Polytypic Programming in Haskell

Ulf Norell
ulfn@cs.chalmers.se

Patrik Jansson
patrikj@cs.chalmers.se

ABSTRACT

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.

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 a 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, there is no need to change the polytypic function.

1.1 Related work

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

- The Haskell extension PolyP [7] allows user-defined polytypic definitions over regular datatypes.
- Derivable type classes [5] is an extension of the Glasgow Haskell Compiler (ghc) which allows limited polytypic definitions. A similar extension exists also for Clean [1].
- Generic Haskell [4] allows polytypic definitions over all Haskell datatypes.
- Recently the DrIFT preprocessor for deriving non-standard Haskell classes has been used together with the Strafunski library [11] to provide generic programming in Haskell.

Other implementations of functional polytypism include Charity [2], FISh [10] and G'Caml [3] but in this paper we focus on the Haskell-based languages.

Specialization is used in both PolyP and Generic Haskell to implement polytypic programming. For each datatype a polytypic function is used at, a specialized version is generated. 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. This technique has been implemented as a Haskell library and as a modification of the PolyP compiler (PolyP version 2). The implementation of PolyP 2 is available from the polytypic programming home page [6]. In the following text we normally omit the version number — PolyP will stand for the the improved language and its (new) compiler.

We can identify a few disjoint sublanguages within a polytypic language:

PolyCore	Polytypic definitions (syntactic extension)
PolyUse	Polytypic definitions in terms of definitions in PolyCore
PolyInst	Instantiating polytypic definitions on specific types
Base	The base language (no polytypic functions) — Haskell
Regular	Definitions of regular datatypes

Using a specializing compiler translating into Base (such as Generic Haskell and old PolyP) we have to compile at least PolyCore, PolyUse, PolyInst and Regular. With the new PolyP we only need to compile PolyCore (and may compile Regular), thus making it possible to write a library of polytypic functions, compile it into Haskell and use it just like any library of regular Haskell functions.

The generic programming library Strafunski/DrIFT requires even less (apart from ghc extensions) for doing generic programming in Haskell: only Regular needs to pass through the DrIFT tool. But neither Generic Haskell nor Strafunski/DrIFT can define the general recursion combinators like catamorphisms and anamorphisms for which PolyP is well known. In PolyP, polytypic functions are restricted to unary, regular datatypes. This restriction turns out to be sufficient to make the general recursion combinators expressible inside Haskell.

1.2 Overview

The rest of this paper is structured as follows. Section 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 shows how the implementation of PolyP has been extended to translate PolyP code to Haskell classes and instances. Section 4 and Section 5 show two case studies of using PolyP and Haskell for polytypic programming. Section 6 concludes.

2. POLYTYPISM IN HASKELL

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

The embedding closely mimics the features of the language PolyP [7], 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.

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, does not contain any function spaces and the arguments to the type constructor are the same in the left-hand side and the right-hand side of the definition.

```
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 Empty p r           = Empty
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}
```

Figure 1: Pattern functor combinators

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 1. (The infix combinators are right associative with (:@:) having the highest precedence and (:+:) the lowest.) For instance 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 = Empty :+: Par :*: Rec
```

An element of ListF p r is either InL Empty corresponding to Nil or it is InR (Par x :∗: Rec xs) corresponding to Cons x xs.

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
```

The last pattern functor Const t captures constant types in datatype definition such as in a binary tree type storing the height of each node.

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

In general we write Φ_D for the pattern functor of the datatype D a, so for example Φ_{List} = ListF. To convert between a datatype and its pattern functor we use the functions *inn* and *out* in the multi-parameter type class FunctorOf shown in figure 2. These functions realize the isomorphism $d\ a \cong \Phi_d\ a\ (d\ a)$, that holds for every regular datatype. (We can view a regular datatype d a as the least fixed point of the corresponding functor Φ_d a.)

```
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
```

Figure 2: The FunctorOf class

In our list example we have

```
instance FunctorOf (Empty :+: Par :*: Rec) List where
    inn (InL Empty)                = Nil
    inn (InR (Par x :*: Rec xs))   = Cons x xs
    out Nil                        = InL Empty
    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. The second parameter of the pattern functor (i.e the recursive call) is an element of the actual datatype.

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 the same pattern functor if they share the same structure, but one datatype can not have more than one associated pattern functor. The members *constructorName* and *datatypeName* (and a few more for fixity and precedence information) are used in the definition of polytypic show and read functions.

2.2 Pattern functor classes

In addition to the class FunctorOf, used to relate datatypes to pattern functors, we also use one *pattern functor class* P_name for each (group of related) polytypic definition(s) *name*. A pattern functor class is just a constructor class with one parameter of kind $\star \to \star \to \star$ with one (or more) polytypic definitions as members. The set of instances for a class P_name defines for which pattern functors the polytypic definition(s) *name* is meaningful.

An example is a generalization of the standard Haskell Prelude class Functor:

```
class Functor f where
    fmap  ::  (a -> b) -> (f a -> f b)
```

The two-argument version of Functor is the pattern functor class P_fmap2:

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

All the pattern functors are instances of the class P_fmap2. Pattern functor classes and their instances are discussed in more detail in section 3.

2.3 PolyLib in Haskell

PolyLib [8] is a library of polytypic definitions including generalized versions of *map*, *zip*, *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. The library functions can be used on all datatypes which are instances of the FunctorOf class and if the user provides the FunctorOf-instances, no tool support is needed. Alternatively, for all regular datatypes, these instances can be generated automatically by the new PolyP compiler (or by DrIFT).

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)
pmap f  =   inn ∘ fmap2 f (pmap f) ∘ out

cata    ::  (FunctorOf f d, P_fmap2 f) =>
            (f a b -> b) -> (d a -> b)
cata φ  =   φ ∘ fmap2 id (cata φ) ∘ out

ana     ::  (FunctorOf f d, P_fmap2 f) =>
            (b -> f a b) -> (b -> d a)
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 member function *fsum* that takes care of the summing at the top-level, provided that the recursive occurrences have already been summed:

```
fsum :: Num a => f a a -> a
```

Then we can sum the elements of a regular datatype by defining

```
psum :: (FunctorOf f d, P_fmap2 f, P_fsum f, Num a)
        => d a -> a
psum = cata fsum
```

We will return to the function *fsum* in section 3.1 when we discuss how the pattern functor classes are defined. 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 to allow wildcards.

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

```
data Bin a  =  Single a
            |  Fork (Bin (a, a))
```

We can view Bin a as a regular datatype with the pattern functor Par :+: Rec :*: Rec, i.e. the same as the ordinary binary tree

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

To do this we give the following instance of the FunctorOf class for Bin.

```
instance FunctorOf (Par :+: Rec :*: Rec) Bin where
    inn (InL (Par x))              =  Single x
    inn (InR (Rec l :*: Rec r))    =  Fork t
        where t                    =  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"
```

This instance declaration uses the functions *split* and *join* on perfect binary trees, defined by

```
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
```

Having defined the above instance allows us to use all the PolyLib functions on perfect binary trees. For instance we could use an anamorphism to generate a full binary tree of a given height as follows.

```
full :: a → Int → Bin a
full x = ana f
    where f 0          =  InL (Par x)
          f (n + 1)    =  InR (Rec n) (Rec n)
```

Note that this definition of *full* can be given a more general type. We could use it to generate a full tree of type Tree a, or in fact any regular datatype with the same pattern functor as Bin, just by changing the type signature.

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
```

We can view the stack as a regular datatype with pattern functor Empty :+: Par :*: Rec, i.e. the same structure as a list, by giving the instance

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

    out s =  case pop s of
               Nothing      →  InL Empty
               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.

2.6 Polytypic functions in Haskell

We have seen how to make different kinds of datatypes fit the polytypic framework 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*

```
coerce :: (FunctorOf f d, FunctorOf f e, P_fmap2 f)
          ⇒ d a → e a
coerce = cata inn
```

that converts between two regular datatypes with the same pattern functor. For instance we could convert a perfect binary tree from section 2.4 to a normal binary tree or convert a list to an element of the abstract stack type from section 2.5. Interestingly we cannot define the *coerce* function in PolyP since the compiler would infer the same type for the argument and the result.

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

```
instance (FunctorOf f d, P_fmap2 f)
           ⇒ Functor d where
    fmap = pmap
```

This requires a few extensions to be enabled in the Haskell compiler, in particular we need overlapping and undecidable instances, apart from multi-parameter type classes. In this example we are forced to give the class constraints explicitly, since the Haskell compiler cannot infer the constraints on an instance declaration.

Within the polytypic framework we can also define more complex functions such as the *transpose* function that transposes two regular datatypes.

```
transpose :: d (e a) → Maybe (e (d a))
```

Using *transpose* we can convert, for instance, a list of trees to a tree of lists. The definition of *transpose* is rather involved and the current implementation is not fit to be presented here. The important point, however, is that it *can* be defined, something that demonstrates the power of this framework.

3. A POLYTYPIC HASKELL EXTENSION

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 3.1 below, we describe how to define polytypic functions from scratch using a modified version of the PolyP language [7]. The polytypic definitions in PolyP *can* also be expressed in Haskell, but the syntax of the language extension is more convenient than writing the classes and the instances by hand. Sections 3.2 to 3.6 discusses how the PolyP definitions are compiled into Haskell.

3.1 The polytypic construct

In section 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.

$$\begin{array}{lcl}
(f \triangledown g)\ (\mathsf{InL}\ x) & = & f\ x \\
(f \triangledown g)\ (\mathsf{InR}\ y) & = & g\ y \\
f \mathbin{-\!\!+\!\!-} g & = & (\mathsf{InL} \circ f) \triangledown (\mathsf{InR} \circ g) \\
\\
\mathit{first}\ (x :\!*\!: y) & = & x \\
\mathit{second}\ (x :\!*\!: y) & = & y \\
(f \triangle g)\ x & = & f\ x :\!*\!: g\ x \\
f \mathbin{-\!\!*\!\!-} g & = & (f \circ \mathit{first}) \triangle (g \circ \mathit{second})
\end{array}$$

Figure 3: Functions over the pattern functors

To facilitate the definition of polytypic functions we define a few useful functions to manipulate the pattern functors (see figure 3). ($\triangledown$) and ($\mathbin{-\!\!+\!\!-}$) are the elimination and map functions for sums, ($\triangle$) is the split function, ($\mathbin{-\!\!*\!\!-}$) is the map function and *first* and *second* are the projection functions for products. The types of these functions are a little more complex than one would like, since they operate on binary functors. For this reason we have chosen to omit them in this presentation.

Using the type case construct and the functions above, we can define the function *fsum* from section 2.3 that operates on pattern functors applied to some numeric type.

$$\begin{array}{l}
\textbf{polytypic}\ \mathit{fsum} :: \mathsf{Num}\ a \Rightarrow f\ a\ a \to a \\
\quad = \textbf{case}\ f\ \textbf{of} \\
\qquad \begin{array}{lcl}
g :\!+\!: h & \to & \mathit{fsum} \triangledown \mathit{fsum} \\
g :\!*\!: h & \to & \lambda(x :\!*\!: y) \to \mathit{fsum}\ x + \mathit{fsum}\ y \\
\mathsf{Empty} & \to & \mathit{const}\ 0 \\
\mathsf{Par} & \to & \mathit{unPar} \\
\mathsf{Rec} & \to & \mathit{unRec} \\
d :\!@\!: g & \to & \mathit{psum} \circ \mathit{pmap}\ \mathit{fsum} \circ \mathit{unComp} \\
\mathsf{Const}\ t & \to & \mathit{const}\ 0
\end{array}
\end{array}$$

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 Empty or a constant type is zero and when we get one Par and Rec they already contain a number so we just return it. If the pattern functor is a regular datatype $d\ a$ composed with a pattern functor g we map *fsum* over d and use the function *psum* to sum the result. Note that since the pattern functors are not type synonyms we have to remove the pattern functor constructors explicitly.

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

$$\begin{array}{l}
\textbf{polytypic}\ p :: \tau \\
\quad = \lambda\, x_1\ \ldots\ x_m \to \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad \begin{array}{ccc}
\rho_1 & \to & e_1 \\
 & \vdots & \\
\rho_n & \to & e_n
\end{array}
\end{array}$$

where f is the pattern functor (occurring somewhere in τ) and ρ_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 $\lambda\, x_1\ \ldots\ x_m \to e_i :: \tau[\rho_i/f]$.

A **polytypic** definition operates on the pattern functor level, but what we are really interested in are functions on the datatype level. 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 sum function *psum* (that is actually used in the $d :\!@\!: g$ case of the definition of *fsum*). This function can be defined as the catamorphism of *fsum*:

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

The class constraint $\mathsf{Regular}\ d$ in PolyP, corresponds to the constraint $\mathsf{FunctorOf}\ \Phi_d\ d$ and constraints for any suitable pattern functor classes on Φ_d in the Haskell code.

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.

3.2 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 three phases each of which is described in the following subsections. In the first phase, described in section 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 3.4) 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 3.5 and consists of inferring the class constraints introduced by our new classes. Section 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 3.1 is exactly the same in the generated Haskell code as in the PolyP code. The type on the other hand does change, but this is taken care of in phase three.

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 describes the datatype structure and *content* combinators that describes the contents of the datatype. The structure combinators, (:+:), (:*:) and Empty, 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 introduce the the *meaning* of g, denoted by $\widehat{g}$, defined by

$$\begin{array}{lcl} \widehat{\mathsf{Par}\ p\ r} & = & p \\ \widehat{\mathsf{Rec}\ p\ r} & = & r \\ \widehat{\mathsf{Const}\ t\ p\ r} & = & t \\ \widehat{d\ {:}@{:}\ g\ p\ r} & = & d\ (\widehat{g}\ p\ r) \end{array}$$

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

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

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 Empty. 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

$$\begin{array}{lcl} to_g & :: & \widehat{g}\ p\ r \rightarrow g\ p\ r \\ from_g & :: & g\ p\ r \rightarrow \widehat{g}\ p\ r \\ \\ \multicolumn{3}{l}{to_g \circ from_g = id} \\ \multicolumn{3}{l}{from_g \circ to_g = id} \end{array}$$

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 d :@: g we also have to map the conversion function for g over the regular datatype $d\ a$, as shown below.

$$\begin{array}{lcl} to_{\mathsf{Par}} & = & \mathsf{Par} \\ from_{\mathsf{Par}} & = & unPar \\ to_{\mathsf{Rec}} & = & \mathsf{Rec} \\ from_{\mathsf{Rec}} & = & unRec \\ to_{\mathsf{Const}\ t} & = & \mathsf{Const} \\ from_{\mathsf{Const}\ t} & = & unConst \\ to_{d@g} & = & \mathsf{Comp} \circ pmap\ to_g \\ from_{d@g} & = & pmap\ from_g \circ unComp \end{array}$$

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

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

For instance the second constructor out of three corresponds to $\iota^3_2\ x = \mathsf{InR}\ (\mathsf{InL}\ x)$ (remember that (:+:) is right associative).

Finally an instance FunctorOf Φ_D D for the general form of a regular datatype $D\ a$ can be defined as shown in figure 4.

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathbf{instance}\ \mathsf{FunctorOf}\ \Phi_D\ D\ \mathbf{where}} \\ \quad inn\ (\iota^n_k\ (x_1 :\!*\!: \ldots :\!*\!: x_{m_k})) & = & \\ \qquad C_k\ (to_{g_{k1}}\ x_1)\ \ldots\ (to_{g_{km_k}}\ x_{m_k}) & & \\ \\ \quad out\ (C_k\ x_1\ \ldots\ x_{m_k}) & = & \\ \qquad \iota^n_k\ (from_{g_{k1}}\ x_1 :\!*\!: \ldots :\!*\!: from_{g_{km_k}}\ x_{m_k}) & & \\ \\ \quad constructorName\ (C_k\ \_\ \ldots\ \_) & = & \text{“}C_k\text{”} \\ \quad datatypeName\ \_ & = & \text{“}D\text{”} \end{array}$$

Figure 4: The FunctorOf instance for the general form of a regular datatype

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 member (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 5 the translation produces the result on the right.

$$\boxed{\begin{array}{l} \mathbf{polytypic}\ p :: \tau \\ \quad = \mathbf{case}\ f\ \mathbf{of} \\ \qquad\quad \rho_1 \ \rightarrow\ e_1 \\ \qquad\qquad\ \vdots \\ \qquad\quad \rho_n \ \rightarrow\ e_n \end{array}} \Longrightarrow \boxed{\begin{array}{l} \mathbf{class}\ \mathsf{P\_}p\ f\ \mathbf{where} \\ \quad p :: \tau \\ \\ \mathbf{instance}\ \mathsf{P\_}p\ \rho_1\ \mathbf{where} \\ \quad p = e_1 \\ \qquad\qquad \vdots \\ \mathbf{instance}\ \mathsf{P\_}p\ \rho_n\ \mathbf{where} \\ \quad p = e_n \end{array}}$$

Figure 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, for example, in the definition of *fsum* from section 3.1, the instance P_fsum $(g :+: h)$ needs instances of P_fsum for g and h. How to infer these constraints is the topic of the next section.

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 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. In the case of instance declarations of the generated classes we *have* to provide the class constraints (as discussed in the previous section). Look at the instance of the class P_fmap2 for $g :+: h$:

```
instance (P_fsum g, P_fsum h) ⇒
           P_fsum (g :+: h) where
    fsum = ...
```

Here the constraints P_fsum g and P_fsum h have to be stated explicitly and cannot be inferred by the Haskell compiler.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 function *pmap* is

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

which is a little too general to be practical. This second category of troublesome definitions is taken care of simply by unifying the inferred type with the type stated in the PolyP code. When doing this we have to replace the constraint Regular d in the PolyP type, by the corresponding Haskell constraint 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

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

with the modified version of the type stated in the PolyP code

$$(\mathsf{FunctorOf}\ f\ d, \alpha) \Rightarrow (a \to b) \to d\ a \to d\ b$$

Here e would be identified with d and α 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 member function body and unify this type with the expected type of the member function. We take the definition of *fsum* as an example.

```
polytypic fsum :: Num a ⇒ f a a → a
   = case f of
       g :+: h   →   fsum ∇ fsum
       ⋮
```

This definition is translated to a class

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

and instance declarations for each branch. In particular we get the following instance for the pattern functor $g :+: h$:

```
instance P_fsum (g :+: h) where
    fsum = fsum ∇ fsum
```

We can now infer the type of the function body to be

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

We add a constraint set variable α to the type of *fsum* as given in the class declaration, serving as a place holder for the extra class constraints:

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

Finally we unify the inferred type and the expected type to get the following substitution

$$\begin{array}{lcl} f & \longrightarrow & g :+: h \\ \alpha & \longrightarrow & \{\mathsf{P\_fsum}\ g, \mathsf{P\_fsum}\ h\} \end{array}$$

The identification of f and $g :+: h$ is exactly what we expected and is a requirement for the instance declaration to be type correct. The part of the substitution that we are interested in is the assignment of α, i.e. the class constraints that are in the instance declaration but not in the class declaration. To get a type correct instance we have to add these constraints to the instance head yielding the following final instance of P_fsum $(g :+: h)$:

```
instance  (P_fsum g, P_fsum h) ⇒
            P_fsum (g :+: h) where
    fsum = fsum ∇ fsum
```

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 (i.e. with polytypic class constraints), 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 Map (pmap)

polytypic fsum :: ...
psum = ... pmap ... fsum ...
```

we would generate a Haskell module looking like this:

```
module Sum (psum, P_fmap2, P_fsum) where
import Map (pmap, P_fmap2)
⋮
```

The P_fmap2 in the import declaration comes from the type of *pmap*, which is looked up in the interface file for the module Map, and the two exported classes comes from the inferred type of *psum*.

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 $[1, 2]$. It is still possible, however, to define a show function that produces the latter output and we have an implementation of a polytypic show function that mimics the behavior of the show function that can be derived by the Haskell compiler.

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
```

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 *a*s to strings and a recursive call to *pshow* to convert the *d a*s 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 don't 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 *fshowRest*. 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   →   fshowSum ∇ fshowSum
        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. The generated code is accepted by current ghc and hugs (with suitable extensions turned on), but if more portable Haskell 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
        Empty     →   const ""
        g         →   λx →  " (" ++ fshowRest 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 *fshowRest*. Finally we define the function *fshowRest* as

```
polytypic fshowRest :: f String String → String
    = case f of
        Par       →   unPar
        Rec       →   unRec
        d :@: g   →   pshow fshowRest ∘ 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 *fshowRest* 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 also illustrates a problem that arises when using polytypic functions in a Haskell program, namely that the class constraints quickly get out of hand. In many cases you can omit the types, thus avoiding the problem, but in the case of an instance declaration like this you are forced to give the constraints. 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 there.

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 [9], 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 →
            (Just ∘ unConst) ∇ const 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
    = λx → case f of
        Const Var :+: h → InL (Const x)
```

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            →  fbindCheck
                              ∇ fbindCheck
        Const Var :∗: h    →  Just ∘ unConst ∘ first
        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 can't 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
    = λx → case f of
        g :+: h            →  frenameBound x
                              -+- frenameBound x
        Const Var :∗: h    →  (const $ Const x)
                              -∗- id
        g                  →  const 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 example 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 t       =  fvarCheck (out t)
pVar x            =  inn (fVar x)
pbindCheck t      =  fbindCheck (out t)
prenameBound x    =  inn ∘ frenameBound x ∘ out
```

Note that we have omitted the types of the functions. One of the problems when using polytypic functions in Haskell is that the class constraints can get quite complicated. In this case they are not that long, for instance

$$pVar :: (\mathsf{FunctorOf}\ f\ d, \mathsf{P\_fVar}\ f) \Rightarrow \mathsf{Var} \rightarrow d\ a$$

but since we get one extra constraint for each polytypic function we use, they can quickly become very large. This is not really that much of a problem, though, since the Haskell compiler can infer the types for us.

Apart from these functions for manipulating variables and variable bindings we are going to 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
```

Here we include the types of the functions as comments to improve the readability. 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.

Before we start defining more interesting functions we combine *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 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 rec $
                             prenameBound z t
      where
        vs   =  pfreeVars u `union` pfreeVars t
        z    =  freshVar vs
        rec  =  subst (y, u) ∘ subst (x, pVar z)

    _            →  pmapC (subst (y, u)) 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. If t is just a variable then we return u if the variable is y, otherwise we return t itself. If t is a variable binding we return t if the bound variable is y. The most difficult case is when t binds a variable x that occurs free in u, in which case we have to alpha rename t to avoid capturing the free occurrences of x in u. First we compute a fresh variable z by applying the function *freshVar* to the list of free variables in t and u, then we rename the binding occurrence of x in t to z using the function *prenameBound* and substitute z for x in the rest of t with a recursive call to *subst* on the children of t. Finally we substitute u for y in the children of t. 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, as we have seen previously, be generated by the PolyP compiler so we can just place our datatype declaration in a separate module and run it through the compiler. Having done this we can import this module together with the Haskell module containing the term function into our Haskell program and start playing around with our lambda 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

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

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

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 and the definition of the example datatype.

6. CONCLUSIONS

In this paper we have shown how to bring 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 [8] 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 custom polytypic 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 give hand made FunctorOf instances, thus extending the applicability of the polytypic functions to datatypes that are not necessarily regular.

One direction for future work could be to use Template Meta-Haskell [13] to internalize the PolyP compiler as a ghc extension. Other research directions are to extend our approach to more datatypes (partially explored in [12]), or to explore in more detail which polytypic functions are expressible in this setting.

7. REFERENCES

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

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

[4] R. Hinze and J. Jeuring. Generic haskell: Practice and theory. To appear in the lecture notes of the Summer School on Generic Programming, LNCS Springer-Verlag, 2002/2003.

[5] 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.

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

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

[8] P. Jansson and J. Jeuring. PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand, June 1998. Available from the Polytypic programming WWW page [6].

[9] 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.

[10] 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.

[11] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In A. SIGPLAN, editor, *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*. ACM Press, 2003. To appear in ACM SIGPLAN Notices.

[12] 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`.

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