunctorOf}\, f\ d,\ \mathsf{P_fmap2}\, f,\ \mathsf{P_ffoldl}\, f) \Rightarrow \\
&\qquad (b \to a \to b) \to b \to d\ a \to b
\end{aligned}$$

A.9 PolyLib.ConstructorName

The FunctorOf class contains a method that returns the name of the top-level constructor of its argument. The ConstructorName module elaborates on this and provides functions for computing the names and arities of datatype constructors.

module PolyLib.ConstructorName **where**

```
import PolyLib.Prelude

class P_ fconstructors f where
  fconstructors :: [f a b]
class P_ fconstructorArity f where
  fconstructorArity :: f a b → Int
class P_ fconstructor2Int f where
  fconstructor2Int :: f a b → Int
```

The above pattern functor classes have instances for all pattern functors, including (:→:).

```
constructors          :: (FunctorOf f d, P_ fconstructors f) ⇒ [d a]
constructorNames      :: (FunctorOf f d, P_ fconstructors f) ⇒
                         d a → [String]
constructorArity      :: (FunctorOf f d, P_ fconstructorArity f) ⇒
                         d a → Int
constructorsAndArities
    :: (FunctorOf f d, P_ fconstructors f, P_ fconstructorArity f) ⇒
       [d a]
constructorNamesAndArities
    :: (FunctorOf f d, P_ fconstructors f, P_ fconstructorArity f) ⇒
       d a → [(d a, Int)]
constructor2Int :: (FunctorOf f d, P_ fconstructor2Int) ⇒ d a → Int
int2constructor :: (FunctorOf f d, P_ fconstructors ⇒ Int → d a
```

The list produced by *constructors* contains elements of the corresponding datatype built from each of the constructors. For instance, for the Maybe type *constructors* returns [Nothing, Just *undefined*] and for lists it returns [[] , *undefined* : *undefined*].

A.10 PolyLib.Show

In Section 3.4 we saw how to implement a polytypic pretty printer using three pattern functor classes P_ fshowSum, P_ fshowProd and P_ fshowContent. The pretty printer defined in PolyLib.Show combines the three into a single pattern functor class P_ fshowsPrec.

```
module PolyLib.Show where

import PolyLib.Prelude
import PolyLib.Base
import PolyLib.ConstructorName

class P_ fshowsPrec f where
  fshowsPrec :: Int → f (Int → ShowS) (Int → ShowS) → ShowS
```

There are P_ fshowsPrec instances for all pattern functors except (:→:), but the instance for Const t requires that t is Showable.

```
pshowsPrec' :: ( FunctorOf f d,
                 P_ fmap2 f,
                 P_ fconstructorArity f,
                 P_ fshowsPrec f
               ) ⇒ (Int → a → ShowS) → Int → d a → ShowS
pshowsPrec  :: ( FunctorOf f d,
                 P_ fmap2 f,
                 P_ fconstructorArity f,
                 P_ fshowsPrec f,
                 Show a
               ) ⇒ Int → d a → ShowS
pshow       :: ( FunctorOf f d,
                 P_ fmap2 f,
                 P_ fconstructorArity f,
                 P_ fshowsPrec f,
                 Show a
               ) ⇒ d a → String
```

The pretty printer defined in Section 3.4 did not handle parenthesis in any clever way, adding parenthesis around everything. This pretty printer does a little better. The Int argument to *pshowsPrec* is the precedence level of the surrounding context and parenthesis will only be added if this is high enough. This pretty printer does not handle infix constructors, so pretty printing the list $[1, 2]$ yields "(:) 1 ((:) 2 [])".

A.11 PolyLib.Thread

The Prelude function *sequence* :: Monad $m \Rightarrow [m\ a] \to m\ [a]$ threads a monad through a list. We define the generalization of this function to an arbitrary regular datatype in the function *thread*.

```
module PolyLib.Thread where
import PolyLib.Prelude
import PolyLib.Base

class P_ fthread f where
  fthread :: Monad m ⇒ f (m a) (m b) → m (f a b)

thread :: (FunctorOf f d, P_ fmap2 f, P_ fthread f, Monad m) ⇒
          d (m a) → m (d a)
```

As with *sequence*, the monad is threaded left-to-right, so *thread* [*print* 1, *print* 2] will output a 1 followed by a 2.

A.12 PolyLib.ThreadFuns

Specializing the *thread* function from Section A.11 to the list monad and the maybe monad we obtain the functions *cross* and *propagate*.

```
module PolyLib.ThreadFuns where

import PolyLib.Prelude
import PolyLib.Thread

cross     :: (FunctorOf f d, P_ fmap2 f, P_ fthread f) ⇒
             d [a] → [d a]
propagate :: (FunctorOf f d, P_ fmap2 f, P_ fthread f) ⇒
             d (Maybe a) → Maybe (d a)
```

The *cross* function takes a d-structure of lists and produces the list of all d-structures we can acquire by just picking one element from each list. For example:

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

cross (Branch (Leaf [1, 2]) (Leaf [3, 4])) == [Branch (Leaf 1) (Leaf 3),
                                               Branch (Leaf 1) (Leaf 4),
                                               Branch (Leaf 2) (Leaf 3),
                                               Branch (Leaf 2) (Leaf 4)
                                              ]
```

The *propagate* function returns Nothing unless all elements in the d-structure are Justs, in which case it returns Just the d-structure with the maybe type stripped away.

A.13 PolyLib.Zip

The Zip module defines generalizations of the Prelude functions *unzip*, *zip* and *zipWith* that works on pairs of lists. When zipping together two lists a natural thing to do if the lists have different lengths is to truncate the longer of the two—this is indeed what the Prelude zip functions do—but for an arbitrary regular datatype truncation is not well defined. For this reason the polytypic zip functions produce results in the maybe monad, if the zipped structures do not match Nothing is returned.

```
module PolyLib.Zip where

import PolyLib.Prelude
import PolyLib.Base
import PolyLib.ThreadFuns

class P_ fzip f where
  fzip :: (f a b, f c d) → Maybe (f (a, c) (b, d))
```

When trying to zip two structures containing constant types (the Const t pattern functor), the ts are required to be the same. Consequently there is an Eq t constraint on the Const t instance of P_fzip.

```
punzip    :: (FunctorOf f d, P_fmap2 f) ⇒ d (a, b) → (d a, d b)
pzip      :: (FunctorOf f d,
              P_fmap2 f,
              P_fthread f,
              P_fzip f
             ) ⇒ (d a, d b) → Maybe (d (a, b))
pzipWith  :: (FunctorOf f d,
              P_fmap2 f,
              P_fthread f,
              P_fzip f
             ) ⇒ ((a, b) → Maybe c) → (d a, d b) → Maybe (d c)
```

Recall the type of $zipWith :: (a \to b \to c) \to [a] \to [b] \to [c]$. Apart from currying a difference between this function and the polytypic *pzipWith* is that the function argument to *pzipWith* is allowed to fail, adding some extra generality to the polytypic function.

A.14 PolyLib.Equal

It is possible to implement polytypic equality using *pzipWith* from Section A.13 and *pand* from Section A.6. In this module, however we choose an independent implementation.

```
module PolyLib.Equal where

import PolyLib.Prelude

class P_fequal f where
  fequal :: (a → c → Bool) → (b → d → Bool) →
            f a b → f c d → Bool
```

As for the polytypic zip functions in the previous section we need equality over all constant types.

```
pequal :: (FunctorOf f d, P_fequal f) ⇒
          (a → b → Bool) → d a → d b → Bool
peq    :: (FunctorOf f d, P_fequal f, Eq a) ⇒ d a → d a → Bool
```

A.15 PolyLib.Compare

The implementation of a polytypic *compare* function works along the same lines as the polytypic equality from Section A.14.

```
module PolyLib.Compare where
```

```
import PolyLib.Prelude

class P_ fcompare f where
  fcompare :: (a → b → Ordering) → (c → d → Ordering) →
              f a c → f b d → Ordering
```

Here we require constant types to be instances of the Ord class, to be able to compare them.

```
pcompare  :: (FunctorOf f d, P_ fcompare f) ⇒
             (a → b → Ordering) → d a → d b → Ordering
pcompare' :: (FunctorOf f d, P_ fcompare f, Ord a) ⇒
             d a → d a → Ordering
```

A.16 PolyLib.EP

So far, most polytypic functions have been defined only on datatypes with no function spaces. In the EP module we define polytypic embedding-projection pairs that allows mapping over datatypes with function spaces.

```
module PolyLib.EP where

import PolyLib.Prelude

data EP a b = EP (a → b) (b → a)
```

An embedding-projection pair is basically a pair of functions $\iota :: a \to b$ and $\pi :: b \to a$. Formally, for (ι, π) to be a true embedding-projection pair they should satisfy $\pi \circ \iota = id$ and $\iota \circ \pi \sqsubseteq id$, but for our purposes we can ignore these requirements.

```
class P_ fEP f where
  fEP :: EP a c → EP b d → EP (f a b) (f c d)

pEP :: (FunctorOf f d, P_ fEP f) ⇒ EP a b → EP (d a) (d b)
```

A.17 PolyLib.Transpose

In Section 3.2.6 we described how to implement the function *transpose* in Haskell using polytypic functions from PolyLib. We now give the complete definition.

```
module PolyLib.Transpose where
```

```
import PolyLib.Prelude        (FunctorOf)
import PolyLib.Base           (P_fmap2, pmap)
import PolyLib.BaseM          (P_fmapM, P_fmapMr, pmapMr, pmapM)
import PolyLib.Zip            (P_fzip, P_fthread, pzipWith)
import Control.Monad.State    (runState, put, get)
```

Apart from a few modules from the polytypic library we also use the state monad implementation from Control.Monad.State. The function $runState :: \mathsf{State}\ s\ a \to s \to (a, s)$ runs a computation in a given state and returns a pair of the result and the final state, and the functions *get* and *put* are used to read and write the state.

```
separate :: (FunctorOf f d, P_fmap2Mr f) ⇒ d a → (d (), [a])
separate x = runState (pmapMr store x) []
    where store x = do  xs ← get
                        put (x : xs)
                        return ()
```

The function *separate* takes an element of a regular datatype $d\ a$ and separates the shape from the contents, returning a d-structure of unit values (the shape) and a list of as (the contents). We use a state monad and the monadic map function *pmapMr* (see Section A.4) to traverse the structure, store away all the as in the state and replace them with unit values. Since *store* adds elements at the head of the list we have to traverse the structure in reverse order to get the elements in the right order.

```
combine :: (FunctorOf f d, P_fmap2M f) ⇒ d () → [a] → d a
combine s xs = fst (runState (pmapM load s) xs)
    where load () = do  x : xs ← get
                        put xs
                        return x
```

The (left) inverse of *separate* is the function *combine* that takes a shape and a list of values and inserts the values into the shape. We replace each unit value in the structure with an element from the state, by mapping the *load* function over the structure.

```
pZipWith_ :: (FunctorOf f d, P_fmap2 f, P_fzip f, P_fthread f) ⇒
             (a → b → c) → d a → d b → d c
pZipWith_ f x y = z
    where f' (x, y) = Just (f x y)
          Just z    = pzipWith f' (x, y)
```

The *pzipWith* function defined in Section A.13 uses the maybe monad to handle the case when the zipped structures do not match. Since we do not care about the exceptional cases here, we define a function *pzipWith_* that gives an error if the structures do not match.

```
listTranspose :: (FunctorOf f d, P_fzip f, P_fthread f, P_fmap2 f) ⇒
                 [d a] → d [a]
```

```
listTranspose [x]      = pmap (:[]) x
listTranspose (x : xs) = pzipWith_ (:) x (listTranspose xs)
```

The special case of transposing the list datatype and another regular datatype d is a bit easier than the general case. If we get a list of a single d-structure x, we should return a d-structure of the same shape as x, but where all the elements are singleton lists, so we just map the function $(:[])$ over x. If we get a list $x : xs$ of more than one d-structure, we transpose the list xs, resulting in a d-structure of lists. To each of these lists we add the corresponding element of x by zipping the two structures together.

```
transpose :: ( FunctorOf f d,
               P_fmap2Mr f,
               P_fmap2M f,
               FunctorOf g e,
               P_fmap2 g,
               P_fthread g,
               P_fzip g
             ) ⇒ d (e a) → e (d a)
transpose x = pmap (combine s) (listTranspose es)
    where (s, es) = separate x
```

Finally the *transpose* function separates the shape and the contents of its argument, yielding a d-shape, $s :: d\,()$, and a list of e-structures, $es :: [e\,a]$. The list es can be transposed using *listTranspose* resulting in an e-structure of lists. Now we just force each of these lists into the shape s and we have an e-structure of d-structures.

Bibliography

[1] A. Alimarine and R. Plasmeijer. A generic programming extension for Clean. In T. Arts and M. Mohnen, editors, *Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001*, volume 2312 of *LNCS*, pages 168–185. Springer-Verlag, 2001.

[2] A. Alimarine and S. Smetsers. Optimizing generic functions. In D. Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 16–31. Springer-Verlag, 2004.

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[4] D. Clarke and A. Löh. Generic haskell, specifically. In J. Gibbons and J. Jeuring, editors, *Proceedings of the IFIP TC2 Working Conference on Generic Programming*, pages 21–48. Kluwer, 2003.

[5] R. Cockett and T. Fukushima. About Charity. Yellow Series Report No. 92/480/18, Dep. of Computer Science, Univ. of Calgary, 1992.

[6] J. Furuse. Generic polymorphism in ML. In *Journées Francophones des Langages Applicatifs*, 2001.

[7] R. Hinze. Polytypic values possess polykinded types. In *Mathematics of Program Construction*, volume 1837 of *LNCS*, pages 2–27. Springer-Verlag, 2000.

[8] R. Hinze. Generics for the masses. In K. Fisher, editor, *Proceedings of the 2004 International Conference on Functional Programming*, 2004.

[9] R. Hinze and J. Jeuring. Generic Haskell: Applications. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 57–97. Springer-Verlag, 2003.

[10] R. Hinze and J. Jeuring. Generic Haskell: Practice and theory. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 1–56. Springer-Verlag, 2003.

[11] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science, 2001.

[12] G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.

[13] P. Jansson. *Functional Polytypic Programming.* PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden, May 2000.

[14] P. Jansson. The WWW home page for polytypic programming. Available from `http://www.cs.chalmers.se/~patrikj/poly/`, 2003.

[15] P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *POPL'97*, pages 470–482. ACM Press, 1997.

[16] P. Jansson and J. Jeuring. PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand, June 1998. Available from the Polytypic programming WWW page [14].

[17] P. Jansson and J. Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In *Workshop on Generic Programming.* Utrecht University, 2000. UU-CS-2000-19.

[18] P. Jansson and J. Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

[19] C. Jay and P. Steckler. The functional imperative: shape! In C. Hankin, editor, *Programming languages and systems: 7th European Symposium on Programming, ESOP'98*, volume 1381 of *LNCS*, pages 139–53. Springer-Verlag, 1998.

[20] M. P. Jones. Type classes with functional dependencies. *Lecture Notes in Computer Science*, 1782:230–??, 2000.

[21] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, Mar. 2003. Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[22] R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings; International Conference on Functional Programming (ICFP 2004).* ACM Press, Sept. 2004. 12 pages; To appear.

[23] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, Jan. 2002.

[24] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[25] C. McBride. Faking it—Simulating dependent types in Haskell, 2001.

[26] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In J. Hughes, editor, *FPCA'91: Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, 1991.

[27] U. Norell. Functional generic programming and type theory. Master's thesis, Computing Science, Chalmers University of Technology, 2002. Available from `http://www.cs.chalmers.se/~ulfn`.

[28] U. Norell and P. Jansson. Polytypic programming in Haskell. In *Implementation of Functional Languages*, LNCS, 2004. In press. Presented at IFL'03.

[29] U. Norell and P. Jansson. Prototyping generic programming in Template Haskell. In D. Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 314–333. Springer-Verlag, 2004.

[30] S. Peyton Jones [editor], J. Hughes [editor], L. Augustsson, D. Barton, B. Boutel, W. Burton, S. Fraser, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/definition/`, Feb. 1999.

[31] T. Sheard. Generic unification via Two-Level types and parameterized modules. In *ICFP'01*, pages 86–97, 2001.

[32] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proceedings of the Haskell workshop*, pages 1–16. ACM Press, 2002.

[33] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, 1994.