Polytypic Programming With Ease (Extended Abstract)

Ralf Hinze

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
ralf@informatik.uni-bonn.de
http://www.informatik.uni-bonn.de/~ralf/

Abstract. This paper proposes a new framework for a polytypic extension of functional programming languages. A functional polytypic program is one that is parameterised by datatype. Since polytypic functions are defined by induction on types rather than by induction on values, they typically operate on a higher level of abstraction than their monotypic counterparts. However, polytypic programming is not necessarily more complicated than conventional programming. In fact, a polytypic function is uniquely defined by its action on constant functors, projection functors, sums, and products. This information is sufficient to specialize a polytypic function to arbitrary datatypes, including mutually recursive datatypes and nested datatypes. The key idea is to use infinite trees as index sets for polytypic functions and to interpret datatypes as algebraic trees. This approach is simpler, more general, and more efficient than previous ones that are based on the initial algebra semantics of datatypes.

1 Introduction

This paper proposes a new framework for a polytypic extension of functional programming languages such as Haskell or Standard ML. A polytypic function is one that is defined by induction on the structure of types. The archetypical example of a polytypic function is $size :: f\ a \to Int$, which counts the number of values of type a in a given value of type $f\ a$. The function *size* can sensibly be defined for each parameterised datatype and it is often—but not always—a tiresomely routine matter to do so. A polytypic programming language enables the user to program *size* once and for all times. The specialization of *size* to concrete instances of f is then handled automatically by the system. Polytypic programs are ubiquitous: typical examples include equality and comparison functions, mapping functions, pretty printers (such as Haskell's *show* function), parsers (such as Haskell's *read* function), data compression [9], and digital searching [6]. The ability to define such programs generically for all datatypes greatly simplifies the construction and maintenance of software systems.

Since polytypic functions are defined by induction on types rather than by induction on values, they tend to be more abstract than their monotypic counterparts. However, once a certain familiarity has been gained, it turns out that

polytypic programming is actually simpler than conventional programming. To support this claim let us define two simple functions, *size* and *sum*, for some illustrative datatypes. As a first example, consider the datatype of rose trees.[1]

$$\begin{aligned}&\textbf{data}\ \mathit{Rose}\ a = \mathit{Branch}\ a\ (\mathit{List}\ (\mathit{Rose}\ a))\\&\textbf{data}\ \mathit{List}\ a\ \ = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)\end{aligned}$$

The size of a rose tree can be determined as follows [2].

$$\begin{aligned}&\mathit{sizer} && :: \mathit{Rose}\ a \to \mathit{Int}\\&\mathit{sizer}\ (\mathit{Branch}\ a\ ts) &&= 1 + \mathit{sum}\ (\mathit{map}\ \mathit{sizer}\ ts)\end{aligned}$$

This definition is already quite sophisticated: it makes use of a general combining form and an auxiliary function. The use of $\mathit{map} :: (a \to b) \to (\mathit{List}\ a \to \mathit{List}\ b)$, however, also incurs a slight run-time penalty: *map* produces an intermediate list, which is immediately consumed by *sum*. This inefficiency can be eliminated by fusing $\mathit{sum} \circ \mathit{map}\ \mathit{sizer}$ into a single function (called *sizef* below).

$$\begin{aligned}&\mathit{sizer} && :: \mathit{Rose}\ a \to \mathit{Int}\\&\mathit{sizer}\ (\mathit{Branch}\ a\ ts) &&= 1 + \mathit{sizef}\ ts\\&\mathit{sizef} && :: \mathit{List}\ (\mathit{Rose}\ a) \to \mathit{Int}\\&\mathit{sizef}\ \mathit{Nil} &&= 0\\&\mathit{sizef}\ (\mathit{Cons}\ t\ ts) &&= \mathit{sizer}\ t + \mathit{sizef}\ ts\end{aligned}$$

Interestingly, this very definition naturally arises if we change the definition of rose trees to

$$\begin{aligned}&\textbf{data}\ \mathit{Rose}'\ a\ \ = \mathit{Branch}\ a\ (\mathit{Forest}\ a)\\&\textbf{data}\ \mathit{Forest}\ a = \mathit{Nil} \mid \mathit{Cons}\ (\mathit{Rose}'\ a)\ (\mathit{Forest}\ a)\ .\end{aligned}$$

The definition of *sum* proceeds completely analogously—it suffices, in fact, to replace '1' by 'a' in the definitions above.

A slightly more complex datatype is the type of perfect binary search trees.

$$\begin{aligned}&\textbf{data}\ \mathit{Perfect}\ a\ k = \mathit{Zero}\ a \mid \mathit{Succ}\ (\mathit{Perfect}\ (\mathit{Node}\ a\ k)\ k)\\&\textbf{data}\ \mathit{Node}\ a\ k\ \ \ = \mathit{Node}\ a\ k\ a\end{aligned}$$

The definition of *Perfect* is somewhat unusual in that the recursive component, $\mathit{Perfect}\ (\mathit{Node}\ a\ k)\ k$, is not identical to the left-hand side of the equation: *Perfect* is an example of a so-called *nested datatype*, a term coined by Bird and Meertens [4]. Since *Perfect* only encompasses perfect binary search trees, the size of a tree of type $\mathit{Perfect}\ a\ a$ can be computed in logarithmic time.

$$\begin{aligned}&\mathit{sizep} && :: \mathit{Perfect}\ a\ k \to \mathit{Int}\\&\mathit{sizep}\ (\mathit{Zero}\ a) &&= 1\\&\mathit{sizep}\ (\mathit{Succ}\ t) &&= 2 * \mathit{sizep}\ t + 1\end{aligned}$$

[1] Examples are given in the functional programming language Haskell 98 [21].

The function *sizep* counts the number of values of type a in a tree of type *Perfect a a*. However, *sizep* cannot be assigned the type *Perfect a a* $\to$ *Int* since the recursive call has type *Perfect* (*Node a k*) *k* $\to$ *Int*.

Summing up a perfect tree of integers is a bit more challenging.

$$\begin{array}{ll} sump & :: Perfect\ Int\ Int \to Int \\ sump\ (Zero\ a) & = a \\ sump\ (Succ\ t) & = sump\ (perfect\ sumn\ id\ t) \\ sumn & :: Node\ Int\ Int \to Int \\ sumn\ (Node\ l\ k\ r) & = l + k + r \end{array}$$

Here, $perfect :: (a \to a') \to (k \to k') \to (Perfect\ a\ k \to Perfect\ a'\ k')$ denotes the mapping function of the binary functor *Perfect*. Improving the efficiency of *sump* by fusing $sump \circ perfect\ sumn\ id$ is left as an exercise to the reader.

Now, let us define *size* and *sum* once and for all times. To this end we must first take a closer look at the structure of types. Reconsider the datatype definitions given above. Haskell's **data** construct combines several features in a single coherent form: sums, products, and recursion. The structure of the types becomes more apparent if the definitions are rewritten as *functor equations*:

$$\begin{array}{ll} Rose & = Id \times List \cdot Rose \\ List & = K1 + Id \times List \\ Perfect & = Fst + Perfect \cdot (Node, Snd) \\ Node & = Fst \times Snd \times Fst\ , \end{array}$$

where $K\,T$ is the constant functor, *Id* is the identity functor, *Fst* and *Snd* are projection functors, and $F \cdot (F_1, \ldots, F_n)$ denotes the composition of an n-ary functor F with n functors, all of the same arity. Sum and product are defined pointwise: $(F_1 + F_2)\ a = F_1\ a + F_2\ a$ and $(F_1 \times F_2)\ a = F_1\ a \times F_2\ a$. In essence, functor equations are written in a compositional or 'point-free' style while **data** definitions are written in an applicative or 'pointwise' style. We treat 1, '+', and '$\times$' as if they were given by the following datatype declarations.

$$\begin{array}{ll} \textbf{data}\ 1 & = () \\ \textbf{data}\ a_1 + a_2 & = Inl\ a_1 \mid Inr\ a_2 \\ \textbf{data}\ a_1 \times a_2 & = (a_1, a_2) \end{array}$$

To define *size* it suffices to specify its action on constant functors, on the identity functor, on sums, and on products. Polytypic functions are written using angle brackets to distinguish them from ordinary functions.

$$\begin{array}{ll} size\langle f \rangle & :: \forall a.f\ a \to Int \\ size\langle Id \rangle\ x & = 1 \\ size\langle K\,t \rangle\ x & = 0 \\ size\langle f_1 + f_2 \rangle\ (Inl\ x_1) & = size\langle f_1 \rangle\ x_1 \\ size\langle f_1 + f_2 \rangle\ (Inr\ x_2) & = size\langle f_2 \rangle\ x_2 \\ size\langle f_1 \times f_2 \rangle\ (x_1, x_2) & = size\langle f_1 \rangle\ x_1 + size\langle f_2 \rangle\ x_2 \end{array}$$

Each equation is more or less inevitable: a value of type $Id\ a = a$ contains one element of type a; a value of type $K\,t\ a = t$ contains no elements. To determine the size of an element of type $f_1\ a + f_2\ a$ we must either calculate the size of a structure of type $f_1\ a$ or that of a structure of type $f_2\ a$. The size of a structure of type $f_1\ a \times f_2\ a$ is given by the sum of the size of the two components. From this definition the following specializations can be automatically derived.

$$
\begin{array}{ll}
\mathit{sizeRose} & = \mathit{sizeRose}_1\ (\mathit{const}\ 1) \\
\mathit{sizeRose}_1\ \varphi\ (\mathit{Branch}\ a\ ts) & = \varphi\ a + \mathit{sizeList}_1\ (\mathit{sizeRose}_1\ \varphi)\ ts \\
\mathit{sizeList}_1\ \varphi\ \mathit{Nil} & = 0 \\
\mathit{sizeList}_1\ \varphi\ (\mathit{Cons}\ a\ as) & = \varphi\ a + \mathit{sizeList}_1\ \varphi\ as \\
\mathit{sizePerfect} & = \mathit{sizePerfect}_2\ (\mathit{const}\ 1, \mathit{const}\ 1) \\
\mathit{sizePerfect}_2\ (\varphi_1, \varphi_2)\ (\mathit{Zero}\ a) & = \varphi_1\ a \\
\mathit{sizePerfect}_2\ (\varphi_1, \varphi_2)\ (\mathit{Succ}\ t) & = \mathit{sizePerfect}_2\ (\mathit{sizeNode}_2\ (\varphi_1, \varphi_2), \varphi_2)\ t \\
\mathit{sizeNode}_2\ (\varphi_1, \varphi_2)\ (\mathit{Node}\ l\ k\ r) & = \varphi_1\ l + \varphi_2\ k + \varphi_1\ r
\end{array}
$$

We postpone a detailed explanation of the specialization process until Section 3. For the moment it suffices to note that the different instances rigidly follow the structure of the respective datatypes. How do these functions relate to the handcrafted definitions? Now, *sizeRose* corresponds to the second, more efficient definition of *sizer*: we have $\mathit{sizer} = \mathit{sizeRose}_1\ (\mathit{const}\ 1)$ and $\mathit{sizef} = \mathit{sizeList}_1\ \mathit{sizer}$. On the negative side, *sizePerfect* is a linear-time implementation as opposed to the logarithmic *sizep*. In general, a 'structure-strict', polytypic function has at least a linear running time. So we cannot reasonably expect to achieve the efficiency of a handcrafted implementation that exploits data-structural invariants.

The polytypic definition of *sum* is equally simple.

$$
\begin{array}{ll}
\mathit{sum}\langle f \rangle & :: f\ \mathit{Int} \to \mathit{Int} \\
\mathit{sum}\langle \mathit{Id} \rangle\ x & = x \\
\mathit{sum}\langle K\,t \rangle\ x & = 0 \\
\mathit{sum}\langle f_1 + f_2 \rangle\ (\mathit{Inl}\ x_1) & = \mathit{sum}\langle f_1 \rangle\ x_1 \\
\mathit{sum}\langle f_1 + f_2 \rangle\ (\mathit{Inr}\ x_2) & = \mathit{sum}\langle f_2 \rangle\ x_2 \\
\mathit{sum}\langle f_1 \times f_2 \rangle\ (x_1, x_2) & = \mathit{sum}\langle f_1 \rangle\ x_1 + \mathit{sum}\langle f_2 \rangle\ x_2
\end{array}
$$

Specializing $\mathit{sum}\langle f \rangle$ to the various datatypes yields definitions similar to those obtained for $\mathit{size}\langle f \rangle$. In this case the generated functions are as efficient as the best handcoded implementations.

The moral of the story so far: giving ad-hoc definitions of polytypic functions like *size* and *sum* is sometimes simple and sometimes involving. While the polytypic definition is slightly more abstract, it is also to a high degree inevitable. It is this feature that makes polytypic programming light and sweet.

The rest of this paper is organized as follows. Section 2 introduces the basic ingredients of polytypic programming: functor equations and algebraic trees. Section 3 explains how to specialize a polytypic function to concrete instances of datatypes. Section 4 presents several examples of polytypic functions, among others polytypic reduction and polytypic equality. Finally, Section 5 reviews related work and points out directions for future work.

2 Datatypes as Algebraic Trees

In the introduction we have seen that functor equations capture the essence of datatype definitions. In general, a system of functor equations has the form $f_1 = e_1; \ldots; f_p = e_p$ where the f_i are functor variables and the e_i are functor expressions. Given a ranked set of primitive functors, $\Sigma = \Sigma^0 \cup \Sigma^2$ with $\Sigma^0 = \{1, \mathit{Int}\}$ and $\Sigma^2 = \{+, \times\}$, the set of *functor expressions of arity* n is inductively defined as follows.

$$F^n ::= \Pi_i^n \mid \Sigma^n \mid F^k \cdot (F_1^n, \ldots, F_k^n)$$

Here Π_i^n denotes the n-ary projection functor selecting the i-th component. For unary and binary projection functors we use the following more familiar names: $\mathit{Id} = \Pi_1^1$, $\mathit{Fst} = \Pi_1^2$, and $\mathit{Snd} = \Pi_2^2$. The expression $F \cdot (F_1, \ldots, F_k)$ denotes the composition of a k-ary functor F with functors F_i, all of arity n. We omit the parentheses when $k = 1$ and we write $K\,T$ instead of $T \cdot ()$ when $k = 0$. Finally, we write $F_1 \oplus F_2$ instead of $\oplus \cdot (F_1, F_2)$ for $\oplus \in \Sigma^2$.

Each functor equation defines a unique infinite tree, whose inner nodes are labelled with primitive functors of arity $\geqslant 1$ and whose leaves are decorated with nullary functors and projection functors. Figure 1 displays the trees defined by the equations for *Rose* and *Perfect* given in the previous section. Note that

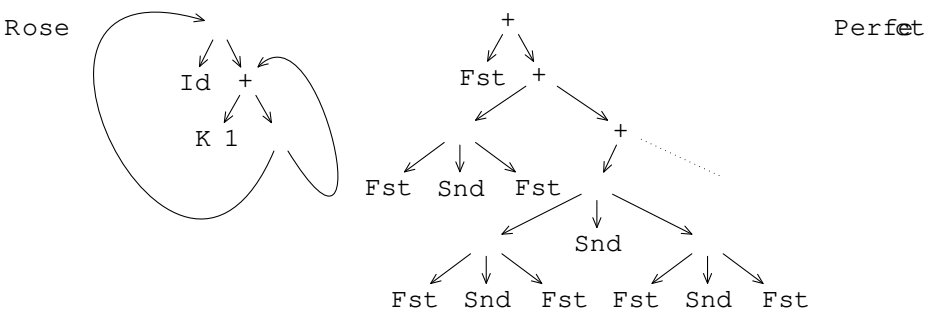

Fig. 1. Types interpreted as infinite trees

Rose is interpreted by a *rational tree* while *Perfect* denotes an *algebraic tree*. A rational tree is a possibly infinite tree that has only a finite number of subtrees. Algebraic trees are obtained as solutions of so-called algebraic equations, which are akin to functor equations. We will view infinite trees naively as a kind of 'infinite normal form' of functor expressions. Expressions in normal form are formed according to the following grammar.

$$T^n ::= \Pi_i^n \mid \Sigma^k \cdot (T_1^n, \ldots, T_k^n)$$

Thus, $P \cdot (F_1, \ldots, F_k)$ with $P \in \Sigma^k$ corresponds to a tree, whose root is labelled with P and which has subtrees $F_1, \ldots, F_k$, and Π_i^n corresponds to a leaf. For a

more formal treatment the interested reader is referred to the full version of this paper [8]. Courcelle's article [5] provides a more detailed survey on the theory of infinite trees. Briefly, the set of infinite trees constitutes an initial continuous algebra. Systems of functor equations have least solutions in this complete partial order.

Roughly speaking, the infinite tree defined by a functor equation is obtained by unrolling the equation ad infinitum applying the following functorial laws.

$$F \cdot (\Pi_1, \ldots, \Pi_n) = F \tag{1}$$
$$\Pi_i \cdot (F_1, \ldots, F_n) = F_i \tag{2}$$
$$(F \cdot (G_1, \ldots, G_n)) \cdot \mathbf{H} = F \cdot (G_1 \cdot \mathbf{H}, \ldots, G_n \cdot \mathbf{H}) \tag{3}$$

In the latter equation $\mathbf{H}$ is a tuple of functor expressions. Functor composition can be seen as a substitution operation on trees. Eq. (2) defines the substitution of a leaf by a tree; Eq. (3) formalizes the propagation of a substitution to the subtrees of a node. As an example, consider the unfolding of *Perfect*:

$$\begin{aligned}
&\mathit{Perfect}\\
&= \mathit{Perfect}_0 + \mathit{Perfect} \cdot (\mathit{Perfect}_1, \mathit{Snd})\\
&= \mathit{Perfect}_0 + (\mathit{Fst} + \mathit{Perfect} \cdot (\mathit{Node}, \mathit{Snd})) \cdot (\mathit{Perfect}_1, \mathit{Snd})\\
&= \mathit{Perfect}_0 + (\mathit{Perfect}_1 + \mathit{Perfect} \cdot (\mathit{Perfect}_2, \mathit{Snd}))\\
&= \mathit{Perfect}_0 + (\mathit{Perfect}_1 + (\mathit{Fst} + \mathit{Perfect} \cdot (\mathit{Node}, \mathit{Snd})) \cdot (\mathit{Perfect}_2, \mathit{Snd}))\\
&= \mathit{Perfect}_0 + (\mathit{Perfect}_1 + (\mathit{Perfect}_2 + \mathit{Perfect} \cdot (\mathit{Perfect}_3, \mathit{Snd})))\\
&= \ldots\ ,
\end{aligned}$$

where $\mathit{Perfect}_0 = \mathit{Fst}$ and $\mathit{Perfect}_{n+1} = \mathit{Perfect}_n \times \mathit{Snd} \times \mathit{Perfect}_n$. The unfolding shows that *Perfect* is the disjoint union of perfectly balanced trees of arbitrary height.

The major difference to previous approaches to polytypic programming [10] lies in the use of infinite instead of finite trees as index sets for polytypic functions. In essence, this means that the class of all datatypes is itself modeled by a *non-strict* datatype as opposed to an inductive datatype. This change is not as problematic as one might think at first sight. Polytypic functions can still be recursively defined. Of course, to make things work the functions involved must be continuous. This condition is, however, trivially satisfied since the functions are programs in Haskell or Standard ML.

Now, let us consider some examples of functors. A functor is called *polynomial* if it is represented by a finite tree.

$$\begin{aligned}
\Delta &= \mathit{Id} \times \mathit{Id}\\
\mathit{Node23} &= \mathit{Fst} \times \mathit{Snd} \times \mathit{Fst} + \mathit{Fst} \times \mathit{Snd} \times \mathit{Fst} \times \mathit{Snd} \times \mathit{Fst}
\end{aligned}$$

The functor Δ is called the diagonal or square functor; *Node23* is used below in the definition of 2-3 trees.

A *regular* functor is one that is given by a rational tree.

$$\mathit{Bintree} = \mathit{Fst} + \mathit{Bintree} \times \mathit{Snd} \times \mathit{Bintree}$$

The functor *Bintree* models external, binary search trees. Further examples of regular functors are *Rose*, *List*, *Rose'*, and *Forest*.

The functor *Perfect* is an example of a *non-regular* or *nested* functor, which is given by an algebraic tree.

$$\begin{aligned} \mathit{Sequ} &= K1 + \mathit{Sequ} \cdot \Delta + \mathit{Id} \times \mathit{Sequ} \cdot \Delta \\ \mathit{Tree23} &= \mathit{Fst} + \mathit{Tree23} \cdot (\mathit{Node23}, \mathit{Snd}) \end{aligned}$$

Okasaki [19] uses the functor *Sequ* as the basis for a sequence implementation with an efficient indexing operation. The functor *Tree23* models 2-3 trees. The novelty of this definition is that the data-structural invariants of 2-3 trees—each interior node has two or three children and each path from the root to a leaf has the same length—are made manifest.

The same functor can usually be defined in a variety of ways. Take, for instance, the two definitions of rose trees given in Section 1.

$$\begin{aligned} \mathit{Rose} &= \mathit{Id} \times \mathit{List} \cdot \mathit{Rose} & \qquad \mathit{Rose}' &= \mathit{Id} \times \mathit{Forest} \\ \mathit{List} &= K1 + \mathit{Id} \times \mathit{List} & \qquad \mathit{Forest} &= K1 + \mathit{Rose}' \times \mathit{Forest} \end{aligned}$$

Both *Rose* and *Rose'* denote the same rational tree depicted in Figure 1.

3 Polytypic Functions

We have already seen two examples of polytypic functions. In general, a polytypic function that is parameterised by m-ary functors is given by the following data (think of the following as a prototype for polytypic definitions).

$$\begin{aligned} &\mathit{poly}\langle f \rangle && :: \tau(f) \\ &\mathit{poly}\langle \Pi_i \rangle && = \mathit{poly}_{\Pi_i} \\ &\mathit{poly}\langle P \cdot (f_1, \ldots, f_k) \rangle && = \mathit{poly}_P \ (\mathit{poly}\langle f_1 \rangle, \ldots, \mathit{poly}\langle f_k \rangle) \end{aligned}$$

The type of $\mathit{poly}\langle f \rangle$ is specified by the type scheme $\tau(f)$. Since f is an m-ary functor, poly_{Π_i} must be specified for $1 \leqslant i \leqslant m$. Furthermore, an equation specifying poly_P must be given for each primitive functor $P \in \Sigma$. This information is sufficient to define a unique function $\mathit{poly}\langle f \rangle$ for each functor f [5]. Briefly, an inductively defined function like $\mathit{poly}\langle f \rangle$ can be uniquely extended to a continuous function on infinite trees.

The question we pursue in this section is how to derive specializations of $\mathit{poly}\langle f \rangle$ for given instances of f. The process of specialization is necessary since $\mathit{poly}\langle f \rangle$ cannot directly be implemented in languages such as Haskell or Standard ML. The reason is simply that the type of *poly* depends on the first argument, which is itself a type (or possibly, the encoding of a type). Even if we circumvented this problem by using encodings into a universal datatype [22] or by using dynamic types and a **typecase** [1], the result would be rather inefficient because *poly* would interpret its type argument at each stage of the recursion.

By specializing $poly\langle f\rangle$ for a given f we remove this interpretative layer. Thus, we can view the following as a very special instance of partial evaluation.

Since types correspond to algebraic trees, which may have an infinite number of different subtrees, the specialization cannot be based on the structure of types. However, using functor equations algebraic trees can be finitely represented so the idea suggests itself to carry out the specialization on the representation of trees. The main challenge lies in the treatment of functor composition. Assume, for instance, that an unary functor is defined in terms of a binary functor: $F = B \cdot (e_1, e_2)$. Now, if we want to specialize $size\langle F\rangle$, how can we generate code for the right-hand side? The central idea is to define an auxiliary function $size_2\langle B\rangle$ that satisfies $size_2\langle B\rangle\ (size\langle e_1\rangle, size\langle e_2\rangle) = size\langle B \cdot (e_1, e_2)\rangle$ for all e_1 and e_2. That is, $size_2\langle B\rangle$ takes $size\langle e_1\rangle$ and $size\langle e_2\rangle$ as arguments and implements $size\langle B \cdot (e_1, e_2)\rangle$. We have seen that functor composition corresponds to a substitution operation on trees. Using the function $size_2\langle B\rangle$ we imitate substitution on the function level. In general, we define, for each arity n, a polytypic function $poly_n\langle g\rangle :: (\tau(g_1), \ldots, \tau(g_n)) \to \tau(g \cdot (g_1, \ldots, g_n))$ satisfying

$$poly_n\langle g\rangle\ (poly\langle g_1\rangle, \ldots, poly\langle g_n\rangle) = poly\langle g \cdot (g_1, \ldots, g_n)\rangle\ . \tag{4}$$

Note the close correspondence between types and functions: g is an n-ary functor sending $g_1, \ldots, g_n$ to $g \cdot (g_1, \ldots, g_n)$; likewise $poly_n\langle g\rangle$ is an n-ary function sending $poly\langle g_1\rangle, \ldots, poly\langle g_n\rangle$ to $poly\langle g \cdot (g_1, \ldots, g_n)\rangle$.

For primitive functors $P \in \Sigma^n$ the function $poly_n\langle P\rangle$ is defined by $poly_n\langle P\rangle = poly_P$. The derivation of $poly_n\langle g\rangle$ for projection functors and for functor composition is a nice example in program calculation and proceeds almost mechanically. We assume that $\varphi = (poly\langle g_1\rangle, \ldots, poly\langle g_n\rangle)$ and reason:

$$\begin{aligned} poly_n\langle \Pi_i\rangle\ \varphi &= poly\langle \Pi_i \cdot (g_1, \ldots, g_n)\rangle && (4)\\ &= poly\langle g_i\rangle\ . && (2) \end{aligned}$$

Generalizing $(poly\langle g_1\rangle, \ldots, poly\langle g_n\rangle)$ to the tuple $(\varphi_1, \ldots, \varphi_n)$ we have calculated $poly_n\langle \Pi_i\rangle\ (\varphi_1, \ldots, \varphi_n) = \varphi_i$. For functor composition we obtain

$$\begin{aligned} &poly_n\langle h \cdot (h_1, \ldots, h_k)\rangle\ \varphi\\ &= poly\langle (h \cdot (h_1, \ldots, h_k)) \cdot (g_1, \ldots, g_n)\rangle && (4)\\ &= poly\langle h \cdot (h_1 \cdot (g_1, \ldots, g_n), \ldots, h_k \cdot (g_1, \ldots, g_n))\rangle && (3)\\ &= poly_k\langle h\rangle\ (poly\langle h_1 \cdot (g_1, \ldots, g_n)\rangle, \ldots, poly\langle h_k \cdot (g_1, \ldots, g_n)\rangle) && (4)\\ &= poly_k\langle h\rangle\ (poly_n\langle h_1\rangle\ \varphi, \ldots, poly_n\langle h_k\rangle\ \varphi)\ . && (4) \end{aligned}$$

Thus, for each arity $n \geqslant 0$ the polytypic function $poly_n\langle g\rangle$ is given by

$$\begin{aligned} poly_n\langle g\rangle &:: (\tau(g_1), \ldots, \tau(g_n)) \to \tau(g \cdot (g_1, \ldots, g_n))\\ poly_n\langle \Pi_i\rangle &= \pi_i\\ poly_n\langle P\rangle &= poly_P\\ poly_n\langle h \cdot (h_1, \ldots, h_k)\rangle &= poly_k\langle h\rangle \star (poly_n\langle h_1\rangle, \ldots, poly_n\langle h_k\rangle)\ . \end{aligned}$$

where $\pi_i\,(\varphi_1,\ldots,\varphi_n) = \varphi_i$ is the i-th projection function and '$\star$' denotes n-ary composition defined by $(\varphi \star (\varphi_1,\ldots,\varphi_n))\;a = \varphi\,(\varphi_1\;a,\ldots,\varphi_n\;a)$. Note that the g_i are m-ary functors while the h_i have arity n. Furthermore, note that the definition of $poly_n\langle g\rangle$ is *inductive on the structure of functor expressions*. On a more abstract level we can view $poly_n$ as an interpretation of functor expressions: Π_i is interpreted by π_i, P by $poly_P$, and '$\cdot$' by '$\star$'.

It remains to define $poly\langle f\rangle$ in terms of $poly_m\langle f\rangle$:

$$\begin{aligned} poly\langle f\rangle &= poly\langle f \cdot (\Pi_1,\ldots,\Pi_m)\rangle && (1)\\ &= poly_m\langle f\rangle\;(poly\langle\Pi_1\rangle,\ldots,poly\langle\Pi_m\rangle) && (4)\\ &= poly_m\langle f\rangle\;(poly_{\Pi_1},\ldots,poly_{\Pi_m})\;. && \text{definition } poly \end{aligned}$$

By now we have the necessary prerequisites at hand to define the specialization of a polytypic function $poly\langle f\rangle$ for a given instance of f. The specialization works by transforming a system of functor equations that defines f into a system of function definitions that defines $poly_m\langle f\rangle$. In particular, recursion on the level of types is implemented by recursion on the function level.

$$\begin{array}{lcl} f_1 = e_1 & & poly_{k_1}\langle f_1\rangle = poly_{k_1}\langle e_1\rangle \\ \vdots & \Longrightarrow & \vdots \\ f_p = e_p & & poly_{k_p}\langle f_p\rangle = poly_{k_p}\langle e_p\rangle \end{array}$$

Here k_i is the arity of f_i. The expression $poly_{k_i}\langle e_i\rangle$ is given by the inductive definition above, additionally setting $poly_{k_i}\langle f_i\rangle = poly_f_i_k_i$, where $poly_f_i_k_i$ is a new function symbol. Additionally, the defining equation for $poly_f$, ie $poly_f = poly_f_m\;(poly_{\Pi_1},\ldots,poly_{\Pi_m})$, must be added.

As an example, consider the datatype *Perfect* of Section 1.

$$Perfect = Fst + Perfect \cdot (Node, Snd)$$

Note that $size_+ = (\triangledown)$, where $(\triangledown)$ is given by $(\varphi_1 \triangledown \varphi_2)\;(Inl\;x_1) = \varphi_1\;x_1$ and $(\varphi_1 \triangledown \varphi_2)\;(Inr\;x_2) = \varphi_2\;x_2$. Using this abbreviation the specialization of $size_2\langle Perfect\rangle$ proceeds as follows.

$$\begin{aligned} & size_2\langle Fst + Perfect \cdot (Node, Snd)\rangle\;(\varphi_1,\varphi_2)\\ = {} & size_2\langle +\rangle\;(size_2\langle Fst\rangle\;(\varphi_1,\varphi_2), size_2\langle Perfect \cdot (Node, Snd)\rangle\;(\varphi_1,\varphi_2))\\ = {} & size_2\langle Fst\rangle\;(\varphi_1,\varphi_2) \triangledown size_2\langle Perfect \cdot (Node, Snd)\rangle\;(\varphi_1,\varphi_2)\\ = {} & \varphi_1 \triangledown size_2\langle Perfect\rangle\;(size_2\langle Node\rangle\;(\varphi_1,\varphi_2),\varphi_2) \end{aligned}$$

Using the original constructor names we obtain

$$\begin{aligned} & sizePerfect_2 && :: (\forall a.f_1\;a \to Int, \forall a.f_2\;a \to Int)\\ & && \quad \to (\forall a.Perfect\;(f_1\;a)\;(f_2\;a) \to Int)\\ & sizePerfect_2\;(\varphi_1,\varphi_2)\;(Zero\;a) = \varphi_1\;a\\ & sizePerfect_2\;(\varphi_1,\varphi_2)\;(Succ\;t) = sizePerfect_2\;(sizeNode_2\;(\varphi_1,\varphi_2),\varphi_2)\;t\;. \end{aligned}$$

Since $size\langle f\rangle$ has the polymorphic type $\tau(f) = \forall a.f\;a \to Int$, the auxiliary functions $size_n\langle g\rangle$ take polymorphic functions to polymorphic functions. In other

words, $sizePerfect_2$ must be assigned a rank-2 type signature [15]. Unfortunately, the above definition passes neither the Hugs 98 [14] nor the GHC [20] type-checker though both accept rank-2 type signatures. The reason is that Haskell provides only a limited form of *type constructor polymorphism*. Consider the subexpression $sizeNode_2\ (\varphi_1, \varphi_2)$ in the last equation. It has type $\forall a.Node\ (g_1\ a)\ (g_2\ a) \to Int$, which is not unifiable with the expected type $\forall a.f_1\ a \to Int$. Since Haskell deliberately omits type abstractions from the language of types [13], we cannot instantiate f_1 to $\lambda u \to Node\ (g_1\ u)\ (g_2\ u)$. Fortunately, there is a way out of this dilemma. We simply replace each application of the functor variables f_1 and f_2 by a new type variable, ie $f_1\ a$ is replaced by x_1 and $f_2\ a$ by x_2.

$$sizePerfect_2 :: (x_1 \to Int, x_2 \to Int) \to (Perfect\ x_1\ x_2 \to Int)$$

This trick works for arbitrary type signatures and as a benevolent side effect the rank-2 type signature has turned into a standard rank-1 signature.

4 Examples

4.1 Polytypic *map*

Categorically speaking, a functor is the combination of a type constructor and a mapping function, which describes the action of the functor on functions. The mapping function can be generically defined for all datatypes.[2]

$$\begin{array}{lll}
fmap\langle f\rangle & :: \forall a.\forall b.(a \to b) \to (f\ a \to f\ b) \\
fmap\langle f\rangle\ \varphi & = map\langle f\rangle \\
\quad \textbf{where}\ map\langle f\rangle & :: f\ a \to f\ b \\
\qquad map\langle Id\rangle\ x & = \varphi\ x \\
\qquad map\langle Kt\rangle\ x & = x \\
\qquad map\langle f_1 + f_2\rangle\ (Inl\ x_1) & = Inl\ (map\langle f_1\rangle\ x_1) \\
\qquad map\langle f_1 + f_2\rangle\ (Inr\ x_2) & = Inr\ (map\langle f_2\rangle\ x_2) \\
\qquad map\langle f_1 \times f_2\rangle\ (x_1, x_2) & = (map\langle f_1\rangle\ x_1, map\langle f_2\rangle\ x_2)
\end{array}$$

It is revealing to consider the typings of the subsidiary functions $map_n\langle g\rangle$.

$$\begin{array}{l}
map_n\langle g\rangle\ ::\ (g_1\ a \to g_1\ b, \ldots, g_n\ a \to g_n\ b) \\
\qquad \to (g\ (g_1\ a)\ \ldots\ (g_n\ a) \to g\ (g_1\ b)\ \ldots\ (g_n\ b))
\end{array}$$

Replacing $g_i\ a$ by x_i and $g_i\ b$ by y_i we obtain the typings of n-ary mapping functions. And, in fact, $map_n\langle g\rangle$ corresponds to the mapping function of the n-ary functor g. Thus, to define $fmap\langle f\rangle$ generically for all types mapping functions of functors of arbitrary arity are required. The good news is that the programmer need not take care of their definition. Instead, they are generated automatically.

[2] We assume that type variables appearing in type signatures are scoped, ie the type variables a and b in the signature of $map\langle f\rangle$ are *not* universally quantified but refer to the occurrences in $fmap\langle f\rangle$'s signature.

4.2 Polytypic Reduction

The functions *size* and *sum* are instances of a more general concept, due to Meertens [16], termed *reduction* or *crush*. A reduction is a function of type $f\ a \to a$ that collapses a structure of values of type a into a single value of type a. To define a reduction two ingredients are required: a value $e :: a$ and a binary operation $op :: a \to a \to a$. Usually but not necessarily e is the neutral element of op.

$$
\begin{array}{lll}
reduce\langle f\rangle & :: \forall a.\, a \to (a \to a \to a) \to f\ a \to a \\
reduce\langle f\rangle\ e\ op & = red\langle f\rangle \\
\quad \textbf{where}\ red\langle f\rangle & :: f\ a \to a \\
\qquad red\langle Id\rangle\ x & = x \\
\qquad red\langle K\,t\rangle\ x & = e \\
\qquad red\langle f_1 + f_2\rangle\ (Inl\ x_1) & = red\langle f_1\rangle\ x_1 \\
\qquad red\langle f_1 + f_2\rangle\ (Inr\ x_2) & = red\langle f_2\rangle\ x_2 \\
\qquad red\langle f_1 \times f_2\rangle\ (x_1, x_2) & = red\langle f_1\rangle\ x_1\ `op`\ red\langle f_2\rangle\ x_2
\end{array}
$$

A number of useful functions can be defined in terms of $reduce\langle f\rangle$ and $fmap\langle f\rangle$, see Figure 2. Meertens [16], Jansson and Jeuring [12] give further applications.

$$
\begin{array}{lll}
sum\langle f\rangle & :: \forall n.(Num\ n) \Rightarrow f\ n \to n \\
sum\langle f\rangle & = reduce\langle f\rangle\ 0\ (+) \\
and\langle f\rangle & :: f\ Bool \to Bool \\
and\langle f\rangle & = reduce\langle f\rangle\ True\ (\wedge) \\
minimum\langle f\rangle & :: \forall a.(Bounded\ a, Ord\ a) \Rightarrow f\ a \to a \\
minimum\langle f\rangle & = reduce\langle f\rangle\ maxBound\ min \\
size\langle f\rangle & :: \forall a.\forall n.(Num\ n) \Rightarrow f\ a \to n \\
size\langle f\rangle & = sum\langle f\rangle \circ fmap\langle f\rangle\ (const\ 1) \\
all\langle f\rangle & :: \forall a.(a \to Bool) \to (f\ a \to Bool) \\
all\langle f\rangle\ p & = and\langle f\rangle \circ fmap\langle f\rangle\ p \\
flatten\langle f\rangle & :: \forall a.f\ a \to List\ a \\
flatten\langle f\rangle & = reduce\langle f\rangle\ [\,]\ (+\!\!+) \circ fmap\langle f\rangle\ wrap\ \textbf{where}\ wrap\ a = [\,a\,] \\
\textbf{data}\ Tree\ a & = Empty \mid Leaf\ a \mid Fork\ (Tree\ a)\ (Tree\ a) \\
shape\langle f\rangle & :: \forall a.f\ a \to Tree\ a \\
shape\langle f\rangle & = reduce\langle f\rangle\ Empty\ Fork \circ fmap\langle f\rangle\ Leaf
\end{array}
$$

Fig. 2. Examples of reductions

4.3 Polytypic Equality and Comparison

The equality function is a nice example of a function that is indexed by a nullary functor. In Haskell it can be automatically derived for datatypes of first-order

kind and the following can be seen as a formalization of that process. We assume that a suitable equality function for *Int* is predefined.

$$\begin{array}{ll}
eq\langle f\rangle & :: f \to f \to Bool \\
eq\langle 1\rangle\ x\ y & = True \\
eq\langle Int\rangle\ x\ y & = eqInt\ x\ y \\
eq\langle f_1 + f_2\rangle\ (Inl\ x_1)\ (Inl\ y_1) & = eq\langle f_1\rangle\ x_1\ y_1 \\
eq\langle f_1 + f_2\rangle\ (Inl\ x_1)\ (Inr\ y_2) & = False \\
eq\langle f_1 + f_2\rangle\ (Inr\ x_2)\ (Inl\ y_1) & = False \\
eq\langle f_1 + f_2\rangle\ (Inr\ x_2)\ (Inr\ y_2) & = eq\langle f_2\rangle\ x_2\ y_2 \\
eq\langle f_1 \times f_2\rangle\ (x_1, x_2)\ (y_1, y_2) & = eq\langle f_1\rangle\ x_1\ y_1 \wedge eq\langle f_2\rangle\ x_2\ y_2
\end{array}$$

Varying the definition of $eq\langle f\rangle$ slightly we can also realize Haskell's *compare* function, which determines the precise ordering of two elements.

$$\begin{array}{ll}
\textbf{data}\ Ordering & = LT \mid EQ \mid GT \\
cmp\langle f\rangle & :: f \to f \to Ordering \\
cmp\langle 1\rangle\ x\ y & = EQ \\
cmp\langle Int\rangle\ x\ y & = cmpInt\ x\ y \\
cmp\langle f_1 + f_2\rangle\ (Inl\ x_1)\ (Inl\ y_1) & = cmp\langle f_1\rangle\ x_1\ y_1 \\
cmp\langle f_1 + f_2\rangle\ (Inl\ x_1)\ (Inr\ y_2) & = LT \\
cmp\langle f_1 + f_2\rangle\ (Inr\ x_2)\ (Inl\ y_1) & = GT \\
cmp\langle f_1 + f_2\rangle\ (Inr\ x_2)\ (Inr\ y_2) & = cmp\langle f_2\rangle\ x_2\ y_2 \\
cmp\langle f_1 \times f_2\rangle\ (x_1, x_2)\ (y_1, y_2) & = \textbf{case}\ cmp\langle f_1\rangle\ x_1\ y_1\ \textbf{of} \\
 & \qquad \{\, EQ \to cmp\langle f_2\rangle\ x_2\ y_2;\ ord \to ord \,\}
\end{array}$$

5 Related and Future Work

This paper can be regarded as a successor to my previous work on polytypic programming [7], where a similar approach using *rational trees* is presented. The major difference between the two frameworks lies in the treatment of functor composition. In the 'rational tree approach' functor composition is regarded as a function symbol, which implies that a polytypic definition must specify the action on $g \cdot (g_1, \ldots, g_n)$ for each $n \geqslant 1$. Clearly, this cannot be done exhaustively. Furthermore, since the cases for functor composition are redundant, there is no guarantee that the polytypic function behaves the same on equal functors such as *Rose* and *Rose'*. Both problems disappear if we handle functor composition on the meta level generalizing rational trees to algebraic trees.

The classic approach to polytypic programming as realized in the polytypic programming language extension PolyP [10] is based on the initial algebra semantics of datatypes. Here, functors are modeled by the following grammar.

$$\begin{array}{l}
F = \mu B \\
B = K\,T \mid Fst \mid Snd \mid B + B \mid B \times B \mid F \cdot B
\end{array}$$

Recursive datatypes are modeled by fixpoints of associated base functors: the functor μB, which is known as a *type functor*, denotes the unary functor F

given as the least solution of the equation $F\ a = B(a, F\ a)$. Polytypic functions are defined according to the above structure of types. In PolyP the polytypic function $size\langle f\rangle$ is defined as follows—modulo change of notation.

$$
\begin{array}{ll}
size\langle f\rangle & :: \forall a.f\ a \to Int \\
size\langle \mu b\rangle & = cata\langle \mu b\rangle\ (bsize\langle b\rangle) \\
bsize\langle b\rangle & :: \forall a.b\ a\ Int \to Int \\
bsize\langle Kt\rangle\ x & = 0 \\
bsize\langle Fst\rangle\ x & = 1 \\
bsize\langle Snd\rangle\ n & = n \\
bsize\langle b_1 + b_2\rangle\ (Inl\ x_1) & = bsize\langle b_1\rangle\ x_1 \\
bsize\langle b_1 + b_2\rangle\ (Inr\ x_2) & = bsize\langle b_2\rangle\ x_2 \\
bsize\langle b_1 \times b_2\rangle\ (x_1, x_2) & = bsize\langle b_1\rangle\ x_1 + bsize\langle b_2\rangle\ x_2 \\
bsize\langle f \cdot b\rangle\ x & = sum\langle f\rangle\ (fmap\langle f\rangle\ (bsize\langle b\rangle)\ x)
\end{array}
$$

The program is quite elaborate as compared to the one given in Section 1: it involves two general combining forms, $cata\langle f\rangle$ and $fmap\langle f\rangle$, and an auxiliary polytypic function, $sum\langle f\rangle$. The disadvantages of the initial algebra approach are fairly obvious. The above definition is redundant: we know that $size\langle f\rangle$ is uniquely defined by its action on constant functors, Id, sums, and products. The definition is incomplete: $size\langle f\rangle$ is only applicable to regular functors (recall that, for instance, *Perfect* is not a regular functor). Furthermore, the regular functor may not depend on functors of arity $\geqslant 2$ since functor composition is only defined for unary functors. Finally, the definition exhibits a slight inefficiency: the combing form $fmap\langle f\rangle$ produces an intermediate data structure, which is immediately consumed by $sum\langle f\rangle$. In other words, $size\langle Rose\rangle$ corresponds to the first, less efficient definition of *sizer*.

An obvious advantage of PolyP is that it allows to define general recursion schemes like cata- and anamorphisms [17]. As an example, the recursion scheme $cata\langle f\rangle$, which is used in $size\langle f\rangle$, is given by

$$
\begin{array}{ll}
cata\langle f\rangle & :: \forall a.\forall b.(f'\ a\ b \to b) \to (f\ a \to b) \\
cata\langle \mu b\rangle\ \varphi & = \varphi \circ bmap\langle b\rangle\ id\ (cata\langle \mu b\rangle\ \varphi) \circ out\ \ .
\end{array}
$$

The operation $(\text{-})'$ (called *FunctorOf* in PolyP) maps a type functor to its base functor, ie $F' = B$ for $F = \mu B$. The function $out :: f\ a \to f'\ a\ (f\ a)$ decomposes an element of type $f\ a$ by unfolding one level of recursion. While the explicit treatment of type recursion is unnecessary for many applications—all polytypic functions of PolyLib [12] can be implemented in our framework, except, of course, cata- and anamorphisms—it is indispensable for others. The polytypic unification algorithm described by Jansson and Jeuring [11], for instance, requires a function that determines the immediate subterms of a term. A similar function appears in the article of Bird, de Moor, and Hoogendijk [3], who present a generalization of the maximum segment sum problem. In both cases the recursive structure of a datatype must be known. Now, since our framework deals with type recursion on the meta level, one could feel tempted to conclude that we

cannot deal with such applications. It appears, however, that the availability of operations on types is an orthogonal design issue. Hence, nothing prevents us from incorporating the operator (-)′ depicted above. Let us furthermore generalize (-)′ so that it maps an n-ary, regular functor to its associated $(n+1)$-ary base functor. Given two polytypic functions $in :: f'\ a_1 \ldots a_n\ (f\ a_1 \ldots a_n) \to f\ a_1 \ldots a_n$ and $out :: f\ a_1 \ldots a_n \to f'\ a_1 \ldots a_n\ (f\ a_1 \ldots a_n)$ we can define cata- and anamorphisms for functors of arbitrary arity. Here are the definitions for unary functors.

$$\begin{aligned}
&cata\langle f\rangle &&:: \forall a.\forall b.(f'\ a\ b \to b) \to (f\ a \to b)\\
&cata\langle f\rangle\ \varphi &&= \varphi \circ bmap\langle f'\rangle\ id\ (cata\langle f\rangle\ \varphi) \circ out\\
&ana\langle f\rangle &&:: \forall a.\forall b.(b \to f'\ a\ b) \to (b \to f\ a)\\
&ana\langle f\rangle\ \psi &&= in \circ bmap\langle f'\rangle\ id\ (ana\langle f\rangle\ \psi) \circ \psi
\end{aligned}$$

It is worth noting that the definition of the subsidiary function $bmap\langle b\rangle$ proceeds as before. In particular, there is no need to consider functor composition or type functors. Furthermore, the base functor may be a nested functor. The function that collects the immediate subterms of a term can be defined as follows.

$$\begin{aligned}
&subterms\langle t\rangle &&:: t \to List\ t\\
&subterms\langle t\rangle &&= flatten\langle t'\rangle \circ out
\end{aligned}$$

Directions for future work suggest themselves. In the full version of this paper [8] we show that polytypic functions satisfy fusion laws analogous to the fusion law for folds. The functorial properties of $fmap\langle f\rangle$ and the fusion law for reductions [16] are, for instance, consequences of the polytypic fusion laws. It remains to broaden the approach to include exponentials and higher-order polymorphism [13]. Adapting the technique of Meijer and Hutton [18] the former extension should be fairly straightforward. Currently, the author is working on the latter extension so that polytypic functions can be defined generically for all datatypes expressible in Haskell. An application of polytypic programming to digital searching is described in a companion paper [6], where we show how to define *tries* and operations on tries generically for arbitrary datatypes of first-order kind. The central insight is that a trie can be considered as a *type-indexed datatype*, which adds an interesting new dimension to polytypic programming.

Acknowledgements

I would like to thank the anonymous referees for many valuable comments.

References

1. M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *J. Functional Programming*, 5(1):111–130, January 1995.
2. Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall Europe, London, 2nd edition, 1998.
3. Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *J. Functional Programming*, 6(1):1–28, Jan. 1996.

4. Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Fourth International Conference on Mathematics of Program Construction, MPC'98*, LNCS 1422, pages 52–67. Springer-Verlag, June 1998.
5. Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, March 1983.
6. Ralf Hinze. Generalizing generalized tries. *J. Functional Programming*, 1999. Accepted for publication.
7. Ralf Hinze. Polytypic functions over nested datatypes. *Discrete Mathematics and Theoretical Computer Science*, 1999. Accepted for publication.
8. Ralf Hinze. Polytypic programming with ease. Technical Report IAI-TR-99-2, Institut für Informatik III, Universität Bonn, February 1999.
9. P. Jansson and J. Jeuring. Polytypic compact printing and parsing. In S. Doaitse Swierstra, editor, *Proceedings European Symposium on Programming, ESOP'99*, LNCS 1576, pages 273–287. Springer-Verlag, 1999.
10. Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *24th Symposium on Principles of Programming Languages, POPL'97*, pages 470–482. ACM-Press, January 1997.
11. Patrik Jansson and Johan Jeuring. Functional Pearl: Polytypic unification. *J. Functional Programming*, 8(5):527–536, September 1998.
12. Patrik Jansson and Johan Jeuring. PolyLib—A library of polytypic functions. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98*. Department of Computing Science, Chalmers University of Technology and Göteborg University, `http://wsinwp01.win.tue.nl:1234/WGPProceedings/index.html`, June 1998.
13. Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, LNCS 925, pages 97–136. Springer-Verlag, 1995.
14. M.P. Jones and J.C. Peterson. *Hugs 98 User Manual*, May 1999. Available from `http://www.haskell.org/hugs`.
15. Nancy Jean McCracken. The typechecking of programs with implicit type structure. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types*, LNCS 173, pages 301–315. Springer-Verlag, 1984.
16. Lambert Meertens. Calculate polytypically! In H. Kuchen and S.D. Swierstra, editors, *8th International Symposium on Programming Languages: Implementations, Logics, and Programs, PLILP'96*, LNCS 1140, pages 1–16. Springer-Verlag, September 1996.
17. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *5th Conference on Functional Programming Languages and Computer Architecture, FPCA'91*, LNCS 523, pages 124–144. Springer-Verlag, 1991.
18. Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *7th Conference on Functional Programming Languages and Computer Architecture, FPCA'95*, pages 324–333. ACM-Press, June 1995.
19. Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
20. Simon Peyton Jones. Explicit quantification in Haskell, 1998. Available from `http://research.microsoft.com/Users/simonpj/Haskell/quantification.html`.
21. Simon Peyton Jones and John Hughes, editors. *Haskell 98 — A Non-strict, Purely Functional Language*. `http://www.haskell.org/onlinereport/`. February 1999.
22. Zhe Yang. Encoding types in ML-like languages. *SIGPLAN Notices*, 34(1):289–300, January 1999.