Polytypic Functions Over Nested Datatypes (Extended Abstract)

Ralf Hinze

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

Abstract. The theory and practice of polytypic programming is intimately connected with the initial algebra semantics of datatypes. This is both a blessing and a curse. It is a blessing because the underlying theory is beautiful and well developed. It is a curse because the initial algebra semantics is restricted to so-called regular datatypes. Recent work by R. Bird and L. Meertens [1] on the semantics of non-regular or nested datatypes suggests that an extension to general datatypes is not entirely straightforward. Here we propose an alternative which extends polytypism to arbitrary datatypes, including nested datatypes and mutually recursive datatypes. The central idea is to use rational trees over a suitable set of functor symbols as index sets for polytypic functions. Besides covering a wider range of types the approach is also simpler and technically less involving than previous ones. The prize one has to pay is a certain loss in expressiveness: general recursion schemes such as *fold* and friends can no longer be defined.

1 Introduction

A polytypic function is a function 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 polymorphic type and it is usually a tiresome routine matter to do so. A polytypic programming language allows 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 programming is usually based on the initial algebra semantics of datatypes. To illustrate, consider the following definition of polymorphic lists.

$$\mathbf{data}\ List\ a = 1 + a \times List\ a$$

The meaning of $List\ a$ is given by the initial F_a-algebra of the functor $F_a(b) = F(a, b) = 1 + a \times b$. The initial algebra can be seen as the least solution of the equation $List\ a = F(a, List\ a)$. The initial algebra approach, however, has its limitations. It fails, for example, to give a meaning to so-called non-regular or nested datatypes such as

$$\mathbf{data}\ Perfect\ a = a + Perfect\ (a \times a)\ .$$

Since the recursive call of *Perfect* on the right-hand side is not a copy of the declared type on the left-hand side, the equation cannot be rewritten into the form $Perfect\ a = G(a, Perfect\ a)$ for a some G.

A way out of this dilemma is to consider initial algebras of higher-order functors [1]. If we lift coproduct $(+)$ and product $(\times)$ to functors, $(F_1 + F_2)\ T = F_1\ T + F_2\ T$ and $(F_1 \times F_2)\ T = F_1\ T \times F_2\ T$, the above type definitions can be rewritten as functor equations:

$$\begin{aligned} List &= K1 + Id \times List\ , \\ Perfect &= Id + Perfect \cdot (Id \times Id)\ , \end{aligned}$$

where $K\,T$ is the constant functor, Id is the identity functor and $(\cdot)$ denotes functor composition. The meaning of *List* and *Perfect* can now be defined as the initial algebra of the associated higher-order functor. While this approach shares some of the elegance and ease of the first-order approach, it also has its drawbacks. It appears that the associated *fold*-operations which are at the heart of polytypic programming are limited in expressibility. The crucial point is that *fold* operates on natural transformations, ie it takes polymorphic functions to polymorphic functions. For instance, the *fold* operator on *Perfect* takes a function of type $\forall a.\, a + f(a \times a) \to f\ a$ to a function of type $\forall a.Perfect\ a \to f\ a$ for some fixed f. This is, however, too restrictive for many applications, for instance, for summing up a tree of integers. Therefore, it is at least questionable whether initial algebras of higher-order functors can serve as a viable basis for polytypism on nested datatypes.

Back to the first-order case. In a polytypic language the function $size :: f\ a \to Int$ is parametrised by the functor f and is defined by induction on the structure of f. As an aside, note that *size* is really a *polyfunctorial*, not a polytypic function. Inspired by the initial algebra semantics the structure of functors is usually given by the following grammar, see [11] or [9].

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

The non-terminal B generates the language of bifunctors. By μB we denote the unary functor F given as the least solution of the equation $F\ a = B(a, F\ a)$. The functor $F = \mu B$ is also known as a *type functor*. This representation entails that a non-recursive, polymorphic functor F is modelled by μB with $B(a, b) = F\ a$, whereas a non-polymorphic, recursive datatype induced by a functor F is modelled by μB with $B(a, b) = F\ b$. Thus, $\mu(K1 + Fst)$ encodes the datatype *Maybe* and $\mu(K1 + Snd)$ encodes the datatype of Peano numerals.

One can reasonably argue that the definition of unary functors via mimimalization of bifunctors is not the most direct way to go. An alternative which we explore in this paper is to define functor expressions as *rational trees* [4] over the following grammar of unary functors.

$$F = K\,T \mid Id \mid F + F \mid F \times F \mid F \cdot F$$

This approach is, of course, inspired by the way types are introduced in most functional programming languages. Type definitions usually take the form of *recursion equations*. If we interpret these equations purely syntactically, each equation defines a unique rational tree. Recall that a rational (or regular) tree is an infinite tree which has only a finite number of subtrees. Fig. 1 displays the rational trees defined by the type equations above. Note that the pictorial

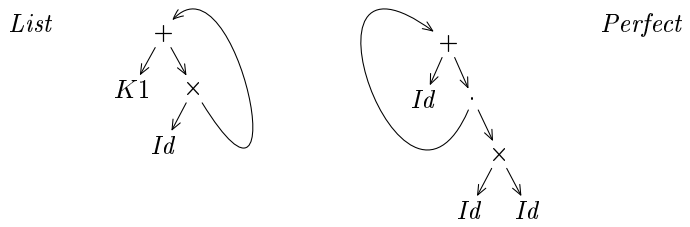

Fig. 1. Types interpreted as rational trees.

representation already employs the defining property of rational trees. Since the set of subtrees is finite, we can always draw a rational tree as a finite tree with additional back or cross edges.

The major difference to previous approaches lies in the use of infinite instead of finite trees as index sets for polytypic functions. This change is not as problematic as one might think at first sight. For instance, we can still define polytypic functions recursively and prove polytypic properties inductively. Of course, to make things work we must impose the usual conditions: functions must be continuous and properties must be chain complete. In essence this means that the class of functors is itself modelled by a *non-strict* datatype as opposed to an inductive datatype. We could even consider functors given by arbitrary infinite trees. The focus on rational trees is, however, necessary for practical considerations. It guarantees that the specialization of polytypic functions always terminates.

The rest of this paper is organized as follows. Sec. 2 introduces rational trees and functor expressions. Sec. 3 and Sec. 4 give several examples for polyfunctorial functions, among others a polytypic *map* and polytypic reduction. We will see that nested datatypes add an interesting new dimension to polytypic programming. For instance, efficiency becomes a concern: a straightforward implementation of *size* has a quadratic running time for some nested datatypes. Improving its running time makes a nice example in polytypic program derivation. Finally, Sec. 5 reviews related work and points out directions for future work.

2 Functor expressions

There are several ways to represented rational trees, see, for instance, [4, Sec. 4]. The most obvious and, in fact, the most convenient way is to use systems of

recursion equations. Let $\Sigma = \bigcup_{n \geqslant 0} \Sigma_n$ be a set of function symbols and X be a set of variables, we denote by $T(\Sigma, X)$ the set of first-order terms over Σ and X. A *system of recursion equations over* Σ is of the form $\{x_1 = t_1, \ldots, x_n = t_n\}$ with $t_i \in T(\Sigma, X)$, $X = \{x_1, \ldots, x_n\}$, and $x_i \neq x_j$ for $i \neq j$. The elements of X are termed *recursion variables*.

A system of recursion equations is in *canonical form* if the right-hand side of each equation is of the form $f(x_1, \ldots, x_n)$ where $f \in \Sigma_n$ is an n-ary function symbol and the x_i are recursion variables. Each system can be transformed into an equivalent system which is in canonical form. We tacitly assume that an equation of the form $x = x$ is replaced by $x = \Omega$ where Ω is a new nullary function symbol representing the 'bottom' tree.

It is well-known that each canonical system of recursion equations has a unique solution in the realm of rational trees. An unrestricted system has in general only a *least* solution. Each tree is, for instance, a solution of the trivial system $\{x = x\}$.

Now, a *functor tree* is a rational tree over the following set of 'functor' symbols

$$F = K\,T \mid Id \mid F + F \mid F \times F \mid F \cdot F \quad ,$$

ie $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_0 = \{\, K\,t \mid t \in T \,\} \cup \{Id\}$ and $\Sigma_2 = \{+, \times, \cdot\}$.

Let us consider some examples. Finite functor trees correspond to so-called *polynomial* functors.

$$\begin{aligned} \Delta &= Id \times Id \\ Maybe &= K1 + Id \end{aligned}$$

The functor Δ is called the diagonal or square functor, $Maybe\ a$ is the type of optional values of type a.

Let us call a functor composition $F \cdot G$ *non-trivial* if neither F nor G equals the identity functor.[1] Possibly infinite functor trees where cycles only pass through the right-hand branches of non-trivial functor compositions are known as *regular* functors.

$$\begin{aligned} List &= K1 + Id \times List \\ Tree &= Id + \Delta \cdot Tree \\ Rose &= Id \times List \cdot Rose \end{aligned}$$

The functor *List* corresponds to the ubiquitous datatype of polymorphic lists, *Tree* encompasses polymorphic, binary leaf trees and *Rose* is the datatype of multi-way branching trees.

If cycles pass through the left-hand branches of non-trivial functor compositions (or through both branches) we have *non-regular* functors.

$$\begin{aligned} Perfect &= Id + Perfect \cdot \Delta \\ Nest &= K1 + Id \times (Nest \cdot \Delta) \\ Bush &= K1 + Id \times (Bush \cdot Bush) \end{aligned}$$

[1] By 'equal' we mean semantic equality of functors, see below.

The meaning of these functor expressions becomes more intelligible if we unroll the definition a few steps and massage the result using the following laws.

$$K\,T \cdot F = K\,T \tag{1}$$
$$Id \cdot F = F \tag{2}$$
$$F \cdot Id = F \tag{3}$$
$$(F_1 + F_2) \cdot G = F_1 \cdot G + F_2 \cdot G \tag{4}$$
$$(F_1 \times F_2) \cdot G = F_1 \cdot G \times F_2 \cdot G \tag{5}$$
$$(F_1 \cdot F_2) \cdot F_3 = F_1 \cdot (F_2 \cdot F_3) \tag{6}$$

Unrolling *Perfect* yields

$$Perfect = Id + (\Delta + (\Delta^2 + (\Delta^3 + \cdots)))$$

where F^n is given by $F^0 = Id$ and $F^{n+1} = F \cdot F^n$. If we interpret Δ^n as the type of perfect binary leaf trees of height n, we have that *Perfect* comprises perfect trees of arbitrary height.

The functor *Nest* has a similar reading.

$$Nest = K1 + Id \times (K1 + \Delta \times (K1 + \Delta^2 \times (K1 + \Delta^3 \times \cdots)))$$

If we compare *Nest* to *List*

$$List = K1 + Id \times (K1 + Id \times (K1 + Id \times (K1 + Id \times \cdots)))$$

we see that *Nest* comprises 'lists' whose i-th element is a perfect tree of height i.

Partially unrolling *Bush* yields

$$Bush = K1 + Id \times (K1 + Bush \times (K1 + Bush^2 \times (K1 + Bush^3 \times \cdots)))$$

Thus, *Bush* contains 'lists' whose i-th element is a member of $Bush^i$. A truly bewildering form of recursion.

Interestingly, the laws above can be used to eliminate functor composition from the definitions of regular functors. Consider the definition of rose trees which contains the composition $List \cdot Rose$. Introducing a new functor for the composition, say, *Forest* we obtain

$$\begin{aligned} Rose' &= Id \times Forest \\ Forest &= K1 + Rose' \times Forest \ . \end{aligned}$$

Since both types, *Rose* and $Rose'$ have the same structure, they are equivalent from a polytypic perspective. This statement can be made precise if we interpret type equations as *algebraic equations* [4, Sec. 5.1]. Then, both *Rose* and $Rose'$ indeed denote the same rational tree. This implies, however, that care must be taken to ensure that polytypic functions behave the same on both definitions (see Sec. 3). As an aside, note that mutually recursive definitions like the two above pose no problems.

3 Polytypic *map*

Our first example for a polytypic function is the function *map* which describes the action of a functor on arrows (ie functions). This is, of course, not a coincidence. As we shall see, *map* is an important building block for virtually all polytypic functions.

To define *map* we make use of the following auxiliary functions which, as a matter of fact, establish $(+)$ and $(\times)$ as bifunctors.[2]

$$\begin{aligned}
(f_1 \triangledown f_2)\ (inl\ a_1) &= f_1\ a_1 \\
(f_1 \triangledown f_2)\ (inr\ a_2) &= f_2\ a_2 \\
f_1 + f_2 &= inl \circ f_1 \triangledown inr \circ f_2 \\
(f_1 \times f_2)\ (a_1, a_2) &= (f_1\ a_1, f_2\ a_2)
\end{aligned}$$

The polytypic *map* is given by the following definition.

$$\begin{aligned}
map\langle f \rangle &:: (a \to b) \to (f\ a \to f\ b) \\
map\langle K\,t \rangle\ \varphi &= id \\
map\langle Id \rangle\ \varphi &= \varphi \\
map\langle f + g \rangle\ \varphi &= map\langle f \rangle\ \varphi + map\langle g \rangle\ \varphi \\
map\langle f \times g \rangle\ \varphi &= map\langle f \rangle\ \varphi \times map\langle g \rangle\ \varphi \\
map\langle f \cdot g \rangle\ \varphi &= map\langle f \rangle\ (map\langle g \rangle\ \varphi)
\end{aligned}$$

The first line specifies the type of *map*. The notation $map\langle f \rangle$ has been chosen to emphasize that *map* is parametrised by a functor. Polytypic functions are always written using angle brackets which makes it easy to distinguish them from ordinary functions.

In general a polyfunctorial function $poly\langle f \rangle :: \tau(f)$ is given by a set of functions $d_s :: \tau(f_1) \times \ldots \times \tau(f_n) \to \tau(s(f_1, \ldots, f_n))$, one for each functor symbol $s \in \Sigma$. We usually define the functions implicitly using equations of the form $poly\langle s(f_1, \ldots, f_n) \rangle = d_s(poly\langle f_1 \rangle, \ldots, poly\langle f_n \rangle)$. Furthermore, we assume that $d_\Omega = \bot$ for the 'bottom' tree Ω. In the example above we have, for instance, $d_{Id} = id$ and $d_{(\cdot)}(m_1, m_2) = m_1 \circ m_2$. This information is sufficient to define a unique function $poly\langle f \rangle$ for each functor expression f [4, Prop. 2.4.2]. However, there is one further requirement. The polyfunctorial should be invariant with respect to the functor equations (1), ..., (6), ie

$$\begin{aligned}
poly\langle K\,T \cdot F \rangle &= poly\langle K\,T \rangle \\
poly\langle Id \cdot F \rangle &= poly\langle F \rangle \\
poly\langle F \cdot Id \rangle &= poly\langle F \rangle \\
poly\langle (F_1 + F_2) \cdot G \rangle &= poly\langle F_1 \cdot G + F_2 \cdot G \rangle \\
poly\langle (F_1 \times F_2) \cdot G \rangle &= poly\langle F_1 \cdot G \times F_2 \cdot G \rangle \\
poly\langle (F_1 \cdot F_2) \cdot F_3 \rangle &= poly\langle F_1 \cdot (F_2 \cdot F_3) \rangle\ .
\end{aligned}$$

[2] Examples are given in a pidgin based on the functional programming language Haskell [16]. We assume, however, that primitive functions are non-curried, ie $(\wedge)$ has type $Bool \times Bool \to Bool$ rather than $Bool \to (Bool \to Bool)$.

If $poly\langle f\rangle$ violates one of these constraints, then it is sensitive to the form in which f is written, a situation which is clearly undesirable. For instance, $poly\langle Rose\rangle$ and $poly\langle Rose'\rangle$ should be identical since $Rose$ and $Rose'$ define the same functor. It is immediate from its definition that $map\langle f\rangle$ satisfies these conditions.

From the definition of $map\langle f\rangle$ we can derive specializations for different instances of f. Here is the familiar definition of map on $List$.

$$map\langle List\rangle\ \varphi = id + \varphi \times map\langle List\rangle\ \varphi$$

Note that the process of specialization always terminates since the 'index functor' is given by a rational tree which has only a finite set of subtrees. Now, if we specialize map for non-regular functors something interesting happens.

$$map\langle Perfect\rangle\ \varphi = \varphi + map\langle Perfect\rangle\ (\varphi \times \varphi)$$

The type of $map\langle P\rangle$ is $(a \to b) \to (P\ a \to P\ b)$.[3] The recursive call on the right-hand side, however, has type $(\Delta\ a \to \Delta\ b) \to (P\ (\Delta\ a) \to P\ (\Delta\ b))$ which is a substitution instance of its type. This means that $map\langle P\rangle$ is a so-called *polymorphically recursive* function [14]. It should be noted that the Hindley-Milner type system, which underlies most of today's functional programming languages, does not allow polymorphic recursion. Furthermore, a suitable extension of the system has been shown to be undecidable [5]. The functional programming language Haskell [16] allows polymorphic recursion only if an explicit type signature for the function is provided. We will assume henceforth that polymorphic recursion is supported by the underlying programming language.

In general the specialization of a polytypic function $poly\langle f\rangle$ for a given instance of f is defined as follows. Assume that the functor is given by a canonical system of recursion equations. For each equation $f = s(f_1, \ldots, f_n)$ a function definition of the form $poly\langle f\rangle = d_s(poly\langle f_1\rangle, \ldots, poly\langle f_n\rangle)$ is generated. It is straightforward to generalize the process of specialization to arbitrary systems of recursion equations.

To prove properties of polytypic function we can employ the principle of *fixpoint induction*. If P is a *chain-complete* property of functor trees, it suffices to show

$$\begin{aligned}
&P(\Omega)\\
&P(Kt)\\
&P(Id)\\
&P(F) \wedge P(G) \implies P(F + G)\\
&P(F) \wedge P(G) \implies P(F \times G)\\
&P(F) \wedge P(G) \implies P(F \cdot G)\ .
\end{aligned}$$

[3] To improve readability we sometimes abbreviate names of types or constructor functions to their first letter.

Using a simple fixpoint induction we can prove, for instance, the following functorial properties of $map\langle f\rangle$.

$$\begin{aligned} map\langle f\rangle\ id &= id \\ map\langle f\rangle\ (\varphi \circ \psi) &= map\langle f\rangle\ \varphi \circ map\langle f\rangle\ \psi \end{aligned}$$

The proof of $map\langle\Omega\rangle\ id = id$ requires a bit of fudging. We have that $map\langle\Omega\rangle\ \varphi = \lambda a \to \bot$ so we must in effect show that $\lambda a \to \bot = id$. This equation holds, however, since Ω accommodates only the bottom element.

4 Polytypic reduction

A *reduction* is a function of type $f\ a \to a$, which collapses a structure of values of type a into a single value of type a. Because of its destructive nature reductions are also known as *crushes*. The archetypical example for a reduction is the function which sums a structure of integers, for instance, a list or a tree of integers. It is well-known that reductions can be defined completely generically for all regular functors [11]. In the sequel we show how to generalize reductions to non-regular functors. Note that the ability to define reductions on nested datatypes is of some importance since it is not possible to sum a nested datatype using a fold.

To define a reduction we require two ingredients: a value $e :: a$ and a binary operation $op :: \Delta\ a \to a$. Usually but not necessarily e is the neutral element of op.

$$\begin{aligned} &reduce\langle f\rangle && :: a \to (\Delta\ a \to a) \to f\ a \to a \\ &reduce\langle f\rangle\ e\ op && = red\langle f\rangle \\ &\quad \textbf{where} \\ &\quad red\langle K\,t\rangle && = const\ e \\ &\quad red\langle Id\rangle && = id \\ &\quad red\langle f + g\rangle && = red\langle f\rangle \triangledown red\langle g\rangle \\ &\quad red\langle f \times g\rangle && = op \circ (red\langle f\rangle \times red\langle g\rangle) \\ &\quad red\langle f \cdot g\rangle && = red\langle f\rangle \circ map\langle f\rangle\ (red\langle g\rangle) \end{aligned}$$

The definition coincides with the one given by L. Meertens [11], the only difference is that the treatment of recursion is implicit (ie on the meta level) rather than explicit (ie on the object level). This alone suffices to define $reduce\langle f\rangle$ for all datatypes including nested datatypes such as *Perfect*.

The most interesting equation is probably the last one: to reduce a value of type $f\ (g\ a)$, the reduction $red\langle g\rangle$ is first mapped on f to give a value of type $f\ a$ which is then reduced to a value of type a. To see $reduce\langle f\rangle$ in action let us specialize the definition to the type of perfect trees (note that $red\langle\Delta\rangle = op$).

$$red\langle Perfect\rangle = id \triangledown red\langle Perfect\rangle \circ map\langle Perfect\rangle\ op$$

Assume that we want to reduce a perfect tree of height n. The function $map\langle P\rangle\ op$ which has type $P\ (\Delta\ a) \to P\ a$ takes a tree of height $i+1$ to a tree of height i

which is then reduced by a recursive call to $red\langle P\rangle$. After n iterations we obtain the desired value of type a. What about $red\langle P\rangle$'s running time? Let us assume that op is a constant-time operation. The call $map\langle P\rangle\ op$ then takes time proportional to the size of the tree. Since the size of the tree is halved in each step, we obtain a running time of $2^n + 2^{n-1} + \cdots + 2 + 1 = \Theta(2^n)$. Thus, the $red\langle P\rangle$ takes time proportional to the size of the tree. Does this hold in general? Unfortunately not, as the following example shows.

$$\begin{aligned} Maybe &= Nothing\ (K1) + Just\ Id \\ Tower &= End\ Id + Recurse\ (Tower \cdot Maybe) \end{aligned}$$

We have added fancy names for the injection functions to make the examples more readable. Elements of $Tower$ are either of the form $R^n EJ^m N$ with $m<n$ or of the form $R^n EJ^n a$. The reduction on $Tower$ is given by the following equation.

$$red\langle Tower\rangle = id\ \triangledown\ red\langle Tower\rangle \circ map\langle Tower\rangle\ (const\ e\ \triangledown\ id)$$

The crush of $R^n EJ^n a$ proceeds essentially as before: $map\langle T\rangle\ (const\ e\ \triangledown\ id)$ maps $R^i EJ^{i+1} a$ to $R^i EJ^i a$ which is then reduced by a recursive call to $red\langle T\rangle$. The major difference to the proceeding example is that in each step the size of the argument is only decreased by a constant amount. Consequently, we have a total running time of $n + n - 1 + \cdots + 2 + 1 = \Theta(n^2)$, which shows that $red\langle T\rangle$'s running time is quadratic with respect to the size of its argument. Interestingly, the efficiency of $reduce\langle f\rangle$ is not a problem if f is a regular functor. In this case $reduce\langle f\rangle$ always takes time linear to the size of the crushed structure. The argument roughly runs as follows: Recall that each regular functor has an equivalent, composition-free definition. If functor composition is not used, $red\langle f\rangle$ clearly runs in linear time. The asymptotic running time of $red\langle f\rangle$ is, however, the same for equivalent functor definitions.

Fortunately, it is straightforward to improve the efficiency of $reduce\langle f\rangle$. We simply define a function which combines a reduction with a map.

$$\begin{aligned} redmap\langle f\rangle\ \varphi &= red\langle f\rangle \circ map\langle f\rangle\ \varphi \\ red\langle f\rangle &= redmap\langle f\rangle\ id \end{aligned}$$

This idiom is, in fact, a very old one. It appears, for instance, in [3] where it is shown that each homomorphism on lists (ie each catamorphism) can be expressed in this form. The derivation of $redmap\langle f\rangle$ is a simple exercise in program calculation and proceeds almost mechanically. For the two base cases we have

$$\begin{aligned} redmap\langle Kt\rangle\ \varphi &= red\langle Kt\rangle \circ map\langle Kt\rangle\ \varphi = const\ e \circ id = const\ e\ , \\ redmap\langle Id\rangle\ \varphi &= red\langle Id\rangle \circ map\langle Id\rangle\ \varphi = id \circ \varphi = \varphi\ . \end{aligned}$$

The derivation for coproducts and products rests on their properties as bifunctors and are left as exercises to the reader. For the composition of functors we obtain

$$\begin{aligned} redmap\langle f \cdot g\rangle\ \varphi &= red\langle f \cdot g\rangle \circ map\langle f \cdot g\rangle\ \varphi \\ &= red\langle f\rangle \circ map\langle f\rangle\ (red\langle g\rangle) \circ map\langle f\rangle\ (map\langle g\rangle\ \varphi) \\ &= red\langle f\rangle \circ map\langle f\rangle\ (red\langle g\rangle \circ map\langle g\rangle\ \varphi) \\ &= redmap\langle f\rangle\ (redmap\langle g\rangle\ \varphi)\ , \end{aligned}$$

which shows that we succeeded in eliminating the annoying call to *map*. Apart from improving the efficiency of $reduce\langle f\rangle$ the synthesized function is also interesting in its own right since it captures a common pattern of recursion.

$$
\begin{array}{ll}
reduce\langle f\rangle\ e\ op & = reducemap\langle f\rangle\ e\ op\ id \\
reducemap\langle f\rangle & :: a \to (\Delta\ a \to a) \to (b \to a) \to f\ b \to a \\
reducemap\langle f\rangle\ e\ op & = redmap\langle f\rangle \\
\quad \textbf{where} & \\
\quad redmap\langle Kt\rangle\ \varphi & = const\ e \\
\quad redmap\langle Id\rangle\ \varphi & = \varphi \\
\quad redmap\langle f + g\rangle\ \varphi & = redmap\langle f\rangle\ \varphi \triangledown redmap\langle g\rangle\ \varphi \\
\quad redmap\langle f \times g\rangle\ \varphi & = op \circ (redmap\langle f\rangle\ \varphi \times redmap\langle g\rangle\ \varphi) \\
\quad redmap\langle f \cdot g\rangle\ \varphi & = redmap\langle f\rangle\ (redmap\langle g\rangle\ \varphi)
\end{array}
$$

Let us again specialize the definition to the type of perfect trees.

$$redmap\langle Perfect\rangle\ \varphi = \varphi \triangledown redmap\langle Perfect\rangle\ (op \circ (\varphi \times \varphi))$$

It is interesting if not revealing to contrast the two definitions of reductions on perfect trees. The first one, $red\langle P\rangle$, essentially proceeds bottom-up. In each step the 'nodes' (ie pairs) on the lowest level are reduced. This step is repeated until the root is reached. By contrast $redmap\langle P\rangle$ operates in two stages: first a suitable reduction function is constructed which then reduces the perfect tree in a top-down pass. Clearly, the latter procedure is more efficient than the former.

The polytypic function $reducemap\langle f\rangle$ can be used to implement a number of general purpose functions, see, for instance, [11], [8] or [6]. As a single example here is the definition of *size* (note that we use the symbol '+' both for coproducts and for addition).

$$
\begin{array}{l}
size\langle f\rangle :: (Num\ n) \Rightarrow f\ a \to n \\
size\langle f\rangle = reducemap\langle f\rangle\ 0\ (+)\ (const\ 1)
\end{array}
$$

Polytypic reduction satisfies a fusion law analogous to the fusion law for catamorphisms [11].

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

The law can be shown using a straightforward fixpoint induction which is left as an exercise to the reader. Instead we apply the fusion law to derive an efficient implementation of $size\langle Perfect\rangle$. A simple consequence of the fusion law is

$$
\begin{array}{l}
\quad reducemap\langle f\rangle\ 0\ (+)\ (const\ n) \\
= mult\ n \circ reducemap\langle f\rangle\ 0\ (+)\ (const\ 1)\ \textbf{where}\ mult\ a\ b = a * b \ .
\end{array}
$$

Now, we reason

$$
\begin{array}{l}
\quad size\langle Perfect\rangle \\
= reducemap\langle Perfect\rangle\ 0\ (+)\ (const\ 1) \\
= const\ 1 \triangledown reducemap\langle Perfect\rangle\ 0\ (+)\ (reducemap\langle \Delta\rangle\ 0\ (+)\ (const\ 1)) \\
= const\ 1 \triangledown reducemap\langle Perfect\rangle\ 0\ (+)\ (const\ 2) \\
= const\ 1 \triangledown mult\ 2 \circ size\langle Perfect\rangle
\end{array}
$$

Defined this way $size\langle Perfect\rangle$ runs in $\Theta(\log n)$ time.

5 Related and future work

The impetus for writing this paper came while reading the article "Nested Datatypes" by R. Bird and L. Meertens [1]. The authors state that "It is possible to define reductions completely generically for all regular types [...], but we do not know at present whether the same can be done for nested datatypes." We have shown that the answer to this question is in the affirmative. Moreover, the solution presented here is surprisingly simple. To define a polytypic function it suffices to specify its action on polynomial functors. The extension to (mutually) recursive datatypes—which is uniquely defined—is then handled automatically by the system. This does not only simplify matters for the programmer, it also removes some of the redundancy present in the classical approach. In the polytypic programming language PolyP [7], for instance, the user must supply definitions for both polynomial and type functors. There is, however, no guarantee that the corresponding functions behave in related ways.

Very recently, R. Bird and R. Paterson [2] introduced so-called *generalised folds* which overcome some of the problems mentioned in the introduction to this paper. Generalised folds, which can be constructed systematically for each first-order polymorphic datatype, possess a uniqueness property analogous to that of ordinary folds. Their work is largely complementary to ours. While generalised folds are more general than the reductions defined in Sec. 4, it is not clear how to define functions like *sum* generically using generalised folds.

An obvious disadvantage of our approach is that it is not possible to define general recursion schemes like cata- and anamorphisms [12]. The reason is simply that type recursion is left implicit rather than made explicit. The situation can be saved, however, by introducing an operation on functors which maps a type functor to its base functor. Inventing a notation we define $F' = B$ for $F = \mu B$ (the operation (-)$'$ corresponds to the type constructor *FunctorOf* used in PolyP). Of course, F' is only defined if F is a regular functor. Given two functions $in :: t'\ t \to t$ and $out :: t \to t'\ t$ we can now define

$$\begin{aligned}
&cata\langle t\rangle &&:: (t'\ a \to a) \to (t \to a)\\
&cata\langle t\rangle\ \varphi &&= \varphi \circ map\langle t'\rangle\ (cata\langle t\rangle\ \varphi) \circ out\\
&ana\langle t\rangle &&:: (a \to t'\ a) \to (a \to t)\\
&ana\langle t\rangle\ \psi &&= in \circ map\langle t'\rangle\ (ana\langle t\rangle\ \psi) \circ \psi\ .
\end{aligned}$$

Directions for future work suggest themselves. In the full version of this paper [6] we show how to generalize the technique to n-ary functors. The full paper also contains additional examples, among others the definition of polytypic equality. It remains to broaden the approach to include exponentials and higher-order polymorphism [10]. The former extension should be fairly straightforward, see [17] or [13]. The latter extension is far more challenging. To illustrate, consider the following generalization of rose trees.

$$\mathbf{data}\ GRose\ f\ a = Node\ a\ (f\ (GRose\ f\ a))$$

This datatype is used, for instance, to extend an implementation of collections (sequences or priority queues) with an efficient operation for combining two collections (catenate or meld), see [15]. From a categorical point of view we could interpret *GRose* as a higher-order functor of type $Fun(\mathbf{C}) \to Fun(\mathbf{C})$ where $Fun(\mathbf{C})$ is the category which has as objects functors of type $\mathbf{C} \to \mathbf{C}$ and as arrows natural transformations between them. Equating natural transformations and polymorphic functions it follows that *GRose*'s map functional has the type $(\forall a.f\ a \to g\ a) \to (\forall a.\mathit{GRose}\ f\ a \to \mathit{GRose}\ g\ a)$. Unfortunately, this is a rank-2 type which is not legal Haskell. An alternative view is to interpret *GRose* as a higher-order functor of type $Cat(\mathbf{C}) \to Cat(\mathbf{C})$ where $Cat(\mathbf{C})$ is the category which has as the only object the category $\mathbf{C}$ and as arrows functors of type $\mathbf{C} \to \mathbf{C}$. This suggests to define a higher-order map of type $(\forall a.\forall b.(a \to b) \to (f\ a \to f\ b)) \to (\forall a.\forall b.(a \to b) \to (\mathit{GRose}\ f\ a \to \mathit{GRose}\ f\ b))$. Both maps are clearly useful: The first operates on the base collection f of the 'bootstrapped' collection *GRose* f while the second operates on the elements contained in a collection. However, it is far from clear how to define these maps generically for all higher-order datatypes.

Acknowledgement Thanks are due to Richard Bird and two anonymous referees for their helpful comments on a draft version of this paper.

References

1. Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, LNCS 1422, pages 52–67. Springer Verlag, June 1998.
2. Richard Bird and Ross Paterson. Generalised folds for nested datatypes, 1998. Submitted for publication.
3. Richard S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computer Science*. Springer Verlag, 1988.
4. Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, March 1983.
5. Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.
6. Ralf Hinze. Polytypic functions over nested datatypes. Technical Report IAI-TR-98-10, Institut für Informatik III, Universität Bonn, October 1998.
7. Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Symp. on Principles of Programming Languages, POPL'97, Paris, France*, pages 470–482, New York, January 1997. ACM Press.
8. Patrik Jansson and Johan Jeuring. PolyLib—A library of polytypic functions. In R. Backhouse and T. Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998*. Dept. of Computing Science, Chalmers Univ. of Techn. and Göteborg Univ.
9. Johan Jeuring and Patrik Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *2nd International School on Advanced Functional Programming*, LNCS 1129, pages 68–114. Springer Verlag, 1996.

10. 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.
11. Lambert Meertens. Calculate polytypically! In H. Kuchen and S.D. Swiestra, editors, *8th Intl. Symp. on Programming Languages: Implementations, Logics, and Programs, PLILP'96*, LNCS 1140, pages 1–16. Springer Verlag, 1996.
12. E. Meijer, M. Fokkinga, and Paterson R. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, MA*, LNCS 523, pages 124–144. Springer Verlag, 1991.
13. Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Intl. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95*, pages 324–333. ACM Press, June 1995.
14. Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *International Symposium on Programming, 6th Colloquium Toulouse*, 1984. LNCS 167.
15. Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.
16. J. Peterson and K. Hammond. Report on the programming language Haskell 1.4, a non-strict, purely functional language. Research Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, March 1997.
17. T. Sheard and L. Fegaras. A fold for all seasons. In *Functional Programming Languages and Computer Architecture*, pages 233–242, Copenhagen, June 1993.