Polytypic Programming With Ease

February 1999

RALF HINZE

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* `ralf@informatik.uni-bonn.de`)

Abstract

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. We show that 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 polymorphic 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 appears both to be simpler, more general, and more efficient than previous ones which are based on the initial algebra semantics of datatypes. Polytypic functions enjoy polytypic properties. We show that well-known properties of various functions are obtained as direct consequences of two general polytypic fusion laws.

1 Introduction

A generic or polytypic function is one which is defined by induction on the structure of types. The archetypical example of a polytypic function is $size :: f\ a \rightarrow Int$ 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 polymorphic type 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.

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. Definitions are usually given in a point-free style, ie functions are described in terms of functional composition and a few additional combining forms. This style of programming may look daunting to the beginner. However, once a certain familiarity has been gained, it turns out that polytypic programming is actually simpler (if not much 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]

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

The size of a rose tree can be determined as follows (Bird, 1998).

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

This definition is already quite sophisticated: it makes use of a general combining form and an auxiliary function. The use of *map*, 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.

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

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

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

We will return to this point in Section 2. The definition of *sum* proceeds completely analogous—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.

$$\mathbf{data}\ \mathit{Perfect}\ a\ k = \mathit{Zero}\ a \mid \mathit{Succ}\ (\mathit{Perfect}\ (a, k, a)\ k)$$

The definition of *Perfect* is somewhat unusual in that the recursive component, $\mathit{Perfect}\ (a, k, a)\ k$, is not identical to the left-hand side of the equation: *Perfect* is an example for a so-called *nested datatype* (Bird & Meertens, 1998). Since *Perfect* only encompasses perfect binary search trees, the size can be computed in logarithmic time.

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

The function *sizep* counts the number of values of type a in a tree of type $\mathit{Perfect}\ a\ a$. However, we cannot assign *sizep* the type $\mathit{Perfect}\ a\ a \to \mathit{Int}$ since the recursive call

[1] Examples are given in the functional programming language Haskell 98 (Peyton Jones [editor] *et al.*, 1998). Deviating from Haskell we assume, however, that primitive functions are non-curried, ie $(+)$ has type $\mathit{Int} \times \mathit{Int} \to \mathit{Int}$ rather than $\mathit{Int} \to (\mathit{Int} \to \mathit{Int})$.

has type $Perfect\ (a, k, a)\ k \rightarrow Int$. As an aside, this also shows that the definition makes use of *polymorphic recursion* (Mycroft, 1984).

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

$$\begin{array}{lcl} sump & :: & Perfect\ Int\ Int \rightarrow Int \\ sump\ (Zero\ a) & = & a \\ sump\ (Succ\ t) & = & sump\ (bmap\ add3\ id\ t) \\ add3 & :: & (Int, Int, Int) \rightarrow Int \\ add3\ (a, b, c) & = & a + b + c \end{array}$$

Here, $bmap$ denotes the mapping function of a bifunctor. Improving the efficiency of $sump$ by fusing $sump \circ bmap\ \varphi_1\ \varphi_2$ into a single function is left as an instructive exercise to the reader.

Now, let us try to 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 we rewrite the definitions as *functor equations*:

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

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.

To define $size$ it suffices to specify its action on constant functors, on the identity functor, on sums, and on products.[2]

$$\begin{array}{lcl} size\langle f\rangle & :: & \forall a.f\ a \rightarrow Int \\ size\langle K\,t\rangle & = & const\ 0 \\ size\langle Id\rangle & = & const\ 1 \\ size\langle f_1 + f_2\rangle & = & size\langle f_1\rangle \ \nabla\ size\langle f_2\rangle \\ size\langle f_1 \times f_2\rangle & = & (+) \circ (size\langle f_1\rangle \times size\langle f_2\rangle) \end{array}$$

Polytypic functions are written using angle brackets to distinguish them from ordinary functions. Each of the defining equations is more or less inevitable: a structure of type $K\,t\ a = t$ contains no elements of type a; $Id\ a = a$ contains exactly one element. To determine the size of an element of type $f\ a + g\ a$ we must either calculate the size of a structure of type $f\ a$ or that of a structure of type $g\ a$ depending on which component of the disjoint sum the argument comes from. Finally, the size of

[2] The definitions of the auxiliary functions are listed in Appendix A.

a structure of type $f\ a \times g\ a$ is given by the sum of the size of the two components. Given these definitions the following specializations can be derived automatically.

$$
\begin{array}{lcl}
\mathit{size}\langle \mathit{Rose}\rangle & = & \mathit{size}_1\langle \mathit{Rose}\rangle\ (\mathit{const}\ 1) \\
\mathit{size}_1\langle \mathit{Rose}\rangle\ \varphi & = & (+) \circ (\varphi \times \mathit{size}_1\langle \mathit{List}\rangle\ (\mathit{size}_1\langle \mathit{Rose}\rangle\ \varphi)) \\
\mathit{size}_1\langle \mathit{List}\rangle\ \varphi & = & \mathit{const}\ 0 \triangledown ((+) \circ (\varphi \times \mathit{size}_1\langle \mathit{List}\rangle\ \varphi)) \\
\mathit{size}\langle \mathit{Perfect} \cdot (\mathit{Id}, \mathit{Id})\rangle & = & \mathit{size}_2\langle \mathit{Perfect}\rangle\ (\mathit{const}\ 1, \mathit{const}\ 1) \\
\mathit{size}_2\langle \mathit{Perfect}\rangle\ (\varphi_1, \varphi_2) & = & \varphi_1 \triangledown \mathit{size}_2\langle \mathit{Perfect}\rangle\ (\mathit{add3} \circ (\varphi_1 \times \varphi_2 \times \varphi_1), \varphi_2)
\end{array}
$$

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: unary functors give rise to unary functions and binary functors to binary functions. How do these functions relate to the handcrafted definitions? If we strip off the abstract clothing of $\mathit{size}\langle \mathit{Rose}\rangle$, we see that it essentially corresponds to the second, more efficient definition of *sizer*: we have $\mathit{sizer} = \mathit{size}_1\langle \mathit{Rose}\rangle\ (\mathit{const}\ 1)$ and $\mathit{sizef} = \mathit{size}_1\langle \mathit{List}\rangle\ \mathit{sizer}$. On the negative side, $\mathit{size}\langle \mathit{Perfect} \cdot (\mathit{Id}, \mathit{Id})\rangle$ 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 which exploits data-structural invariants.

The polytypic definition of *sum* is equally simple.

$$
\begin{array}{lcl}
\mathit{sum}\langle f\rangle & :: & f\ \mathit{Int} \to \mathit{Int} \\
\mathit{sum}\langle K\, t\rangle & = & \mathit{const}\ 0 \\
\mathit{sum}\langle \mathit{Id}\rangle & = & \mathit{id} \\
\mathit{sum}\langle f_1 + f_2\rangle & = & \mathit{sum}\langle f_1\rangle \triangledown \mathit{sum}\langle f_2\rangle \\
\mathit{sum}\langle f_1 \times f_2\rangle & = & (+) \circ (\mathit{sum}\langle f_1\rangle \times \mathit{sum}\langle f_2\rangle)
\end{array}
$$

Note that $\mathit{sum}\langle f\rangle$ is identical to $\mathit{size}\langle f\rangle$ except for the third line. 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 generic functions like *size* and *sum* is sometimes simple and sometimes involving. While the polytypic definition is arguably more abstract, it is also to a high degree inevitable. It is this feature which makes polytypic programming light and sweat. In some cases, which are fortunately rare, the hand-coded definitions are more efficient. Still, polytypic programming may be of use as the improved definition can sometimes be derived from the polytypic one, see Section 5.2.

The rest of this paper is organized as follows. Section 2 introduces the basic ingredients of polytypic programming: algebraic trees and functor equations. Section 3 explains how to specialize a polytypic function to concrete instances of datatypes. Polytypic functions enjoy polytypic properties. Fusion laws analogous to the fusion law for catamorphisms are described in Section 4. Section 5 presents several examples for polytypic functions, among others polytypic reduction and polytypic equality. Finally, Section 6 reviews related work and points out directions for future work.

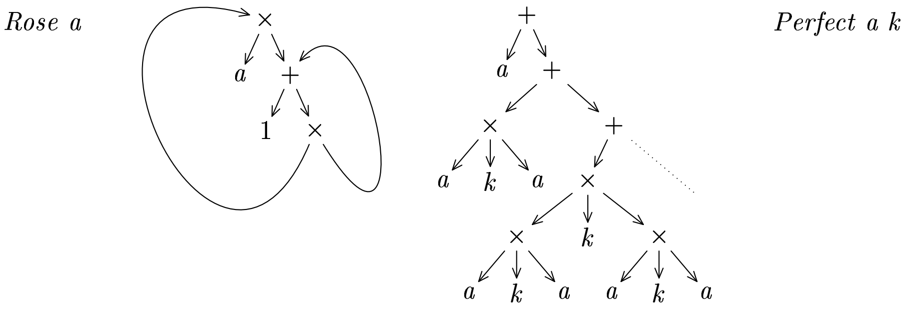


Fig. 1. Types interpreted as infinite trees.

2 Datatypes as algebraic trees

Type definitions in Haskell take the form of recursion equations. If we interpret these equations syntactically, each equation defines a unique *infinite tree*. Figure 1 displays the trees defined by the type equations given in the previous section. Note that *Rose* is interpreted by a *rational tree* while *Perfect* denotes an *algebraic tree*—both classes will be defined below.

The major difference to previous approaches to polytypic programming (Jeuring & Jansson, 1996; Jansson & Jeuring, 1997) 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. For instance, one can still define polytypic functions recursively and prove polytypic properties inductively. Of course, to make things work one must impose the usual conditions: functions must be continuous and properties must be chain complete. Both conditions are usually met.

While the theory works for arbitrary infinite trees, it is necessary and, in fact, sufficient to focus on algebraic trees. The restriction is necessary for practical considerations: the specialization of polytypic functions works only for this restricted class. Nothing is lost, however, since first-order polymorphic datatype definitions can only describe algebraic trees.

In what follows we assume a basic familiarity with infinite trees, see, for instance (Courcelle, 1983). There are several ways to represent algebraic trees. We will use *algebraic equations* which are akin to functor equations shown in the previous section. An algebraic equation has the general form $f = e$ where f is an unknown and e is a so-called *monomial*, which is the 'tree-theory equivalent' of a functor expression. Let $\Sigma = \bigcup_{n \geqslant 0} \Sigma^n$ be a ranked set of function symbols, the set of *monomials of rank* n *over* Σ, denoted $M^n(\Sigma)$, is inductively defined as follows.

$$M^n \quad = \quad \Pi_i^n \mid \Sigma^n \mid M^k \cdot (M_1^n, \ldots, M_k^n)$$

The correspondence to functor expressions should be fairly obvious: Π_i^n is the analogue of the n-ary projection functor selecting its i-th component. The monomial $e \cdot (e_1, \ldots, e_k)$ corresponds to the composition of a k-ary functor e with functors e_i, all of arity n.

Now, let $V = \{a_1, a_2, \ldots\}$ be a fixed set of variables. Every monomial can be interpreted as a tree, ie a polymorphic type, in the following way:

$$\begin{aligned} [\![\Pi_i]\!] &= a_i \\ [\![F]\!] &= F(a_1, \ldots, a_k) \quad \text{for } F \in \Sigma^k \\ [\![e \cdot (e_1, \ldots, e_n)]\!] &= [\![e]\!] \circ ([\![e_1]\!], \ldots, [\![e_n]\!]) \ , \end{aligned}$$

where the composition is interpreted as a substitution operation on infinite trees: $t \circ (t_1, \ldots, t_n) = t[a_1/t_1, \ldots, a_n/t_n]$ for $t \in M^n(\Sigma)$. The substitution operation satisfies the following properties:

$$t \circ (a_1, \ldots, a_n) = t \tag{1}$$

$$a_i \circ (t_1, \ldots, t_n) = t_i \tag{2}$$

$$(t \circ (u_1, \ldots, u_n)) \circ \mathbf{v} = t \circ (u_1 \circ \mathbf{v}, \ldots, u_n \circ \mathbf{v}) \ . \tag{3}$$

Note that vectors are written in bold face. It will be convenient to use the same notation for trees and for their representation by monomials. To this end, we define trees as infinite terms formed according to

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

In other words, we identify a_i with Π_i and $F(t_1, \ldots, t_n)$ with $F \cdot (t_1, \ldots, t_n)$.

Let $\Phi = \{f_1, f_2, \ldots\}$ be a ranked set. A *system of algebraic equations over* Σ *in the set of unknowns* Φ is of the form $\langle f_1 = e_1, \ldots, f_n = e_n \rangle$ with $f_i \in \Phi^{k_i}$, $e_i \in M^{k_i}(\Sigma \cup \Phi)$, and $f_i \neq f_j$ for $i \neq j$. A system ist termed *normalized* if each right-hand side is of the form $F \cdot (e'_1, \ldots, e'_k)$ with $F \in \Sigma^k$ for some k. Each algebraic system can be transformed into an equivalent system which is in normalized form. In this process an equation of the form $f = f$ is replaced by $f = \Omega$ where Ω is a new nullary function symbol representing the 'bottom' tree. A system of algebraic equations where each right-hand side is of the form Π_i^n or $F \cdot (f_1, \ldots, f_k)$ with $F \in \Sigma^k$ and $f_i \in \Phi$ is called a *system of regular equations*.

A solution of the system $\langle f_1 = e_1, \ldots, f_n = e_n \rangle$ is an n-tuple $\langle t_1, \ldots, t_n \rangle$ such that $t_i = [\![e_i]\!]$ where the interpretation of the unknowns is given by $[\![f_i]\!] = t_i$ for all i. It is well-known that each normalized algebraic system has a *unique* solution in the realm of infinite trees. The trees obtained as solutions are called *algebraic trees*. An unrestricted system has in general only a *least* solution. Each tree is, for instance, a solution of the trivial system $\langle f = f \rangle$. Solutions of regular systems are called *rational trees*. Note that a rational tree has only a finite number of subtrees.

Now, *functor equations* are algebraic equations over the following set of symbols.

$$T = 1 \mid \mathit{Int} \mid \mathit{Double} \mid \mathit{Char} \mid T + T \mid T \times T \ ,$$

ie $\Sigma = \Sigma^0 \cup \Sigma^2$ with $\Sigma^0 = \{1, \mathit{Int}, \mathit{Double}, \mathit{Char}\}$ and $\Sigma^2 = \{+, \times\}$. To improve readability, we write $t \cdot ()$ as $K\,t$ for $t \in \Sigma^0$ and $\oplus \cdot (f, g)$ as $f \oplus g$ for $\oplus \in \Sigma^2$. For unary and binary projection functors we use the following more familiar names: $\mathit{Id} = \Pi_1^1$, $\mathit{Fst} = \Pi_1^2$, and $\mathit{Snd} = \Pi_2^2$.

Let us consider some examples. 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 which 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 for 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.

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

The functor *Sequ* is used in (Okasaki, 1998) 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.

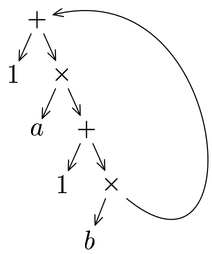
$$\begin{aligned} \mathit{Rose} &= \mathit{Id} \times \mathit{List} \cdot \mathit{Rose} & \qquad \mathit{Rose}' &= \mathit{Id} \times \mathit{Forest} \\ \mathit{List} &= K1 + \mathit{Id} \times \mathit{List} & \qquad \mathit{Forest} &= K1 + \mathit{Rose}' \times \mathit{Forest} \end{aligned}$$

Both *Rose* and *Rose'* denote the same rational tree depicted in Figure 1. Note that *Rose* is given by an algebraic system while *Rose'* is defined by a regular system. We can show $\mathit{Rose} = \mathit{Rose}'$ and $\mathit{List} \cdot \mathit{Rose} = \mathit{Forest}$ using a *co-inductive* argument.

$$\begin{array}{ll} & \mathit{Rose} \\ = & \quad \{ \text{ def. } \mathit{Rose} \ \} \\ & \mathit{Id} \times \mathit{List} \cdot \mathit{Rose} \\ = & \quad \{ \text{ ex hypothesi } \} \\ & \mathit{Id} \times \mathit{Forest} \\ = & \quad \{ \text{ def. } \mathit{Rose}' \ \} \\ & \mathit{Rose}' \end{array} \qquad \begin{array}{ll} & \mathit{List} \cdot \mathit{Rose} \\ = & \quad \{ \text{ def. } \mathit{List} \ \} \\ & (K1 + \mathit{Id} \times \mathit{List}) \cdot \mathit{Rose} \\ = & \quad \{ \text{ (2) and (3) } \} \\ & K1 + \mathit{Rose} \times \mathit{List} \cdot \mathit{Rose} \\ = & \quad \{ \text{ ex hypothesi } \} \\ & K1 + \mathit{Rose}' \times \mathit{Forest} \\ = & \quad \{ \text{ def. } \mathit{Forest} \ \} \\ & \mathit{Forest} \end{array}$$

The equality problem for algebraic trees is by no means trivial. It is well-known that this problem and the equivalence problem for deterministic pushdown automata are interreducible (Courcelle, 1983). It was, however, only recently shown that the latter problem is decidable (Sénizergues, 1997).

Finally, let us note that the classification of functors into polynomial, regular, and nested functors deviates slightly from the usual notions. To exemplify, consider the following datatype definition.



$$\mathbf{data}\ \mathit{ZigZag}\ a\ b \quad = \quad \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{ZigZag}\ b\ a)$$

Since the recursive call on the right-hand side, $\mathit{ZigZag}\ b\ a$, is not a copy of the left-hand side, *ZigZag* qualifies as a nested datatype. It denotes, however, the rational tree depicted on the right. An equivalent, non-nested definition for *ZigZag* is given by

$$\begin{aligned} \mathbf{data}\ \mathit{Zig}\ a\ b \quad &= \quad \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{Zag}\ a\ b) \\ \mathbf{data}\ \mathit{Zag}\ a\ b \quad &= \quad \mathit{Nil} \mid \mathit{Cons}\ b\ (\mathit{Zig}\ a\ b)\ . \end{aligned}$$

We can prove $\mathit{ZigZag} = \mathit{Zig}$ and $\mathit{ZigZag} \cdot (\mathit{Snd}, \mathit{Fst}) = \mathit{Zag}$ using a co-inductive argument. While the standard classification of functors is purely syntactical—the form matters—, the classification via tree classes has a mild semantic flavour—the structure matters.

To summarize: polymorphic types are modeled by infinite trees, whose inner nodes are labelled with primitive functors of arity $\geqslant 1$ and whose leafs are decorated with primitive types and type variables. Trees are represented by algebraic equations which have the general form $f = e$ where f is a functor variable and e is a functor expression. Regular types correspond to rational trees and nested types to algebraic trees.

3 Polytypic functions

A polytypic function is given by the following data.

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

If f is an n-ary functor, poly_{Π_i} must be specified for $1 \leqslant i \leqslant n$. Furthermore, an equation specifying poly_F must be given for each primitive functor $F \in \Sigma$. This information is sufficient to define a unique function $\mathit{poly}\langle f\rangle$ for each functor f (Courcelle, 1983, Prop. 2.4.2). We tacitly assume that $\mathit{poly}_\Omega = \bot$ for the 'bottom' tree Ω. Note that the definition of $\mathit{poly}\langle f\rangle$ is *inductive on the structure of types*—recall that we identify a_i with Π_i and $F(t_1, \ldots, t_n)$ with $F \cdot (t_1, \ldots, t_n)$. The question we pursue in this section is how to derive specializations of $\mathit{poly}\langle f\rangle$ for given instances of f.

The central idea is to let function follow type. We have seen that functor composition corresponds to a substitution operation on trees. Simulating substitution on the function level we define, for each n-ary functor f, a polytypic function

$poly_n\langle f\rangle :: (\tau(g_1),\ldots,\tau(g_n)) \to \tau(f \cdot (g_1,\ldots,g_n))$ which satisfies the following property

$$poly_n\langle f\rangle\ \varphi = poly\langle f\cdot(g_1,\ldots,g_n)\rangle \quad \Longleftarrow \quad \varphi = (poly\langle g_1\rangle,\ldots,poly\langle g_n\rangle)\ .$$

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

Now, for primitive functors $poly_n\langle F\rangle$ is defined by $poly_n\langle F\rangle = poly_F$. The definition of *size* given in Section 1 may serve as an example. We have $size_{K\,t} = const\ 0$, $size_{Id} = const\ 1$, $size_+(\varphi_1,\varphi_2) = \varphi_1 \triangledown \varphi_2$, and $size_\times(\varphi_1,\varphi_2) = (+)\circ(\varphi_1\times\varphi_2)$. The derivation of $poly_n\langle f\rangle$ for projection functors and for functor composition is a nice example in program calculation and proceeds entirely mechanically. We assume that $\varphi = (poly\langle g_1\rangle,\ldots,poly\langle g_n\rangle)$ and reason:

$$\begin{array}{ll}
 & poly_n\langle \Pi_i\rangle\ \varphi \\
= & \quad \{\ \text{specification}\ \} \\
 & poly\langle \Pi_i\cdot(g_1,\ldots,g_n)\rangle \\
= & \quad \{\ (2)\ \} \\
 & poly\langle g_i\rangle \\
= & \quad \{\ \text{assumption}\ \} \\
 & \varphi_i\ .
\end{array}$$

For functor composition we obtain

$$\begin{array}{ll}
 & poly_n\langle f\cdot(h_1,\ldots,h_k)\rangle\ \varphi \\
= & \quad \{\ \text{specification}\ \} \\
 & poly\langle (f\cdot(h_1,\ldots,h_k))\cdot(g_1,\ldots,g_n)\rangle \\
= & \quad \{\ (3)\ \} \\
 & poly\langle f\cdot(h_1\cdot(g_1,\ldots,g_n),\ldots,h_k\cdot(g_1,\ldots,g_n))\rangle \\
= & \quad \{\ \text{specification}\ \} \\
 & poly_k\langle f\rangle\ (poly\langle h_1\cdot(g_1,\ldots,g_n)\rangle,\ldots,poly\langle h_k\cdot(g_1,\ldots,g_n)\rangle) \\
= & \quad \{\ \text{specification and assumption}\ \} \\
 & poly_k\langle f\rangle\ (poly_n\langle h_1\rangle\ \varphi,\ldots,poly_n\langle h_k\rangle\ \varphi)\ .
\end{array}$$

It remains to define $poly\langle f\rangle$. We argue

$$\begin{array}{ll}
 & poly\langle f\rangle \\
= & \quad \{\ (1)\ \} \\
 & poly\langle f\cdot(\Pi_1,\ldots,\Pi_n)\rangle \\
= & \quad \{\ \text{specification}\ \} \\
 & poly_n\langle f\rangle\ (poly\langle\Pi_1\rangle,\ldots,poly\langle\Pi_n\rangle) \\
= & \quad \{\ \text{def. } poly\ \} \\
 & poly_n\langle f\rangle\ (poly_{\Pi_1},\ldots,poly_{\Pi_n})\ .
\end{array}$$

To summarize:

$$\begin{array}{lcl}
poly\langle f\rangle & :: & \tau(f) \\
poly\langle f\rangle & = & poly_n\langle f\rangle\ (poly_{\Pi_1},\ldots,poly_{\Pi_n})
\end{array}$$

$$\begin{aligned}
\mathit{poly}_n\langle f\rangle &:: (\tau(g_1),\dots,\tau(g_n)) \to \tau(f\cdot(g_1,\dots,g_n))\\
\mathit{poly}_n\langle \Pi_i\rangle &= \pi_i\\
\mathit{poly}_n\langle F\rangle &= \mathit{poly}_F\\
\mathit{poly}_n\langle f\cdot(h_1,\dots,h_k)\rangle &= \mathit{poly}_k\langle f\rangle \star (\mathit{poly}_n\langle h_1\rangle,\dots,\mathit{poly}_n\langle h_k\rangle)\ ,
\end{aligned}$$

where $\pi_i\,(\varphi_1,\dots,\varphi_n) = \varphi_i$ is the i-th projection function and '$\star$' denotes n-ary composition defined by $(\varphi \star (\varphi_1,\dots,\varphi_n))\,a = \varphi\,(\varphi_1\ a,\dots,\varphi_n\ a)$. Note that the definition of $\mathit{poly}_n\langle f\rangle$ 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, F by poly_F, and '$\cdot$' by '$\star$'. Furthermore note, that polytypic functions which differ only in their action on projection functors yield identical poly_n functions: we have, for instance, $\mathit{size}_n\langle f\rangle = \mathit{sum}_n\langle f\rangle$.

By now we have the necessary prerequisites at hand to define the specialization of a polytypic function $\mathit{poly}\langle f\rangle$ for a given instance of f. Assume that the functor is defined by the algebraic system $\langle f_1 = e_1,\dots,f_m = e_m\rangle$ with $f = f_i$ for some i. For each equation $f_i = e_i$ with $f_i \in \Phi^{k_i}$ a function definition of the form $\mathit{poly}_{k_i}\langle f_i\rangle = \mathit{poly}_{k_i}\langle e_i\rangle$ is generated. The expression $\mathit{poly}_{k_i}\langle e_i\rangle$ is given by the inductive definition above, additionally setting $\mathit{poly}_{k_i}\langle f_i\rangle = \mathit{poly\_k_i\_f_i}$, where $\mathit{poly\_k_i\_f_i}$ is a new function symbol. Finally, the defining equation for $\mathit{poly\_f}$, ie $\mathit{poly\_f} = \mathit{poly\_n\_f}\ (\mathit{poly}_{\Pi_1},\dots,\mathit{poly}_{\Pi_n})$, must be added.

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

$$\mathit{Perfect} \quad = \quad \mathit{Fst} + \mathit{Perfect}\cdot(\mathit{Fst}\times\mathit{Snd}\times\mathit{Fst},\mathit{Snd})$$

The specialization of $\mathit{size}_2\langle\mathit{Perfect}\rangle = \mathit{size2P}$ proceeds as follows.

$$\begin{aligned}
& \mathit{size}_2\langle \mathit{Fst} + \mathit{Perfect}\cdot(\mathit{Fst}\times\mathit{Snd}\times\mathit{Fst},\mathit{Snd})\rangle\ (\varphi_1,\varphi_2)\\
={}& \mathit{size}_2\langle(+)\rangle\ (\mathit{size}_2\langle\mathit{Fst}\rangle\ (\varphi_1,\varphi_2), \mathit{size}_2\langle\mathit{Perfect}\cdot(\mathit{Fst}\times\mathit{Snd}\times\mathit{Fst},\mathit{Snd})\rangle\ (\varphi_1,\varphi_2))\\
={}& \mathit{size}_2\langle\mathit{Fst}\rangle\ (\varphi_1,\varphi_2)\ \triangledown\ \mathit{size}_2\langle\mathit{Perfect}\cdot(\mathit{Fst}\times\mathit{Snd}\times\mathit{Fst},\mathit{Snd})\rangle\ (\varphi_1,\varphi_2)\\
={}& \varphi_1\ \triangledown\ \mathit{size}_2\langle\mathit{Perfect}\rangle\ (\mathit{size}_2\langle\mathit{Fst}\times\mathit{Snd}\times\mathit{Fst}\rangle\ (\varphi_1,\varphi_2), \mathit{size}_2\langle\mathit{Snd}\rangle\ (\varphi_1,\varphi_2))\\
={}& \varphi_1\ \triangledown\ \mathit{size}_2\langle\mathit{Perfect}\rangle\ (\mathit{add3}\circ(\varphi_1\times\varphi_2\times\varphi_1),\varphi_2)
\end{aligned}$$

Thus, we obtain

$$\begin{aligned}
\mathit{size2P} &:: (\forall a.f_1\ a \to \mathit{Int}, \forall a.f_2\ a \to \mathit{Int}) \to (\forall a.\mathit{Perfect}\ (f_1\ a)\ (f_2\ a) \to \mathit{Int})\\
\mathit{size2P}\ (\varphi_1,\varphi_2) &= \varphi_1\ \triangledown\ \mathit{size2P}\ (\mathit{add3}\circ(\varphi_1\times\varphi_2\times\varphi_1),\varphi_2)\ .
\end{aligned}$$

Since $\mathit{size}\langle f\rangle$ has a polymorphic type, the auxiliary functions $\mathit{size}_n\langle f\rangle$ take polymorphic functions to polymorphic functions. In other words, *size2P* must be assigned a rank-2 type signature (McCracken, 1984).

The correctness of the specialization is implied by the following proposition. While the derivation above happily mingles functor expressions and their interpretation as trees, we have been careful to separate 'syntax and semantics' below.

Proposition 1
Let $\mathit{poly}\langle f\rangle$ be a polytypic function and let $\langle f_1 = e_1,\dots,f_m = e_m\rangle$ be an algebraic system. For each $f_i \in \Phi^{k_i}$ we have

$$\mathit{poly}\langle[\![f_i]\!]\rangle \quad = \quad \mathit{poly\_k_i\_f_i}\ (\mathit{poly}_{\Pi_1},\dots,\mathit{poly}_{\Pi_{k_i}})\ ,$$

where the $\mathit{poly\_k_i\_f_i}$ are given by

$$\begin{aligned} \mathit{poly\_k_1\_f_1} &= \mathit{poly}_{k_1}\langle e_1\rangle \\ &\vdots \\ \mathit{poly\_k_m\_f_m} &= \mathit{poly}_{k_m}\langle e_m\rangle \ . \end{aligned}$$

Proof

The algebraic system $\langle f_1 = e_1, \ldots, f_m = e_m\rangle$ can be seen as defining m infinite functor expressions $\bar{e}_1, \ldots, \bar{e}_m$, which are obtained by unrolling the equations ad infinitum.[3] Let $\bar{e}$ be a possibly infinite functor expression and let $t_1, \ldots, t_n$ be infinite trees, then

$$\mathit{poly}\langle [\![\bar{e}]\!] \circ (t_1, \ldots, t_n)\rangle \quad = \quad \mathit{poly}_n\langle \bar{e}\rangle\ (\mathit{poly}\langle t_1\rangle, \ldots, \mathit{poly}\langle t_n\rangle) \tag{4}$$

can be shown using a fixpoint induction on the structure of functor expressions. The proofs are obtained by restructuring the derivations in the text above. The proposition is an immediate consequence of (4) setting $t_i = \Pi_i$ and noting that $[\![f_i]\!] = [\![\bar{e}_i]\!]$ and $\mathit{poly\_k_i\_f_i} = \mathit{poly}_{k_i}\langle \bar{e}_i\rangle$. $\square$

It is worth mentioning that the specialization yields the same result for different representations of the same tree, ie

$$[\![f]\!] = [\![g]\!] \quad \Longrightarrow \quad \mathit{poly}_n\langle f\rangle = \mathit{poly}_n\langle g\rangle \ .$$

This is a simple consequence of Proposition 1. We have, for instance, $\mathit{poly}_n\langle \mathit{Rose}\rangle = \mathit{poly}_n\langle \mathit{Rose}'\rangle$. If we translate $\mathit{poly}_n\langle \mathit{Rose}\rangle$ and $\mathit{poly}_n\langle \mathit{Rose}'\rangle$ into Haskell, we obtain, of course, two different functions since Haskell uses named sums and each datatype must use a different set of constructor names.

To summarize: polytypic functions are defined by induction on the structure of types. The specialization of a polytypic function is defined on the representation of types and as a consequence is inductive on the structure of functor expressions.

4 Polytypic fusion

Polytypic functions satisfy fusion laws, analogous to the fusion law for catamorphisms. The fusion laws are very general: they hold, in fact, for *every* polytypically defined function. This brings about a certain level of abstractness. It is likely that the reader will only fully acknowledge the value of these laws until Section 5 when we take a look at several examples. The first fusion law states conditions under which $\psi \circ \mathit{poly}_n\langle f\rangle\ \boldsymbol{\varphi} = \mathit{poly}'_n\langle f\rangle\ (\psi \circ \boldsymbol{\varphi})$ where $\psi \circ \boldsymbol{\varphi}$ is given by $\psi \circ (\varphi_1, \ldots, \varphi_n) = (\psi \circ \varphi_1, \ldots, \psi \circ \varphi_n)$.

[3] Technically, this means that the algebraic system is interpreted as a *system of regular equations* (Courcelle, 1983, Sec. 4.2) where '$\cdot$' is interpreted as a function symbol.

Proposition 2
Let $poly_n\langle f\rangle$ and $poly'_n\langle f\rangle$ be two polytypic functions and let ψ be a strict function. If $\psi \circ poly_F\ \boldsymbol{\varphi} = poly'_F\ (\psi \circ \boldsymbol{\varphi})$ holds for all $\boldsymbol{\varphi}$ and for all $F \in \Sigma$, then

$$\psi \circ poly_n\langle f\rangle\ \boldsymbol{\varphi} \quad = \quad poly'_n\langle f\rangle\ (\psi \circ \boldsymbol{\varphi}) \quad \text{for all } \boldsymbol{\varphi} \text{ and for all } f.$$

Proof
The proposition is shown by fixpoint induction over the structure of functor expressions. Case $f = \Omega$: holds trivially since ψ is strict. Cases $f = \Pi_i$ and $f = F$:

$$\begin{array}{cl} & \psi \circ poly_n\langle \Pi_i\rangle\ \boldsymbol{\varphi} \\ = & \quad \{\text{ def. } poly_n \} \\ & \psi \circ \varphi_i \\ = & \quad \{\text{ def. } poly'_n \} \\ & poly'_n\langle \Pi_i\rangle\ (\psi \circ \boldsymbol{\varphi}) \end{array} \qquad \begin{array}{cl} & \psi \circ poly_n\langle F\rangle\ \boldsymbol{\varphi} \\ = & \quad \{\text{ def. } poly_n \} \\ & \psi \circ poly_F\ \boldsymbol{\varphi} \\ = & \quad \{\text{ assumption } \} \\ & poly'_F\ (\psi \circ \boldsymbol{\varphi}) \\ = & \quad \{\text{ def. } poly'_n \} \\ & poly'_n\langle F\rangle\ (\psi \circ \boldsymbol{\varphi}) \ . \end{array}$$

Case $f = g \cdot (h_1, \ldots, h_k)$:

$$\begin{array}{cl} & \psi \circ poly_n\langle g \cdot (h_1, \ldots, h_k)\rangle\ \boldsymbol{\varphi} \\ = & \quad \{\text{ def. } poly_n \} \\ & \psi \circ poly_k\langle g\rangle\ (poly_n\langle h_1\rangle\ \boldsymbol{\varphi}, \ldots, poly_n\langle h_k\rangle\ \boldsymbol{\varphi}) \\ = & \quad \{\text{ ex hypothesi } \} \\ & poly'_k\langle g\rangle\ (\psi \circ poly_n\langle h_1\rangle\ \boldsymbol{\varphi}, \ldots, \psi \circ poly_n\langle h_k\rangle\ \boldsymbol{\varphi}) \\ = & \quad \{\text{ ex hypothesi } \} \\ & poly'_k\langle g\rangle\ (poly'_n\langle h_1\rangle\ (\psi \circ \boldsymbol{\varphi}), \ldots, poly'_n\langle h_k\rangle\ (\psi \circ \boldsymbol{\varphi})) \\ = & \quad \{\text{ def. } poly'_n \} \\ & poly'_n\langle g \cdot (h_1, \ldots, h_k)\rangle\ (\psi \circ \boldsymbol{\varphi}) \qquad \square \end{array}$$

To illustrate the first fusion law let us show

$$size\langle f\rangle \quad = \quad length \circ flatten\langle f\rangle \ , \tag{5}$$

where $flatten\langle f\rangle$ is given by

$$\begin{array}{lcl} flatten\langle f\rangle & :: & \forall a. f\ a \to List\ a \\ flatten\langle K\,t\rangle & = & const\ [\,] \\ flatten\langle Id\rangle & = & wrap\ \mathbf{where}\ wrap\ a = [\,a\,] \\ flatten\langle f_1 + f_2\rangle & = & flatten\langle f_1\rangle \triangledown flatten\langle f_2\rangle \\ flatten\langle f_1 \times f_2\rangle & = & (+\!\!+) \circ (flatten\langle f_1\rangle \times flatten\langle f_2\rangle) \ . \end{array}$$

By Proposition 2 we know

$$length \circ flatten_1\langle f\rangle\ wrap \quad = \quad size_1\langle f\rangle\ (length \circ wrap) \ , \tag{6}$$

provided that

$$\begin{array}{rcl} length \circ const\ [\,] & = & const\ 0 \\ length \circ (\varphi_1 \triangledown \varphi_2) & = & (length \circ \varphi_1) \triangledown (length \circ \varphi_2) \\ length \circ (+\!\!+) \circ (\varphi_1 \times \varphi_2) & = & (+) \circ ((length \circ \varphi_1) \times (length \circ \varphi_2)) \ . \end{array}$$

All three conditions hold trivially.[4] Noting that $\mathit{length} \circ \mathit{wrap} = \mathit{const}\ 1$ equation (5) follows immediately from (6).

The second fusion law states conditions under which a computation of two polytypic functions can be fused into a single polytypic function. Setting $(\varphi_1, \ldots, \varphi_n) \circ (\psi_1, \ldots, \psi_n) = (\varphi_1 \circ \psi_1, \ldots, \varphi_n \circ \psi_n)$, we have

Proposition 3

Let $\mathit{poly}_n\langle f\rangle$, $\mathit{poly}'_n\langle f\rangle$, and $\mathit{poly}''_n\langle f\rangle$ be three polytypic functions. If the relation $\mathit{poly}_F\ \boldsymbol{\varphi} \circ \mathit{poly}'_F\ \boldsymbol{\psi} = \mathit{poly}''_F\ (\boldsymbol{\varphi} \circ \boldsymbol{\psi})$ holds for all $\boldsymbol{\varphi}$, $\boldsymbol{\psi}$, and $F \in \Sigma$, then

$$\mathit{poly}_n\langle f\rangle\ \boldsymbol{\varphi} \circ \mathit{poly}'_n\langle f\rangle\ \boldsymbol{\psi} \quad = \quad \mathit{poly}''_n\langle f\rangle\ (\boldsymbol{\varphi} \circ \boldsymbol{\psi}) \quad \text{for all } \boldsymbol{\varphi},\ \boldsymbol{\psi} \text{ and } f.$$

Proof

The proof proceeds again by fixpoint induction over the structure of functor expressions. Case $f = \Omega$: holds trivially. Cases $f = \Pi_i$ and $f = F$:

$$\begin{array}{cl}
 & \mathit{poly}_n\langle \Pi_i\rangle\ \boldsymbol{\varphi} \circ \mathit{poly}'_n\langle \Pi_i\rangle\ \boldsymbol{\psi} \\
= & \quad \{\ \text{def. } \mathit{poly}_n \text{ and } \mathit{poly}'_n\ \} \\
 & \varphi_i \circ \psi_i \\
= & \quad \{\ \text{def. } \mathit{poly}''_n\ \} \\
 & \mathit{poly}''_n\langle \Pi_i\rangle\ (\boldsymbol{\varphi} \circ \boldsymbol{\psi})
\end{array}
\qquad
\begin{array}{cl}
 & \mathit{poly}_n\langle F\rangle\ \boldsymbol{\varphi} \circ \mathit{poly}'_n\langle F\rangle\ \boldsymbol{\psi} \\
= & \quad \{\ \text{def. } \mathit{poly}_n \text{ and } \mathit{poly}'_n\ \} \\
 & \mathit{poly}_F\ \boldsymbol{\varphi} \circ \mathit{poly}'_F\ \boldsymbol{\psi} \\
= & \quad \{\ \text{assumption}\ \} \\
 & \mathit{poly}''_F\ (\boldsymbol{\varphi} \circ \boldsymbol{\psi}) \\
= & \quad \{\ \text{def. } \mathit{poly}''_n\ \} \\
 & \mathit{poly}''_n\langle F\rangle\ (\boldsymbol{\varphi} \circ \boldsymbol{\psi})\ .
\end{array}$$

Case $f = g \cdot (h_1, \ldots, h_k)$:

$$\begin{array}{cl}
 & \mathit{poly}_n\langle g \cdot (h_1, \ldots, h_k)\rangle\ \boldsymbol{\varphi} \circ \mathit{poly}'_n\langle g \cdot (h_1, \ldots, h_k)\rangle\ \boldsymbol{\psi} \\
= & \quad \{\ \text{def. } \mathit{poly}_n \text{ and } \mathit{poly}'_n\ \} \\
 & \mathit{poly}_k\langle g\rangle\ (\mathit{poly}_n\langle h_1\rangle\ \boldsymbol{\varphi}, \ldots, \mathit{poly}_n\langle h_k\rangle\ \boldsymbol{\varphi}) \circ \mathit{poly}'_k\langle g\rangle\ (\mathit{poly}'_n\langle h_1\rangle\ \boldsymbol{\psi}, \ldots, \mathit{poly}'_n\langle h_k\rangle\ \boldsymbol{\psi}) \\
= & \quad \{\ \text{ex hypothesi}\ \} \\
 & \mathit{poly}''_k\langle g\rangle\ (\mathit{poly}_n\langle h_1\rangle\ \boldsymbol{\varphi} \circ \mathit{poly}'_n\langle h_1\rangle\ \boldsymbol{\psi}, \ldots, \mathit{poly}_n\langle h_k\rangle\ \boldsymbol{\varphi} \circ \mathit{poly}'_n\langle h_k\rangle\ \boldsymbol{\psi}) \\
= & \quad \{\ \text{ex hypothesi}\ \} \\
 & \mathit{poly}''_k\langle g\rangle\ (\mathit{poly}''_n\langle h_1\rangle\ (\boldsymbol{\varphi} \circ \boldsymbol{\psi}), \ldots, \mathit{poly}''_n\langle h_k\rangle\ (\boldsymbol{\varphi} \circ \boldsymbol{\psi})) \\
= & \quad \{\ \text{def. } \mathit{poly}''_n\ \} \\
 & \mathit{poly}''_n\langle g \cdot (h_1, \ldots, h_k)\rangle\ (\boldsymbol{\varphi} \circ \boldsymbol{\psi}) \qquad \Box
\end{array}$$

Applications of the second fusion law will be given in the following section.

To summarize: polytypic functions satisfy two general fusion laws, which state conditions for fusing a monotypic function with a polytypic function and for fusing two polytypic functions.

[4] The relevant laws are listed in Appendix A.

5 Examples

5.1 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.[5]

$$\begin{array}{lcl}
\mathit{fmap}\langle f\rangle & :: & \forall a.\forall b.(a \to b) \to (f\ a \to f\ b)\\
\mathit{fmap}\langle f\rangle\ \varphi & = & \mathit{map}\langle f\rangle\\
\quad\textbf{where}\ \mathit{map}\langle f\rangle & :: & f\ a \to f\ b\\
\qquad\quad \mathit{map}\langle K t\rangle & = & \mathit{id}\\
\qquad\quad \mathit{map}\langle \mathit{Id}\rangle & = & \varphi\\
\qquad\quad \mathit{map}\langle f_1 + f_2\rangle & = & \mathit{map}\langle f_1\rangle + \mathit{map}\langle f_2\rangle\\
\qquad\quad \mathit{map}\langle f_1 \times f_2\rangle & = & \mathit{map}\langle f_1\rangle \times \mathit{map}\langle f_2\rangle
\end{array}$$

It is revealing to consider the typings of the subsidiary functions $\mathit{map}_n\langle f\rangle$.

$$\begin{array}{lcl}
\mathit{map}_n\langle f\rangle & :: & (g_1\ a \to g_1\ b, \ldots, g_n\ a \to g_n\ b)\\
& \to & (f\ (g_1\ a)\ \ldots\ (g_n\ a) \to f\ (g_1\ b)\ \ldots\ (g_n\ b))
\end{array}$$

Setting $g_i = \mathit{Id}$ we obtain the typings of n-ary mapping functions. And, in fact, $\mathit{map}_n\langle f\rangle$ corresponds to the mapping function of the n-ary functor f. Thus, to define $\mathit{fmap}\langle f\rangle$ 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.

This example shows that the subsidiary functions $\mathit{poly}_n\langle f\rangle$ are often interesting in their own right. Since the functions $\mathit{map}_n\langle f\rangle$ do not depend on the parameter φ, we can even turn them into global functions and define, for instance,

$$\mathit{bmap}\langle f\rangle\ \varphi_1\ \varphi_2 \quad = \quad \mathit{map}_2\langle f\rangle\ (\varphi_1, \varphi_2)\ .$$

The mapping function of an n-ary functor is required to satisfy the following two functorial properties.

$$\begin{array}{rcl}
\mathit{map}_n\langle f\rangle\ \boldsymbol{id} & = & \mathit{id}\\
\mathit{map}_n\langle f\rangle\ (\boldsymbol{\varphi} \circ \boldsymbol{\psi}) & = & \mathit{map}_n\langle f\rangle\ \boldsymbol{\varphi} \circ \mathit{map}_n\langle f\rangle\ \boldsymbol{\psi}
\end{array}$$

It is relatively easy to see and to prove that the first property holds. The second property is an instance of the second polytypic fusion law. Setting $\mathit{poly}_n\langle f\rangle = \mathit{poly}'_n\langle f\rangle = \mathit{poly}''_n\langle f\rangle = \mathit{map}_n\langle f\rangle$, we must show $\mathit{map}_F\ \boldsymbol{\varphi} \circ \mathit{map}_F\ \boldsymbol{\psi} = \mathit{map}_F\ (\boldsymbol{\varphi} \circ \boldsymbol{\psi})$ for $\mathit{map}_{K\,t} = \mathit{Id}$, $\mathit{map}_+(\varphi_1, \varphi_2) = \varphi_1 + \varphi_2$, and $\mathit{map}_\times(\varphi_1, \varphi_2) = \varphi_1 \times \varphi_2$. All of these conditions hold.

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

$$\begin{array}{lcl}
sum\langle f\rangle & :: & (Num\ n) \Rightarrow f\ n \rightarrow n \\
sum\langle f\rangle & = & reduce\langle f\rangle\ 0\ (+) \\
and\langle f\rangle & :: & f\ Bool \rightarrow Bool \\
and\langle f\rangle & = & reduce\langle f\rangle\ True\ (\wedge) \\
minimum\langle f\rangle & :: & (Bounded\ a, Ord\ a) \Rightarrow f\ a \rightarrow a \\
minimum\langle f\rangle & = & reduce\langle f\rangle\ maxBound\ min \\
size\langle f\rangle & :: & (Num\ n) \Rightarrow f\ a \rightarrow n \\
size\langle f\rangle & = & sum\langle f\rangle \circ fmap\langle f\rangle\ (const\ 1) \\
all\langle f\rangle & :: & (a \rightarrow Bool) \rightarrow (f\ a \rightarrow Bool) \\
all\langle f\rangle\ p & = & and\langle f\rangle \circ fmap\langle f\rangle\ p \\
flatten\langle f\rangle & :: & f\ a \rightarrow List\ a \\
flatten\langle f\rangle & = & reduce\langle f\rangle\ [\,]\ (+\!\!+) \circ fmap\langle f\rangle\ wrap\ \mathbf{where}\ wrap\ a = [a]
\end{array}$$

Fig. 2. Examples for reductions.

5.2 Polytypic reduction

The functions *size* and *sum* are instances of a more general concept termed *reduction* (Meertens, 1996). A reduction is a function of type $f\ a \rightarrow a$, which collapses a structure of values of type a into a single value of type a. To define a reduction we require two ingredients: a value $e :: a$ and a binary operation $op :: \Delta\ a \rightarrow a$. Usually but not necessarily e is the neutral element of op.

$$\begin{array}{lcl}
reduce\langle f\rangle & :: & \forall a.a \rightarrow (\Delta\ a \rightarrow a) \rightarrow f\ a \rightarrow a \\
reduce\langle f\rangle\ e\ op & = & red\langle f\rangle \\
\quad \mathbf{where}\ red\langle f\rangle & :: & f\ a \rightarrow a \\
\qquad red\langle K\,t\rangle & = & const\ e \\
\qquad red\langle Id\rangle & = & id \\
\qquad red\langle f_1 + f_2\rangle & = & red\langle f_1\rangle \triangledown red\langle f_2\rangle \\
\qquad red\langle f_1 \times f_2\rangle & = & op \circ (red\langle f_1\rangle \times red\langle f_2\rangle)
\end{array}$$

A number of useful functions can be defined in terms of $reduce\langle f\rangle$ and $fmap\langle f\rangle$, see Figure 2. Further examples can be found, for instance, in (Meertens, 1996) or (Jansson & Jeuring, 1998b).

Again, it is interesting to inspect the subsidiary functions $reduce_n\langle f\rangle$ more closely. The following property

$$reduce_n\langle f\rangle\ \varphi\ e\ op \quad = \quad reduce_n\langle f\rangle\ \boldsymbol{id}\ e\ op \circ map_n\langle f\rangle\ \varphi\ ,$$

which is an immediate consequence of Proposition 3, shows that $reduce_n\langle f\rangle$ combines a reduction with a map. This idiom is, in fact, a very old one. It appears, for instance, in (Bird, 1988) where it is shown that each homomorphism on lists (ie each catamorphism) can be expressed in this form. In a sense, $reduce_n\langle f\rangle$ can be seen as a generalization of that scheme.

Specializing Proposition 2 gives an important fusion law for reductions. Let ψ be a strict function, then

$$\begin{array}{ll}
& \psi \circ reduce_n\langle f\rangle\ \varphi\ e\ op = reduce_n\langle f\rangle\ (\psi \circ \varphi)\ e'\ op' \\
\Longleftarrow & \psi \circ const\ e = const\ e' \quad \wedge \quad \psi \circ op = op' \circ (\psi \times \psi)\ .
\end{array}$$

This law can be used to derive the efficient implementation of $size\langle Perfect\rangle$ shown in Section 1 from the polytypic definition, see (Hinze, 1999).

The implementation of $flatten\langle f\rangle$ given in Figure 2 has a quadratic running time since the computation of $x +\!\!+\, y$ takes time proportional to the length of x. Using the well-known technique of *accumulation* (Bird, 1998) we can improve the running time to $O(n)$. The basic idea is to define a function $redr\langle f\rangle$ such that

$$op\ (red\langle f\rangle\ x, a) \quad = \quad redr\langle f\rangle\ x\ a\ \ .$$

In a point-free style this condition can be written more succinctly as

$$curry\ op \circ red\langle f\rangle \quad = \quad redr\langle f\rangle\ \ ,$$

which suggests to apply Proposition 2 for deriving a definition of $redr\langle f\rangle$. The calculation, which is left as an exercise to the reader, shows that the following definition improves $reduce\langle f\rangle$ provided op is associative and e is the neutral element of op.

$$\begin{array}{lcl}
reduce'\langle f\rangle & :: & \forall a.a \to (\Delta\ a \to a) \to f\ a \to a \\
reduce'\langle f\rangle\ e\ op\ x & = & redr\langle f\rangle\ x\ e \\
\quad \textbf{where}\ redr\langle f\rangle & :: & f\ a \to (a \to a) \\
\qquad\qquad redr\langle K\,t\rangle & = & const\ id \\
\qquad\qquad redr\langle Id\rangle & = & curry\ op \\
\qquad\qquad redr\langle f_1 + f_2\rangle & = & redr\langle f_1\rangle \triangledown redr\langle f_2\rangle \\
\qquad\qquad redr\langle f_1 \times f_2\rangle & = & (\circ) \circ (redr\langle f_1\rangle \times redr\langle f_2\rangle)
\end{array}$$

The implementation guarantees that applications of op are only nested to the right as in $op\ (a_1, op\ (a_2, \ldots\ op\ (a_n, e)\ \ldots))$—hence the name of the auxiliary function. This property also reveals that the type of $reduce'\langle f\rangle$ is unnecessarily restricted: the two arguments of op need not have the same type. Therefore, we may generalize the type signature as follows—additionally, we assume that op is curried and rearrange the arguments to emphasize the structure.

$$\begin{array}{lcl}
reducer\langle f\rangle & :: & \forall a.\forall b.(a \to (b \to b)) \to (f\ a \to (b \to b)) \\
reducer\langle f\rangle\ op & = & redr\langle f\rangle \\
\quad \textbf{where}\ redr\langle f\rangle & :: & f\ a \to (b \to b) \\
\qquad\qquad redr\langle K\,t\rangle & = & const\ id \\
\qquad\qquad redr\langle Id\rangle & = & op \\
\qquad\qquad redr\langle f_1 + f_2\rangle & = & redr\langle f_1\rangle \triangledown redr\langle f_2\rangle \\
\qquad\qquad redr\langle f_1 \times f_2\rangle & = & (\circ) \circ (redr\langle f_1\rangle \times redr\langle f_2\rangle)
\end{array}$$

Building upon $reducer\langle f\rangle$ we can now give a linear-time program for $flatten\langle f\rangle$.

$$\begin{array}{lcl}
flatten\langle f\rangle & :: & f\ a \to List\ a \\
flatten\langle f\rangle\ x & = & reducer\langle f\rangle\ (curry\ (:))\ x\ [\,]
\end{array}$$

5.3 Polytypic equality

The equality function is a nice example for a function which is indexed by a nullary functor. In Haskell it can be automatically derived for arbitrary datatypes, and the following can be seen as a formalization of that process. We assume that suitable equality functions for *Int*, *Double* and *Char* are predefined.

$$\begin{array}{lcl}
eq\langle f\rangle & :: & f \times f \to Bool \\
eq\langle 1\rangle & = & const\ True \\
eq\langle Int\rangle & = & eqInt \\
eq\langle Double\rangle & = & eqDouble \\
eq\langle Char\rangle & = & eqChar \\
eq\langle f_1 + f_2\rangle & = & ((eq\langle f_1\rangle \triangledown const\ False) \triangledown (const\ False \triangledown eq\langle f_2\rangle)) \circ dist \\
eq\langle f_1 \times f_2\rangle & = & (\wedge) \circ (eq\langle f_1\rangle \times eq\langle f_2\rangle) \circ trans
\end{array}$$

The auxiliary function *dist*, defined in Appendix A, applies the distributive laws to map an element of $(a + b) \times (c + d)$ to $(a \times c + a \times d) + (b \times c + b \times d)$, ie $(inl\ a, inr\ d)$ becomes $inl\ (inr\ (a, d))$. The function *trans* transposes a 2×2 matrix, ie $((a, b), (c, d))$ becomes $((a, c), (b, d))$.

5.4 Polytypic comparison

Varying the definition of $eq\langle f\rangle$ slightly we can also realize Haskell's *compare* function which allows to determine the precise ordering of elements with a single comparison.

$$\begin{array}{lcl}
cmp\langle f\rangle & :: & f \times f \to Ordering \\
cmp\langle 1\rangle & = & const\ EQ \\
cmp\langle Int\rangle & = & cmpInt \\
cmp\langle Double\rangle & = & cmpDouble \\
cmp\langle Char\rangle & = & cmpChar \\
cmp\langle f_1 + f_2\rangle & = & ((cmp\langle f_1\rangle \triangledown const\ LT) \triangledown (const\ GT \triangledown cmp\langle f_2\rangle)) \circ dist \\
cmp\langle f_1 \times f_2\rangle & = & lex \circ (cmp\langle f_1\rangle \times cmp\langle f_2\rangle) \circ trans
\end{array}$$

The helper function *lex* used in the last equation implements the lexicographic ordering of two relations.

$$\begin{array}{lcl}
lex & :: & Ordering \times Ordering \to Ordering \\
lex\ LT\ rel & = & LT \\
lex\ EQ\ rel & = & rel \\
lex\ GT\ rel & = & GT
\end{array}$$

6 Related and future work

This paper can be regarded as a successor to (Hinze, 1999), where a similar approach to polytypic programming using rational trees is presented. The major difference between the two frameworks lies in the treatment of functor composition. In (Hinze, 1999) the composition of functors is regarded as a function symbol which implies that a polytypic definition must include cases for specifying the action on $f \cdot (g_1, \ldots, g_k)$ for each $k \geqslant 1$. 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 (Jansson & Jeuring, 1997) is based on the initial algebra semantics of datatypes. Here, functors are modeled by the following grammar.

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

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 $\mathit{size}\langle f\rangle$ is defined as follows—modulo change of notation.

$$\begin{array}{lcl} \mathit{bsize}\langle b\rangle & :: & \forall a.b\ a\ \mathit{Int} \to \mathit{Int} \\ \mathit{bsize}\langle K\,t\rangle & = & \mathit{const}\ 0 \\ \mathit{bsize}\langle \mathit{Fst}\rangle & = & \mathit{const}\ 1 \\ \mathit{bsize}\langle \mathit{Snd}\rangle & = & \mathit{id} \\ \mathit{bsize}\langle b_1 + b_2\rangle & = & \mathit{bsize}\langle b_1\rangle \triangledown \mathit{bsize}\langle b_2\rangle \\ \mathit{bsize}\langle b_1 \times b_2\rangle & = & (+) \circ (\mathit{bsize}\langle b_1\rangle \times \mathit{bsize}\langle b_2\rangle) \\ \mathit{bsize}\langle f \circ b\rangle & = & \mathit{sum}\langle f\rangle \circ \mathit{fmap}\langle f\rangle\ (\mathit{bsize}\langle b\rangle) \\ \mathit{size}\langle f\rangle & :: & \forall a.f\ a \to \mathit{Int} \\ \mathit{size}\langle \mu b\rangle & = & \mathit{cata}\langle \mu b\rangle\ (\mathit{bsize}\langle b\rangle) \end{array}$$

The program is quite elaborate as compared to the one given in Section 1: it involves two general combining forms, $\mathit{fmap}\langle f\rangle$ and $\mathit{cata}\langle f\rangle$, and an auxiliary polytypic function, $\mathit{sum}\langle f\rangle$. The disadvantages of the initial algebra approach are fairly obvious. The definition is redundant: $\mathit{size}\langle f\rangle$ is uniquely defined by its action on constant functors, Id, sums, and products. The definition is incomplete: $\mathit{size}\langle f\rangle$ is only applicable to regular functors. 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 $\mathit{fmap}\langle f\rangle$ produces an intermediate data structure which is immediately consumed by $\mathit{sum}\langle f\rangle$. In other words, $\mathit{size}\langle \mathit{Rose}\rangle$ 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 (Meijer *et al.*, 1991). As an example, the recursion scheme $\mathit{cata}\langle f\rangle$, which is used in $\mathit{size}\langle f\rangle$, is given by

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

The operation $(\text{-})'$ (called *FunctorOf* in PolyP) maps a type functor to its base functor, ie $F' = B$ for $F = \mu B$. The function $\mathit{out} :: f\ a \to f'\ a\ (f\ a)$ 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 listed in (Jansson & Jeuring, 1998b) can be implemented in our framework, except, of course, cata- and anamorphisms—it is indispensable for others. The polytypic unification algorithm described in (Jansson & Jeuring, 1998a), for instance, requires a function which determines the immediate subterms of a term. A similar function is needed in (Bird *et al.*, 1996) where a generalization of the maximum segment sum problem is presented. In both cases the recursive structure of a datatype must be known. Alas, in our framework type recursion is left implicit which suggests that we cannot deal with applications like these. Fortunately, nothing prevents us from incorporating the operator $(\text{-})'$ depicted above. Let us furthermore generalize $(\text{-})'$ so that it maps an n-ary, regular functor to its associated $(n+1)$-ary base functor. Given two generic 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 is the definition of cata- and anamorphisms for unary functors.

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

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. The function which collects the immediate subterms of a term can be defined as follows.

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

The operator $(\text{-})'$ is not the only useful operation on functors. In (Meertens, 1996) a diagonalization operator $(\text{-})^*$ is introduced which maps the n-ary functor F to the unary functor $F \cdot (Id, \ldots, Id)$. This operator allows to generalize reductions to n-ary functors. We simply assign $reduce\langle f\rangle$ the type $\forall a.a \to (\Delta\ a \to a) \to f^*\ a \to a$ and replace the equation $red\langle Id\rangle = id$ by the more general $red\langle \Pi_i\rangle = id$.

Directions for future work suggest themselves. It remains to broaden the approach to include exponentials and higher-order polymorphism (Jones, 1995). The former extension should be fairly straightforward, see (Sheard & Fegaras, 1993) or (Meijer & Hutton, 1995). Currently, the author is working on the latter extension which allows to define polytypic functions generically for all datatypes expressible in Haskell. The main idea is to model higher-order types by (possibly infinite) expressions of the simply typed λ-calculus (note that the types of type expressions are known as *kinds*). An application of polytypic programming to digital searching is described in a companion paper (Hinze, 1998), where we show how to define *tries* and operations on tries generically for arbitrary first-order polymorphic datatypes. The central insight is that a trie can be considered as a *type-indexed datatype*, which adds an interesting new dimension to polytypic programming.

References

Bird, Richard. (1998). *Introduction to functional programming using Haskell*. 2nd edn. London: Prentice Hall Europe.

Bird, Richard, & de Moor, Oege. (1997). *Algebra of programming*. London: Prentice Hall Europe.

Bird, Richard, & Meertens, Lambert. (1998). Nested datatypes. *Pages 52–67 of:* Jeuring, J. (ed), *Fourth international conference on mathematics of program construction, MPC'98, Marstrand, Sweden*. Lecture Notes in Computer Science, vol. 1422. Springer Verlag.

Bird, Richard, de Moor, Oege, & Hoogendijk, Paul. (1996). Generic functional programming with types and relations. *Journal of functional programming*, **6**(1), 1–28.

Bird, Richard S. (1988). Lectures on constructive functional programming. Broy, Manfred (ed), *Constructive methods in computer science*. Springer Verlag.

Courcelle, Bruno. (1983). Fundamental properties of infinite trees. *Theoretical computer science*, **25**(2), 95–169.

Hinze, Ralf. 1998 (November). *Generalizing generalized tries*. Tech. rept. IAI-TR-98-11. Institut für Informatik III, Universität Bonn.

Hinze, Ralf. 1999 (March). Polytypic functions over nested datatypes (extended abstract). *3rd latin-american conference on functional programming (CLaPF'99)*. To appear.

Jansson, Patrik, & Jeuring, Johan. (1997). PolyP—a polytypic programming language extension. *Pages 470–482 of: Conf. record 24th ACM SIGPLAN-SIGACT symp. on principles of programming languages, POPL'97, Paris, France*. New York: ACM Press.

Jansson, Patrik, & Jeuring, Johan. (1998a). Functional pearl: Polytypic unification. *Journal of functional programming*, **8**(5).

Jansson, Patrik, & Jeuring, Johan. (1998b). PolyLib—A library of polytypic functions. Backhouse, Roland, & Sheard, Tim (eds), *Informal proceedings workshop on generic programming, WGP'98, Marstrand, Sweden, 18 june 1998*. Dept. of Computing Science, Chalmers Univ. of Techn. and Göteborg Univ.

Jeuring, Johan, & Jansson, Patrik. (1996). Polytypic programming. *Pages 68–114 of:* Launchbury, J., Meijer, E., & Sheard, T. (eds), *Tutorial text 2nd international school on advanced functional programming, Olympia, WA, USA*. Lecture Notes in Computer Science, vol. 1129. Springer Verlag.

Jones, Mark P. (1995). Functional programming with overloading and higher-order polymorphism. *Pages 97–136 of: First international spring school on advanced functional programming techniques*. Lecture Notes in Computer Science, vol. 925. Springer Verlag.

McCracken, Nancy Jean. (1984). The typechecking of programs with implicit type structure. *Pages 301–315 of:* Kahn, Gilles, MacQueen, David B., & Plotkin, Gordon D. (eds), *Semantics of data types: International symposium, Sophia-Antipolis, France*. Lecture Notes in Computer Science, vol. 173. Berlin: Springer Verlag.

Meertens, Lambert. (1996). Calculate polytypically! *Pages 1–16 of:* Kuchen, H., & Swiestra, S.D. (eds), *Proceedings 8th intl. symp. on programming languages: Implementations, logics, and programs, PLILP'96, Aachen, Germany, 24–27 sept 1996*. Lecture Notes in Computer Science, vol. 1140. Springer Verlag.

Meijer, E., Fokkinga, M., & Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. *Pages 124–144 of: 5th ACM conference on functional programming languages and computer architecture, Cambridge, MA*. Lecture Notes in Computer Science, vol. 523. Springer Verlag.

Meijer, Erik, & Hutton, Graham. (1995). Bananas in space: Extending fold and unfold to exponential types. *Pages 324–333 of: Conf. record 7th ACM SIGPLAN/SIGARCH* *and IFIP WG 2.8 intl. conf. on functional programming languages and computer architecture, FPCA'95, La Jolla, San Diego, CA, USA*. ACM Press.

Mycroft, Alan. (1984). Polymorphic type schemes and recursive definitions. Paul, M., & Robinet, B. (eds), *International symposium on programming, 6th colloquium Toulouse*. LNCS 167.

Okasaki, Chris. (1998). *Purely functional data structures*. Cambridge University Press.

Peyton Jones [editor], Simon, Hughes [editor], John, Augustsson, Lennart, Barton, Dave, Boutel, Brian, Burton, Warren, Fraser, Simon, Fasel, Joseph, Hammond, Kevin, Hinze, Ralf, Hudak, Paul, Johnsson, Thomas, Jones, Mark, Launchbury, John, Meijer, Erik, Peterson, John, Reid, Alastair, Runciman, Colin, & Wadler, Philip. 1998 (December). *Haskell 98 — A non-strict, purely functional language*.

Sénizergues, Géraud. (1997). The equivalence problem for deterministic pushdown automata is decidable. *Pages 671–681 of:* Degano, Pierpaolo, Gorrieri, Robert, & Marchetti-Spaccamela, Alberto (eds), *Automata, languages and programming, 24th international colloquium*. Lecture Notes in Computer Science, vol. 1256. Bologna, Italy: Springer Verlag.

Sheard, Tim, & Fegaras, Leonidas. (1993). A fold for all seasons. *Pages 233–242 of: Proceedings 6th ACM SIGPLAN/SIGARCH intl. conf. on functional programming languages and computer architecture, FPCA'93, Copenhagen, Denmark, 9–11 june 1993*. New York: ACM Press.

A Auxiliary functions

This appendix contains the definitions of the basic categorical combinators used in the main text. The notation is fairly standard so the cognoscenti should have no problems in reading the paper. For background material the reader is referred to the textbook by R. Bird and O. de Moor (1997).

Sums We treat disjoint sums or coproducts as if they were given by the following datatype definition.

$$\begin{aligned}
\mathbf{data}\ a + b &= \mathit{inl}\ a \mid \mathit{inr}\ b \\
(f_1 \triangledown f_2)\ (\mathit{inl}\ a_1) &= f_1\ a_1 \\
(f_1 \triangledown f_2)\ (\mathit{inr}\ a_2) &= f_2\ a_2 \\
f_1 + f_2 &= \mathit{inl} \circ f_1 \triangledown \mathit{inr} \circ f_2
\end{aligned}$$

Elements of $a + b$ are synthesized using the injection function *inl* and *inr*; they are analysed using the 'case' operator $(\triangledown)$. The disjoint sum is a bifunctor; its mapping function is given by $(+)$. The operators $(\triangledown)$ and $(+)$ satisfy a variety of properties, among others

$$\begin{aligned}
h \circ (f \triangledown g) &= (h \circ f) \triangledown (h \circ g) \\
(f + g) \circ (h + k) &= (f \circ h) + (g \circ k) \ .
\end{aligned}$$

Products Products are given by the following datatype.

$$\begin{aligned}
&\mathbf{data}\ a \times b &&= (a, b)\\
&\mathit{outl}\ (a, b) &&= a\\
&\mathit{outr}\ (a, b) &&= b\\
&(f_1 \vartriangle f_2)\ a &&= (f_1\ a, f_2\ a)\\
&f_1 \times f_2 &&= f_1 \circ \mathit{outl} \vartriangle f_2 \circ \mathit{outr}
\end{aligned}$$

Elements of $a \times b$ are analysed using the projection functions *outl* and *outr*; they are synthesized using the 'split' operator $(\vartriangle)$. The product is a bifunctor, as well; its mapping function is given by $(\times)$. The properties of $(\vartriangle)$ and $(\times)$ are dual to those of $(\triangledown)$ and $(+)$.

$$\begin{aligned}
(f \vartriangle g) \circ h &= (f \circ h) \vartriangle (g \circ h)\\
(f \times g) \circ (h \times k) &= (f \circ h) \times (g \circ k)\ .
\end{aligned}$$

We require one further operation which transposes a 2×2 matrix.

$$\begin{aligned}
&\mathit{trans} &&:: (a \times b) \times (c \times d) \to (a \times c) \times (b \times d)\\
&\mathit{trans}\ ((a, b), (c, d)) &&= ((a, c), (b, d))
\end{aligned}$$

Distributivity The function *distr* distributes $(\times)$ rightward through $(+)$; leftward distribution is denoted by *distl*.

$$\begin{aligned}
&\mathit{distr} &&:: a \times (b + c) \to a \times b + a \times c\\
&\mathit{distr}\ (a, \mathit{inl}\ b) &&= \mathit{inl}\ (a, b)\\
&\mathit{distr}\ (a, \mathit{inr}\ c) &&= \mathit{inr}\ (a, c)
\end{aligned}$$

Finally, *dist* combines *distr* and *distl*.

$$\begin{aligned}
&\mathit{dist} &&:: (a + b) \times (c + d) \to (a \times c + a \times d) + (b \times c + b \times d)\\
&\mathit{dist} &&= (\mathit{distr} + \mathit{distr}) \circ \mathit{distl}
\end{aligned}$$