# Polytypic Programming in Haskell

Ulf Norell and Patrik Jansson

Computing Science, Chalmers University of Technology, Sweden
http://www.cs.chalmers.se/~{ulfn,patrikj}/
{ulfn,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.

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 that 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 [8] compiler (PolyP version 2). The implementation of PolyP 2 is available from the polytypic programming home page [7]. In the following text we normally omit the version number — PolyP will stand for the the improved language and its (new) compiler.

### 1.1 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 discusses briefly the structure of a polytypic language. Section 5 describes related work and 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 to their structures.

The embedding closely mimics the features of the language PolyP [8], 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 with another typeand if the argument to the type constructor is the same in the left-hand side and the right-hand side of the definition.

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 and their order of precedence is, from lower to higher: (:+:), (:*:), (:->:), (:@:).) For instance for

$$
\begin{aligned}
&\mathbf{data}\ (g :+: h)\ p\ r && = \mathsf{InL}\ (g\ p\ r)\ \mid\ \mathsf{InR}\ (h\ p\ r)\\
&\mathbf{data}\ (g :*: h)\ p\ r && = g\ p\ r :*: h\ p\ r\\
&\mathbf{data}\ \mathsf{Empty}\ p\ r && = \mathsf{Empty}\\
&\mathbf{newtype}\ \mathsf{Par}\ p\ r && = \mathsf{Par}\ \{unPar\ ::\ p\}\\
&\mathbf{newtype}\ \mathsf{Rec}\ p\ r && = \mathsf{Rec}\ \{unRec\ ::\ r\}\\
&\mathbf{newtype}\ (d :@: h)\ p\ r && = \mathsf{Comp}\ \{unComp\ ::\ d\ (g\ p\ r)\}\\
&\mathbf{newtype}\ \mathsf{Const}\ t\ p\ r && = \mathsf{Const}\ \{unConst\ ::\ t\}\\
&\mathbf{newtype}\ (g :\rightarrow: h)\ p\ r && = \mathsf{Fun}\ \{unFun\ ::\ g\ p\ r \rightarrow\ h\ p\ r\}
\end{aligned}
$$

**Fig. 1.** Pattern functor combinators

the datatype List $a$ we can use these combinators to define the pattern functor ListF as follows:

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{List}\ a = \mathsf{Nil} \mid \mathsf{Cons}\ a\ (\mathsf{List}\ a)\\
&\mathbf{type}\ \mathsf{ListF}\ = \mathsf{Empty} :+: \mathsf{Par} :*: \mathsf{Rec}
\end{aligned}
$$

An element of ListF $p\ r$ can take either the form InL Empty, corresponding to Nil or the form 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:

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{Rose}\ a = \mathsf{Fork}\ a\ (\mathsf{List}\ (\mathsf{Rose}\ a))\\
&\mathbf{type}\ \mathsf{RoseF}\ = \mathsf{Par} :*: \mathsf{List} :@: \mathsf{Rec}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{HTree}\ a = \mathsf{Leaf}\ a \mid \mathsf{Branch}\ \mathsf{Int}\ (\mathsf{HTree}\ a)\ (\mathsf{HTree}\ a)\\
&\mathbf{type}\ \mathsf{HTreeF}\ = \mathsf{Par} :+: \mathsf{Const}\ \mathsf{Int} :*: \mathsf{Rec} :*: \mathsf{Rec}
\end{aligned}
$$

The pattern functor $(:\rightarrow:)$ is used to model datatypes with function spaces. Only a few polytypic functions are possible to define for such datatypes. We include the combinator $(:\rightarrow:)$ here because our system can handle it, but for the rest of the paper we assume regular datatypes *without* function spaces.

In general we write $\Phi_D$ 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:

$$
\begin{aligned}
&\mathbf{class}\ \mathsf{FunctorOf}\ f\ d \mid d \rightarrow f\ \mathbf{where}\\
&\quad inn ::\ f\ a\ (d\ a) \rightarrow d\ a\\
&\quad out ::\ d\ a \rightarrow f\ a\ (d\ a)
\end{aligned}
$$

The functions *inn* and *out* 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 $\Phi_d\ a$.)

In our list example we have

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{FunctorOf}\ (\mathsf{Empty}\ {:}{+}{:}\ \mathsf{Par}\ {:}{*}{:}\ \mathsf{Rec})\ \mathsf{List}\ \textbf{where} \\
\quad \begin{array}{lcl}
inn\ (\mathsf{InL}\ \mathsf{Empty}) & = & \mathsf{Nil} \\
inn\ (\mathsf{InR}\ (\mathsf{Par}\ x\ {:}{*}{:}\ \mathsf{Rec}\ xs)) & = & \mathsf{Cons}\ x\ xs \\
out\ \mathsf{Nil} & = & \mathsf{InL}\ \mathsf{Empty} \\
out\ (\mathsf{Cons}\ x\ xs) & = & \mathsf{InR}\ (\mathsf{Par}\ x\ {:}{*}{:}\ \mathsf{Rec}\ xs)
\end{array}
\end{array}
$$

Note that *inn* (*out*) only folds (unfolds) the top level structure and it is therefore normally a constant time operations.

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.

### 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 methods. 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 to the pattern functor class P_fmap2:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Functor}\ f\ \textbf{where} \\
\quad fmap\ ::\ (a \to b) \to (f\ a \to f\ b) \\
\textbf{class}\ \mathsf{P\_fmap2}\ f\ \textbf{where} \\
\quad fmap2\ ::\ (a \to c) \to (b \to d) \to (f\ a\ b \to f\ c\ d)
\end{array}
$$

All pattern functors except $({:}{\to}{:})$ 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 [9] 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. 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, or potentially by Template Haskell).

Using $\mathit{fmap}2$ from the $\mathsf{P\_fmap2}$-class and $\mathit{inn}$ and $\mathit{out}$ from the $\mathsf{FunctorOf}$ class we can already define quite a few polytypic functions from the Haskell version of PolyLib. For instance

$$\begin{array}{lll}
\mathit{pmap} & :: (\mathsf{FunctorOf}\ f\ d, \mathsf{P_fmap2}\ f) \Rightarrow & (a \to b) \to (d\ a \to d\ b) \\
\mathit{cata} & :: (\mathsf{FunctorOf}\ f\ d, \mathsf{P_fmap2}\ f) \Rightarrow & (f\ a\ b \to b) \to (d\ a \to b) \\
\mathit{ana} & :: (\mathsf{FunctorOf}\ f\ d, \mathsf{P_fmap2}\ f) \Rightarrow & (b \to f\ a\ b) \to (b \to d\ a)
\end{array}$$

$$\begin{array}{ll}
\mathit{pmap}\ f & = \mathit{inn} \circ \mathit{fmap}2\ f\ (\mathit{pmap}\ f) \circ \mathit{out} \\
\mathit{cata}\ \varphi & = \varphi \circ \mathit{fmap}2\ \mathit{id}\ (\mathit{cata}\ \varphi) \circ \mathit{out} \\
\mathit{ana}\ \psi & = \mathit{inn} \circ \mathit{fmap}2\ \mathit{id}\ (\mathit{ana}\ \psi) \circ \psi
\end{array}$$

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

$$\mathit{fsum} :: \mathsf{Num}\ a \Rightarrow f\ a\ a \to a$$

(Method $\mathit{fsum}$ 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

$$\begin{array}{ll}
\mathit{psum} & :: (\mathsf{FunctorOf}\ f\ d,\ \mathsf{P_fmap2}\ f, \mathsf{P_fsum}\ f, \mathsf{Num}\ a) \Rightarrow d\ a \to a \\
\mathit{psum} & = \mathit{cata}\ \mathit{fsum}
\end{array}$$

We return to the function $\mathit{fsum}$ in section 3.1 when we discuss how the pattern functor classes are defined. In the type of $\mathit{psum}$ 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

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

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

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

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{FunctorOf}\ (\mathsf{Par}\ {:}{+}{:}\ \mathsf{Rec}\ {:}{*}{:}\ \mathsf{Rec})\ \mathsf{Bin}\ \textbf{where} \\
\quad \mathit{inn}\ (\mathsf{InL}\ (\mathsf{Par}\ x)) \qquad\qquad\quad = \mathsf{Single}\ x \\
\quad \mathit{inn}\ (\mathsf{InR}\ (\mathsf{Rec}\ l\ {:}{*}{:}\ \mathsf{Rec}\ r)) = \mathsf{Fork}\ (\mathit{join}\ (l, r)) \\
\quad \mathit{out}\ (\mathsf{Single}\ x) \qquad\qquad\quad\ \ = \mathsf{InL}\ (\mathsf{Par}\ x) \\
\quad \mathit{out}\ (\mathsf{Fork}\ t) \qquad\qquad\qquad\ = \mathsf{InR}\ (\mathsf{Rec}\ l\ {:}{*}{:}\ \mathsf{Rec}\ r) \\
\qquad\quad \textbf{where}\ (l, r) \qquad\qquad\ \ = \mathit{split}\ t
\end{array}
$$

$$
\begin{array}{l}
\mathit{join} :: (\mathsf{Bin}\ a, \mathsf{Bin}\ a) \to \mathsf{Bin}\ (a, a) \\
\mathit{join}\ (\mathsf{Single}\ x, \mathsf{Single}\ y) = \mathsf{Single}\ (x, y) \\
\mathit{join}\ (\mathsf{Fork}\ l, \mathsf{Fork}\ r) \qquad = \mathsf{Fork}\ (\mathit{join}\ (l, r))
\end{array}
$$

$$
\begin{array}{l}
\mathit{split} :: \mathsf{Bin}\ (a, a) \to (\mathsf{Bin}\ a, \mathsf{Bin}\ a) \\
\mathit{split}\ (\mathsf{Single}\ (x, y)) = (\mathsf{Single}\ x, \mathsf{Single}\ y) \\
\mathit{split}\ (\mathsf{Fork}\ t) \qquad\quad = (\mathsf{Fork}\ l,\ \mathsf{Fork}\ r) \\
\qquad \textbf{where}\ (l, r) = \mathit{split}\ t
\end{array}
$$

**Fig. 2.** A FunctorOf instance for perfect binary trees

By defining an instance of the FunctorOf class for Bin (see Fig. 2) 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.

$$
\begin{array}{l}
\mathit{full} :: a \to \mathsf{Int} \to \mathsf{Bin}\ a \\
\mathit{full}\ x = \mathit{ana}\ (\mathit{step}\ x) \\
\qquad \textbf{where}\ \mathit{step}\ x\ 0 \qquad\quad\ = \mathsf{InL}\ (\mathsf{Par}\ x) \\
\qquad\qquad\quad\ \mathit{step}\ x\ (n+1) = \mathsf{InR}\ (\mathsf{Rec}\ n)\ (\mathsf{Rec}\ n)
\end{array}
$$

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 didn't generate a full tree, we would get a run-time error (pattern match failure in *join*) instead of a type error.

### 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

$$
\begin{array}{l}
\mathit{push} \ \ :: a \to \mathsf{Stack}\ a \to \mathsf{Stack}\ a \\
\mathit{pop} \quad\ :: \mathsf{Stack}\ a \to \mathsf{Maybe}\ (a,\ \mathsf{Stack}\ a) \\
\mathit{empty} :: \mathsf{Stack}\ a
\end{array}
$$

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

$$
\textbf{instance}\ \mathsf{FunctorOf}\ (\mathsf{Empty}\ {:}{+}{:}\ \mathsf{Par}\ {:}{*}{:}\ \mathsf{Rec})\ \mathsf{Stack}\ \textbf{where}
$$

$$
\begin{aligned}
&\mathit{inn}\ (\mathsf{InL\ Empty}) &&= \mathit{empty}\\
&\mathit{inn}\ (\mathsf{InR}\ (\mathsf{Par}\ x \mathrel{:*:} \mathsf{Rec}\ s)) &&= \mathit{push}\ x\ s
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{out}\ s = \mathbf{case}\ \mathit{pop}\ s\ \mathbf{of}\\
&\quad \mathsf{Nothing} \quad\ \ \to \mathsf{InL\ Empty}\\
&\quad \mathsf{Just}\ (x, s') \to \mathsf{InR}\ (\mathsf{Par}\ x \mathrel{:*:} \mathsf{Rec}\ s')
\end{aligned}
$$

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, 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*

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

that converts between two regular datatypes with the same pattern functor. For instance we could convert a perfect binary tree from section 2.4 to a normal binary tree or convert a list to an element of the abstract stack type from section 2.5.

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

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

This requires Haskell extensions (available in ghc and hugs) for overlapping and undecidable instances, in addition to the multi-parameter type classes.

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 a 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.

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

The function *pzipWith* is the polytypic version of the Haskell prelude function *zipWith* and has type $\ldots \Rightarrow (a \to b \to c) \to d\ a \to d\ b \to d\ c$. If the structures of the arguments to *pzipWith* differ the function fails. Using *listTranspose* we can define *transpose* as follows:

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

The idea is to separate the structure and the contents of the argument to *transpose* using the function $\mathit{separate} :: \ldots \Rightarrow d\ a \to (d\ (),\ [a])$. The unstructured representation is then transposed using *listTranspose* and the structure is re-applied using $\mathit{combine} :: \ldots \Rightarrow d\ () \to [a] \to d\ a$. Again *combine* might fail if the length of the list doesn't match the number of holes in the structure. It is easy to modify *transpose* to use the Maybe monad to catch the potential failures.

# 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 [8]. 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 discuss how the PolyP definitions are compiled into Haskell.

## 3.1 The polytypic construct

In section 2.1 we introduced the pattern functor $\Phi_d$ 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.

To facilitate the definition of polytypic functions we define a few useful functions to manipulate the pattern functors.

$$
\begin{aligned}
&(f \triangledown g)\ (\mathsf{InL}\ x) = f\ x\\
&(f \triangledown g)\ (\mathsf{InR}\ y) = g\ y\\
&f \mathbin{-\!\!+\!\!-} g \qquad\quad\ = (\mathsf{InL} \circ f) \triangledown (\mathsf{InR} \circ g)
\end{aligned}
$$

The operators $(\triangledown)$ and $(\mathbin{-\!\!+\!\!-})$ are the elimination and map functions for sums. 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, in Fig. 3 we define the function *fsum* from section 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

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

**Fig. 3.** Defining *fsum* using the **polytypic** construct

*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.

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

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

where $f$ is the pattern functor (occurring somewhere in $\tau$) and $\varphi_i$ 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 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 function *psum* which can be defined as the catamorphism of *fsum*:

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

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

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.

### 3.2 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 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 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 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

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

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 $\Phi_D$ is

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

where we represent a nullary product by $\mathsf{Empty}$. When defining the functions $\mathit{inn}$ and $\mathit{out}$ 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 $\mathit{to}_g$ and $\mathit{from}_g$ such that

$$
\begin{aligned}
&\mathit{to}_g &&:: \widehat{g}\ p\ r \to g\ p\ r && \text{— } \mathit{to}_g \circ \mathit{from}_g = \mathit{id} \\
&\mathit{from}_g &&:: g\ p\ r \to \widehat{g}\ p\ r && \text{— } \mathit{from}_g \circ \mathit{to}_g = \mathit{id}
\end{aligned}
$$

For the pattern functors $\mathsf{Par}$, $\mathsf{Rec}$ and $\mathsf{Const}$, $\mathit{to}$ and $\mathit{from}$ 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.

| | |
|---|---|
| $\mathit{to}_{\mathsf{Par}} = \mathsf{Par}$ | $\mathit{from}_{\mathsf{Par}} = \mathit{unPar}$ |
| $\mathit{to}_{\mathsf{Rec}} = \mathsf{Rec}$ | $\mathit{from}_{\mathsf{Rec}} = \mathit{unRec}$ |
| $\mathit{to}_{\mathsf{Const}\ t} = \mathsf{Const}$ | $\mathit{from}_{\mathsf{Const}\ t} = \mathit{unConst}$ |
| $\mathit{to}_{d\ @\ g} = \mathsf{Comp} \circ\ \mathit{pmap}\ \mathit{to}_g$ | $\mathit{from}_{d\ @\ g} = \mathit{pmap}\ \mathit{from}_g \circ\ \mathit{unComp}$ |

Now define $\iota^n_m$ to be the sequence of $\mathsf{InL}$ and $\mathsf{InR}$'s corresponding to the $m^{\text{th}}$ 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 is $\iota^3_2\ x \;=\; \mathsf{InR}\ (\mathsf{InL}\ x)$.

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

$$
\begin{aligned}
&\mathbf{instance}\ \mathsf{FunctorOf}\ \Phi_D\ D\ \mathbf{where} \\
&\quad \mathit{inn}\ (\iota^n_k\ (x_1 :\!*\!: \ldots :\!*\!: x_{m_k})) = C_k\ (\mathit{to}_{g_{k1}}\ x_1)\ \ldots\ (\mathit{to}_{g_{km_k}}\ x_{m_k}) \\
&\quad \mathit{out}\ (C_k\ x_1\ \ldots\ x_{m_k}) \qquad\quad = \iota^n_k\ (\mathit{from}_{g_{k1}}\ x_1 :\!*\!: \ldots :\!*\!: \mathit{from}_{g_{km_k}}\ x_{m_k})
\end{aligned}
$$

### 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

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

**Fig. 4.** Translation of a polytypic construct to a class and instances

formally, given a polytypic function definition like the left side in Fig. 4 the translation produces the result on the right.

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 $\rho_i$. For example, in the definition of *fsum* from section 3.1, the instance $\mathsf{P\_fsum}\,(g :\!+\!: h)$ needs instances of $\mathsf{P\_fsum}$ 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. 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 function *pmap* is

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

which is a little too general to be practical. For instance, in the expression $psum\;(pmap\;(1+)\;[1,2,3])$, the compiler wouldn't 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 $\Phi_d$ 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:

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

Here $e$ would be identified with $d$ and $\rho$ 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 Fig. 3 as an example. This definition is translated to a class and instance declarations for each branch:

$$
\begin{array}{llll}
\textbf{class} & \mathsf{P\_fsum}\ f & \textbf{where}\ \mathit{fsum} & ::\ \mathsf{Num}\ a \Rightarrow f\ a\ a \rightarrow a \\
\textbf{instance}\ \rho_{+} \Rightarrow & \mathsf{P\_fsum}\ (g :+: h) & \textbf{where}\ \mathit{fsum} & =\ \mathit{fsum} \bigtriangledown \mathit{fsum} \\
\vdots & & &
\end{array}
$$

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

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

This type is then unified with the type of *fsum* extended with the constraint set variable $\rho_{+}$ serving as a place holder for the extra class constraints:

$$(\mathsf{Num}\ a, \rho_{+}) \Rightarrow f\ a\ a \rightarrow a$$

In this case the result of the unification would be

$$
\begin{array}{ll}
f & \mapsto g :+: h \\
\rho_{+} & \mapsto \{\mathsf{P\_fsum}\ g, \mathsf{P\_fsum}\ h\}
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{instance}\ (\mathsf{P\_fsum}\ g,\ \mathsf{P\_fsum}\ h) \Rightarrow \mathsf{P\_fsum}\ (g :+: h)\ \textbf{where} \\
\quad \mathit{fsum}\ =\ \mathit{fsum} \bigtriangledown \mathit{fsum}
\end{array}
$$

### 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), because 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** $fsum$ :: ... = $\langle$definition using $psum\rangle$
$psum = cata\ fsum$

we would generate a Haskell module looking like this:

**module** Sum ($psum$, P_fmap2, P_fsum) **where**
**import** Map ($pmap$, P_fmap2)
$\vdots$

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 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 be possible to add.

## 4 Discussion

One of the benefits of using the class system is that we do not need to rely on a polytypic compiler to the same extent as when using a specializing approach. To make this more precise we identify a few disjoint sublanguages within a polytypic language:

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

Using a specializing compiler translating into Base we have to compile at least PolyCore, PolyUse, PolyInst and Regular. With the new PolyP we only need to compile PolyCore (and may choose to 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.

## 5 Related work

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

– The old PolyP [8] 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 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 the old version, while in the new version it is merely a minor inconvenience.
– Generic Haskell [2, 5] 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 all all 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 access to the recursive calls 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 only allows definitions of polytypic functions 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 [6] 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 $\mathsf{FunctorOf}$-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].
– DrIFT preprocessor for deriving non-standard Haskell classes has been used together with the Strafunski library [13, 14] 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 [12] 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 [16] 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.

Other implementations of functional polytypism include Charity [3], FISh [11] and G'Caml [4] but in this paper we focus on the Haskell-based languages.

## 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 [9] as a Haskell library as well as define new polytypic functions such as *coerce* that converts between two datatypes of the same shape 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 [17] to internalize the PolyP compiler as a ghc extension. Other research directions are to extend our approach to more datatypes (partially explored in [15]), or to explore in more detail which polytypic functions are expressible in this setting.

## 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. 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.
3. R. Cockett and T. Fukushima. About Charity. Yellow Series Report No. 92/480/18, Dep. of Computer Science, Univ. of Calgary, 1992.
4. J. Furuse. Generic polymorphism in ML. In *Journées Francophones des Langages Applicatifs*, 2001.
5. 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.
6. 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.
7. P. Jansson. The WWW home page for polytypic programming. Available from `http://www.cs.chalmers.se/~patrikj/poly/`, 2003.
8. P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *POPL'97*, pages 470–482. ACM Press, 1997.
9. P. Jansson and J. Jeuring. PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand, June 1998. Available from the Polytypic programming WWW page [7].
10. 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.
11. 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.
12. 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.
13. R. Lämmel and J. Visser. Strategic polymorphism requires just two combinators! Technical Report cs.PL/0212048, arXiv, Dec. 2002.
14. 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.
15. 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`.
16. T. Sheard. Generic unification via Two-Level types and parameterized modules. In *ICFP'01*, pages 86–97, 2001.
17. T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proceedings of the Haskell workshop*, pages 1–16. ACM Press, 2002.