Discrete Mathematics and Theoretical Computer Science **3**, 1999, 193–214

Polytypic Functions Over Nested Datatypes[†]

Ralf Hinze

Institut für Informatik III, Universität Bonn, Römerstraße 164, 53117 Bonn, Germany

received January 18, 1999, *revised July 27, 1999*, *accepted September 14, 1999*.

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 [3] 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 that 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 type arguments for polytypic functions. Besides covering a wider range of types the approach is also simpler and technically less involving than previous ones. We present several examples of polytypic functions, among others polytypic reduction and polytypic equality. The presentation assumes some background in functional and in polytypic programming. A basic knowledge of monads is required for some of the examples.

Keywords: Functional programming, generic programming, nested datatypes, rational trees, reductions.

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.

$$\textbf{data}\ List\ a \quad = \quad 1 + a \times List\ a$$

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 the type of perfectly balanced, binary leaf trees [9]

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

[†]An extended abstract of this article appeared in the proceedings of the 3rd Latin-American Conference on Functional Programming (CLaPF'99).

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 some G.

A way out of this dilemma is to consider initial algebras of higher-order functors [3]. 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:

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

where KT 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, since *size* is parametrised by a functor, we will qualify *size* more precisely as a *polyfunctorial* function. The qualifier *polytypic* will be used as a general term for functions that are parametrised by functors of arbitrary arity. Now, inspired by the initial algebra semantics the structure of functors is usually given by the following grammar, see [15] or [12].

$$\begin{array}{lcl} B & = & KT \mid Fst \mid Snd \mid B + B \mid B \times B \mid F \cdot B \\ F & = & \mu B \end{array}$$

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 minimalization of bifunctors is not the most direct way to go. An alternative that we explore in this article is to define functor expressions as *rational trees* [6] over the following grammar of unary functors.

$$F \quad = \quad KT \mid Id \mid F + F \mid F \times F \mid F \cdot F$$

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 that has only a finite number of subtrees. Fig. 1 displays the rational trees defined by the type equations above. Note that the pictorial representation already employs the defining property of

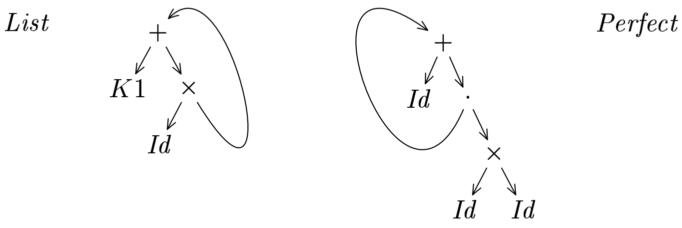

Fig. 1: Types interpreted as rational trees.

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 type arguments 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 article is organized as follows. Sec. 2 introduces rational trees and functor expressions. Sec. 3 and Sec. 4 give several examples of 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. Sec. 5 extends the approach to n-ary functors and presents further examples, among others polytypic equality and a polytypic monadic map. This section requires a basic knowledge of monads. Finally, Sec. 6 reviews related work and points out directions for future work.

2 Functor expressions

There are several ways to represent rational trees, see, for instance, [6, 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 that is in canonical form. For instance, the system $\{x = Id + x \cdot (Id \times Id)\}$ has the canonical form $\{x = x_1 + x_2, x_1 = Id, x_2 = x \cdot x_3, x_3 = Id \times Id\}$. Note that the canonicalization introduces additional recursion variables. Furthermore, 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

$$F \quad = \quad KT \mid Id \mid F + F \mid F \times F \mid F \cdot F \ ,$$

ie $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_0 = \{ Kt \mid t \in T \} \cup \{Id\}$ and $\Sigma_2 = \{+, \times, \cdot\}$. We agree upon that $(\cdot)$ binds more tightly than $(\times)$, which in turn takes precedence over $(+)$. For instance, $F + G \times H \cdot H$ means $F + (G \times (H \cdot H))$.

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.[‡] 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.

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

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

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

Unrolling *Perfect* yields

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

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

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 perfectly balanced, binary leaf trees of height n, we have that *Perfect* comprises perfect trees of arbitrary height.

The functor *Nest* has a similar reading.

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

If we compare *Nest* to

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

we see that *Nest* comprises 'lists' whose i-th element is a perfect tree of height i. Alternatively, we may view values of type *Nest* as *node-oriented* perfect trees, where the i-th list element describes the sequence of labels at depth i.

Partially unrolling *Bush* yields

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

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

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

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* [6, Sec. 5.1]. Then, both *Rose* and $Rose'$ indeed denote the same rational tree. This implies, in particular, 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 are covered by recursion equations in a natural way.

3 Polytypic *map*

Our first example of 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.[§]

$$
\begin{array}{lcl}
(f_1 \bigtriangledown f_2)\ (inl\ a_1) & = & f_1\ a_1 \\
(f_1 \bigtriangledown f_2)\ (inr\ a_2) & = & f_2\ a_2 \\
f_1 + f_2 & = & (inl \circ f_1) \bigtriangledown (inr \circ f_2) \\
(f_1 \vartriangle f_2)\ a & = & (f_1\ a, f_2\ a) \\
outl\ (a_1, a_2) & = & a_1 \\
outr\ (a_1, a_2) & = & a_2 \\
f_1 \times f_2 & = & (f_1 \circ outl) \vartriangle (f_2 \circ outr)
\end{array}
$$

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

$$
\begin{array}{lcl}
map\langle f\rangle & :: & (a \to b) \to (f\ a \to f\ b) \\
map\langle Kt\rangle\ \varphi & = & id \\
map\langle Id\rangle\ \varphi & = & \varphi \\
map\langle f+g\rangle\ \varphi & = & map\langle f\rangle\ \varphi + map\langle g\rangle\ \varphi \\
map\langle f\times g\rangle\ \varphi & = & map\langle f\rangle\ \varphi \times map\langle g\rangle\ \varphi \\
map\langle f\cdot g\rangle\ \varphi & = & map\langle f\rangle\ (map\langle g\rangle\ \varphi)
\end{array}
$$

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 $d_\Omega = \perp$ 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 [6, Prop. 2.4.2]. However, there is one further requirement. The polyfunctorial should be invariant with respect to the functor equations (1), …, (6), ie

$$
\begin{array}{rcl}
poly\langle KT\cdot F\rangle & = & poly\langle KT\rangle \\
poly\langle Id\cdot F\rangle & = & poly\langle F\rangle \\
poly\langle F\cdot Id\rangle & = & poly\langle F\rangle \\
poly\langle (F_1+F_2)\cdot G\rangle & = & poly\langle F_1\cdot G + F_2\cdot G\rangle \\
poly\langle (F_1\times F_2)\cdot G\rangle & = & poly\langle F_1\cdot G \times F_2\cdot G\rangle \\
poly\langle (F_1\cdot F_2)\cdot F_3\rangle & = & poly\langle F_1\cdot (F_2\cdot F_3)\rangle\ .
\end{array}
$$

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.

[§] Examples are given in a pidgin based on the functional programming language Haskell 98 [21]. We assume, however, that primitive functions are non-curried, ie $(\wedge)$ has type $Bool \times Bool \to Bool$ rather than $Bool \to (Bool \to Bool)$.

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 'functor argument' 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.

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

The type of $map\langle Perfect\rangle$ is $(a \to b) \to (Perfect\ a \to Perfect\ b)$. The recursive call on the right-hand side, however, has type $(\Delta\ a \to \Delta\ b) \to (Perfect\ (\Delta\ a) \to Perfect\ (\Delta\ b))$, which is a substitution instance of $map\langle Perfect\rangle$'s type. This means that $map\langle Perfect\rangle$ is a so-called *polymorphically recursive* function [19]. 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 [8]. The functional programming language Haskell [21] allows polymorphic recursion only if an explicit type signature is provided for the function. 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

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

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

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

The proof for coproduct and product relies on the fact that $(+)$ and $(\times)$ are themselves bifunctors.

$$(f_1 + f_2) \circ (g_1 + g_2) \quad = \quad (f_1 \circ g_1) + (f_2 \circ g_2) \qquad (7)$$

$$(f_1 \times f_2) \circ (g_1 \times g_2) \quad = \quad (f_1 \circ g_1) \times (f_2 \circ g_2) \qquad (8)$$

Only 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 $f\ a \to a$ that 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 of a reduction is the function that 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 [15]. In the sequel we show how to generalize reductions to non-regular functors. 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. Note that this implies that reductions are in a sense more general than folds. They are, however, also less general since, for instance, mapping functions on regular functors can be defined using folds, but cannot be expressed in the form of a crush. Thus, reductions [15] and folds [14] are different, incomparable generalizations of the classic reduction operator $(/)$ on join-lists [5]. For a more detailed comparison we refer the interested reader to [15].

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.

$$
\begin{array}{lcl}
\mathit{reduce}\langle f\rangle & :: & a \to (\Delta\ a \to a) \to (f\ a \to a) \\
\mathit{reduce}\langle f\rangle\ e\ op & = & \mathit{red}\langle f\rangle \\
\quad \mathbf{where} & & \\
\quad \mathit{red}\langle Kt\rangle & = & \mathit{const}\ e \\
\quad \mathit{red}\langle Id\rangle & = & \mathit{id} \\
\quad \mathit{red}\langle f+g\rangle & = & \mathit{red}\langle f\rangle \triangledown \mathit{red}\langle g\rangle \\
\quad \mathit{red}\langle f\times g\rangle & = & op \circ (\mathit{red}\langle f\rangle \times \mathit{red}\langle g\rangle) \\
\quad \mathit{red}\langle f\cdot g\rangle & = & \mathit{red}\langle f\rangle \circ \mathit{map}\langle f\rangle\ (\mathit{red}\langle g\rangle)
\end{array}
$$

The definition coincides with the one given by L. Meertens [15], the only difference is that the treatment of recursion is implicit—the specialization automatically takes care of recursion—rather than explicit. This alone suffices to define $\mathit{reduce}\langle f\rangle$ 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 $\mathit{red}\langle g\rangle$ 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 $\mathit{reduce}\langle f\rangle$ in action let us specialize the definition to the type of perfect trees (note that $\mathit{red}\langle\Delta\rangle = op$).

$$
\mathit{red}\langle \mathit{Perfect}\rangle \quad = \quad \mathit{id} \triangledown \mathit{red}\langle \mathit{Perfect}\rangle \circ \mathit{map}\langle \mathit{Perfect}\rangle\ op
$$

Assume that we want to reduce a perfect tree of height n. The function $\mathit{map}\langle \mathit{Perfect}\rangle\ op$, which has type $\mathit{Perfect}\ (\Delta\ a) \to \mathit{Perfect}\ a$, takes a tree of height $i+1$ to a tree of height i, which is then reduced by a recursive call to $\mathit{red}\langle \mathit{Perfect}\rangle$. After n iterations we obtain the desired value of type a. What about $\mathit{red}\langle \mathit{Perfect}\rangle$'s running time? Let us assume that op is a constant-time operation. The call $\mathit{map}\langle \mathit{Perfect}\rangle\ op$ 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 $\mathit{red}\langle \mathit{Perfect}\rangle$ takes time proportional to the size of the tree. Does this hold in general? Unfortunately not, as the following example shows.

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

We have added fancy names for the injection functions to make the examples more readable. Now, elements of type *Tower* are either of the form $\mathit{Recurse}^n\ (\mathit{End}\ (\mathit{Just}^m \mathit{Nothing}))$ with $m < n$ or of the form $\mathit{Recurse}^n\ (\mathit{End}\ (\mathit{Just}^n a))$. The reduction on *Tower* is given by the following equation.

$$\mathit{red}\langle \mathit{Tower}\rangle \quad = \quad \mathit{id} \mathbin{\triangledown} \mathit{red}\langle \mathit{Tower}\rangle \circ \mathit{map}\langle \mathit{Tower}\rangle\ (\mathit{const}\ e \mathbin{\triangledown} \mathit{id})$$

The crush of $\mathit{Recurse}^n\ (\mathit{End}\ (\mathit{Just}^n a))$ proceeds essentially as before: $\mathit{map}\langle \mathit{Tower}\rangle\ (\mathit{const}\ e \mathbin{\triangledown} \mathit{id})$ maps the value $\mathit{Recurse}^i\ (\mathit{End}\ (\mathit{Just}^{i+1} a))$ to $\mathit{Recurse}^i\ (\mathit{End}\ (\mathit{Just}^i a))$, which is then reduced by a recursive call to $\mathit{red}\langle \mathit{Tower}\rangle$. The major difference to the preceding 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 $\mathit{red}\langle \mathit{Tower}\rangle$'s running time is quadratic with respect to the size of its argument. Interestingly, the efficiency of $\mathit{reduce}\langle f\rangle$ is not a problem if f is a regular functor. In this case $\mathit{reduce}\langle f\rangle$ 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, $\mathit{red}\langle f\rangle$ clearly runs in linear time. The asymptotic running time of $\mathit{red}\langle f\rangle$ is, however, the same for equivalent functor definitions.

Fortunately, it is straightforward to improve the efficiency of $\mathit{reduce}\langle f\rangle$. We simply define a function that combines a reduction with a map, ie we seek for a function $\mathit{redmap}\langle f\rangle$ that satisfies

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

This idiom is, in fact, a very old one. It appears, for instance, in [5] where it is shown that each homomorphism on lists (ie each fold) can be expressed in this form. The derivation of $\mathit{redmap}\langle f\rangle$ is a simple exercise in program calculation and proceeds almost mechanically. For the two base cases we have

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

The derivation for coproducts and products rests on their properties as bifunctors, ie on equations (7) and (8).

$$\begin{aligned}
\mathit{redmap}\langle f+g\rangle\ \varphi \quad &= \quad \mathit{red}\langle f+g\rangle \circ \mathit{map}\langle f+g\rangle\ \varphi\\
&= \quad (\mathit{red}\langle f\rangle \mathbin{\triangledown} \mathit{red}\langle g\rangle) \circ (\mathit{map}\langle f\rangle\ \varphi + \mathit{map}\langle g\rangle\ \varphi)\\
&= \quad (\mathit{red}\langle f\rangle \circ \mathit{map}\langle f\rangle\ \varphi) \mathbin{\triangledown} (\mathit{red}\langle g\rangle \circ \mathit{map}\langle g\rangle\ \varphi)\\
&= \quad \mathit{redmap}\langle f\rangle\ \varphi \mathbin{\triangledown} \mathit{redmap}\langle g\rangle\ \varphi\\
\mathit{redmap}\langle f\times g\rangle\ \varphi \quad &= \quad \mathit{red}\langle f\times g\rangle \circ \mathit{map}\langle f\times g\rangle\ \varphi\\
&= \quad \mathit{op} \circ (\mathit{red}\langle f\rangle \times \mathit{red}\langle g\rangle) \circ (\mathit{map}\langle f\rangle\ \varphi \times \mathit{map}\langle g\rangle\ \varphi)\\
&= \quad \mathit{op} \circ ((\mathit{red}\langle f\rangle \circ \mathit{map}\langle f\rangle\ \varphi) \times (\mathit{red}\langle g\rangle \circ \mathit{map}\langle g\rangle\ \varphi))\\
&= \quad \mathit{op} \circ (\mathit{redmap}\langle f\rangle\ \varphi \times \mathit{redmap}\langle g\rangle\ \varphi)
\end{aligned}$$

And finally, for the composition of functors we obtain

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

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

Fig. 2: Examples of reductions.

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

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

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

$$\mathit{redmap}\langle \mathit{Perfect}\rangle\ \varphi \quad = \quad \varphi \bigtriangledown \mathit{redmap}\langle \mathit{Perfect}\rangle\ (\mathit{op} \circ (\varphi \times \varphi))$$

It is interesting if not revealing to contrast the two definitions of reductions on perfect trees. The first one, $\mathit{red}\langle \mathit{Perfect}\rangle$, 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, $\mathit{redmap}\langle \mathit{Perfect}\rangle$ operates in two stages: first a suitable reduction function is constructed, which then reduces the perfect tree in a single top-down pass. Clearly, the latter procedure is more efficient than the former.

Fig. 2 lists some typical applications of $\mathit{reduce}\langle f\rangle$ and $\mathit{reducemap}\langle f\rangle$. Further examples can be found, for instance, in [15] or [11].

Polytypic reduction satisfies a fusion law analogous to the fusion law for folds [15].

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

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 $\mathit{size}\langle\mathit{Perfect}\rangle$. A simple consequence of the fusion law is

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

Now, we reason

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

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

5 Extending the approach to n-ary functors

By now we have seen several examples of polyfunctorial functions, ie functions that are parametrised by an unary functor. In this section we extend the approach from unary to n-ary functors. It is relatively straightforward to extend functor expressions to the general case: An *n-ary functor expression* is a rational tree over the following set of 'functor' symbols.

$$\begin{array}{rcll} F^{(n)} & = & KT & \\ & \mid & P_i^n & (1 \leqslant i \leqslant n) \\ & \mid & F^{(n)} + F^{(n)} & \\ & \mid & F^{(n)} \times F^{(n)} & \\ & \mid & F^{(k)} \cdot (F_1^{(n)}, \ldots, F_k^{(n)}) & (1 \leqslant k) \end{array}$$

The superscript in $F^{(n)}$ denotes the arity of the functor. As before, KT denotes the n-ary constant functor. The n-ary projection functor P_i^n is given by $P_i^n\ \vec{T} = T_i$ for $\vec{T} = (T_1, \ldots, T_n)$. For $n = 1$ and $n = 2$ we use the following more familiar names: $\mathit{Id} = P_1^1$, $\mathit{Fst} = P_1^2$, and $\mathit{Snd} = P_2^2$. The coproduct and product of n-ary functors are defined as usual: $(F_1 + F_2)\ \vec{T} = F_1\ \vec{T} + F_2\ \vec{T}$ and $(F_1 \times F_2)\ \vec{T} = F_1\ \vec{T} \times F_2\ \vec{T}$. Finally, $F \cdot (G_1, \ldots, G_k)$ denotes the composition of the k-ary functor F with k n-ary functors: $(F \cdot (G_1, \ldots, G_k))\ \vec{T} = F\ (G_1\ \vec{T}, \ldots, G_k\ \vec{T})$. We omit the parentheses when $k = 1$, ie we write $F \cdot F_1$ instead of $F \cdot (F_1)$.

From the definitions of KT, P^n_i, $(+)$, $(\times)$, and $(\cdot)$ we can easily derive the following laws.

$$\begin{aligned}
KT \cdot (F_1, \ldots, F_n) &= KT \\
P^n_i \cdot (F_1, \ldots, F_n) &= F_i \\
F \cdot (P^n_1, \ldots, P^n_n) &= F \\
(F_1 + F_2) \cdot (G_1, \ldots, G_n) &= F_1 \cdot (G_1, \ldots, G_n) + F_2 \cdot (G_1, \ldots, G_n) \\
(F_1 \times F_2) \cdot (G_1, \ldots, G_n) &= F_1 \cdot (G_1, \ldots, G_n) \times F_2 \cdot (G_1, \ldots, G_n) \\
(F \cdot (G_1, \ldots, G_n)) \cdot (H_1, \ldots, H_o) &= F \cdot (G_1 \cdot (H_1, \ldots, H_o), \ldots, G_m \cdot (H_1, \ldots, H_o))
\end{aligned}$$

For $n = o = 1$ we obtain the laws given in Sec. 2.

Coproduct and product are the simplest examples of bifunctors.

$$\begin{aligned}
\mathit{Coproduct} &= \mathit{Fst} + \mathit{Snd} \\
\mathit{Product} &= \mathit{Fst} \times \mathit{Snd}
\end{aligned}$$

It is easy to verify that $\mathit{Coproduct} \cdot (F_1, F_2) = F_1 + F_2$ and $\mathit{Product} \cdot (F_1, F_2) = F_1 \times F_2$, which shows that we could have added $(+)$ and $(\times)$ as constants to $F^{(2)}$. The infix notation is, however, more convenient to use.

External search trees make nice examples of bifunctors. An external search tree of type $\mathit{ST}\ a\ b$ has external nodes (ie leafs) labelled with values of type a and internal nodes (ie branches) labelled with values of type b.

$$\begin{aligned}
\mathit{Branch} &= \mathit{Fst} \times \mathit{Snd} \times \mathit{Fst} \\
\mathit{ST} &= \mathit{Fst} + \mathit{Branch} \cdot (\mathit{ST}, \mathit{Snd}) \\
\mathit{PST} &= \mathit{Fst} + \mathit{PST} \cdot (\mathit{Branch}, \mathit{Snd})
\end{aligned}$$

External search trees come in two flavours: ST is the standard textbook type while PST is the type of *perfect* external search trees. Note that ST is right recursive whereas PST is left recursive. The classification of functors into polynomial, regular, and non-regular functors also carries over to the n-ary case. It is easy to see that Branch is a polynomial, ST a regular, and PST a non-regular bifunctor.

5.1 Polytypic and polyfunctorial functions

Our first (and our last) example of a polytypic function that is indexed by a nullary functor is polytypic equality. Besides implementing a fundamental operation the example also demonstrates the interaction between polytypic and polyfunctorial functions. Note that the definition of equality is based upon the approach taken in PolyLib [11]. In PolyLib, however, equality is parametrised by an unary functor. Now, to be able to give the definitions in a point-free style, we make use of the auxiliary functions shown in Fig. 3. The first cases of $eq\langle t\rangle$ are straightforward. We assume that the primitive types are 1, Int, and Double and that suitable equality functions for Int and Double are predefined.

$$\begin{aligned}
eq\langle t\rangle &:: t \times t \to \mathit{Bool} \\
eq\langle K1\rangle &= \mathit{const}\ \mathit{True} \\
eq\langle K\mathit{Int}\rangle &= \mathit{eqint} \\
eq\langle K\mathit{Double}\rangle &= \mathit{eqdouble} \\
eq\langle t_1 + t_2\rangle &= \mathit{ok}\ (eq\langle t_1\rangle \mathbin{\triangledown} eq\langle t_2\rangle) \circ \mathit{couple} \\
eq\langle t_1 \times t_2\rangle &= (\wedge) \circ (eq\langle t_1\rangle \times eq\langle t_2\rangle) \circ \mathit{transpose}
\end{aligned}$$

$$
\begin{array}{lcl}
(p \to f, g)\ a & = & \mathbf{if}\ p\ a\ \mathbf{then}\ f\ a\ \mathbf{else}\ g\ a \\
couple\ (inl\ a_1, inl\ b_1) & = & Just\ (inl\ (a_1, b_1)) \\
couple\ (inl\ a_1, inr\ b_2) & = & Nothing \\
couple\ (inr\ a_2, inl\ b_1) & = & Nothing \\
couple\ (inr\ a_2, inr\ b_2) & = & Just\ (inr\ (a_2, b_2)) \\
transpose\ ((a_1, a_2), (b_1, b_2)) & = & ((a_1, b_1), (a_2, b_2)) \\
ok\ \varphi\ Nothing & = & False \\
ok\ \varphi\ (Just\ a) & = & \varphi\ a
\end{array}
$$

Fig. 3: Auxiliary functions for polytypic equality.

It remains to define $eq\langle t\rangle$ for the case that t is a composition of a k-ary functor with k nullary functors. To keep the example manageable we confine ourselves to unary functors. How do we proceed if t takes the form $f\ u$? Now, two elements of $f\ u$ are equal if they have the same shape and the elements of type u at corresponding positions are equal. To bring corresponding elements together we make use of an auxiliary, polyfunctorial function called $zip\langle f\rangle$. As the name indicates, $zip\langle f\rangle :: (f\ a, f\ b) \to Maybe\ (f\ (a,b))$ turns a pair of structures into a structure of pairs. If the structures have not the same shape, $zip\langle f\rangle$ returns *Nothing*, otherwise it yields *Just* z where z is the desired structure. Using $zip\langle f\rangle$ the last case of $eq\langle t\rangle$ can be defined as

$$eq\langle f \cdot u\rangle \quad = \quad ok\ (all\langle f\rangle\ (eq\langle u\rangle)) \circ zip\langle f\rangle \ ,$$

which is more or less a translation of the english description above.

The auxiliary function $zip\langle f\rangle$ makes a nice example of polytypic, *monadic* programming. Before we discuss its definition let us briefly review the basics of monads. For a more in-depth treatment we refer the interested reader to P. Wadler's papers [23, 24, 25], which also contain supplementary pointers to relevant work. One can think of a monad as an abstract type for computations. In Haskell monads are captured by the following class declaration.

$$
\begin{array}{l}
\mathbf{class}\ Monad\ m\ \mathbf{where} \\
\quad \begin{array}{lcl}
return & :: & a \to m\ a \\
(\gg\!=) & :: & m\ a \to (a \to m\ b) \to m\ b
\end{array}
\end{array}
$$

The essential idea of monads is to distinguish between *computations* and *values*. This distinction is reflected on the type level: an element of $m\ a$ represents a computation that yields a value of type a. A computation may involve, for instance, state, exceptions, or nondeterminism. The trivial computation that immediately returns the value a is denoted by $return\ a$. The operator $(\gg\!=)$, commonly called 'bind', combines two computations: $m \gg\!= k$ applies k to the result of the computation m.

The monadic operations must be related by the following so-called *monad laws*.

$$return\ a \gg\!= k \quad = \quad k\ a \tag{9}$$

$$m \gg\!= return \quad = \quad m \tag{10}$$

$$(m \gg\!= k) \gg\!= \ell \quad = \quad m \gg\!= (\lambda a \to k\ a \gg\!= \ell) \tag{11}$$

Roughly speaking *return* is the neutral element of $(\gg\!=)$ and $(\gg\!=)$ is associative. The monoidal structure becomes more apparent if the laws are rephrased in terms of the monadic composition, see below.

Haskell supports monadic programming by providing a more readable, first-order syntax for $(\gg\!=)$, the so-called **do**-*notation*. The syntax and semantics of **do**-expressions are given by the following identities:

$$\begin{aligned}\mathbf{do}\,\{x \leftarrow m; e\} &= m \gg\!= \lambda x \rightarrow \mathbf{do}\,\{e\}\\ \mathbf{do}\,\{e\} &= e\ .\end{aligned}$$

Several datatypes have a computational content. For instance, the type *Maybe* can be used to model exceptions: *Just x* represents a 'normal' or successful computation yielding the value *x* while *Nothing* represents an exceptional or failing computation—$zip\langle f\rangle$ employs *Maybe* in this sense. The following instance declaration shows how to define *return* and $(\gg\!=)$ for *Maybe*.

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad\ Maybe}\ \textbf{where}\\ \quad \begin{array}{lcl}\mathit{return} &=& \mathit{Just}\\ \mathit{Nothing} \gg\!= k &=& \mathit{Nothing}\\ \mathit{Just}\ a \gg\!= k &=& k\ a\end{array}\end{array}$$

Thus, $m \gg\!= k$ can be seen as a strict postfix application: if m is an exception, the exception is propagated; if m succeeds, then k is applied to the resulting value.

In the previous examples we have made heavy use of general combining forms such as $(\circ)$, $(+)$, and $(\times)$. The function $zip\langle f\rangle$ can be defined quite succinctly if we raise these combinators to *procedures*. A procedure is by definition a function that maps values to computations, ie it is a function of type $a \rightarrow m\ b$ where m is a monad.

$$\begin{aligned}(h_1 \circledcirc h_2)\ a &= \mathbf{do}\,\{b \leftarrow h_2\ a; h_1\ b\}\\ (h_1 \otimes h_2)\ (a_1, a_2) &= \mathbf{do}\,\{b_1 \leftarrow h_1\ a_1; b_2 \leftarrow h_2\ a_2; \mathit{return}\ (b_1, b_2)\}\\ (h_1 \oplus h_2)\ (\mathit{inl}\ a_1) &= \mathbf{do}\,\{b_1 \leftarrow h_1\ a_1; \mathit{return}\ (\mathit{inl}\ b_1)\}\\ (h_1 \oplus h_2)\ (\mathit{inr}\ a_2) &= \mathbf{do}\,\{b_2 \leftarrow h_2\ a_2; \mathit{return}\ (\mathit{inr}\ b_2)\}\end{aligned}$$

While the definitions for $(\circledcirc)$ and $(\oplus)$ are inevitable, there is a choice to be made in the case of $(\otimes)$. The two computations $h_1\ a_1$ and $h_2\ a_2$ can be sequenced in two possible ways, either $h_1\ a_1$ before $h_2\ a_2$ or vice versa. Fortunately, the two definitions coincide for $m = \mathit{Maybe}$.[¶] From the definitions above we can easily derive the following equations:

$$\begin{aligned}f \circledcirc (g \circ h) &= (f \circledcirc g) \circ h && (12)\\ f \circledcirc \mathit{return} &= f && (13)\\ \mathit{return} \circledcirc f &= f && (14)\\ f \circledcirc (g \circledcirc h) &= (f \circledcirc g) \circledcirc h && (15)\\ (f_1 \oplus f_2) \circledcirc (g_1 \oplus g_2) &= (f_1 \circledcirc g_1) \oplus (f_2 \circledcirc g_2) && (16)\\ (f_1 \otimes f_2) \circledcirc (g_1 \otimes g_2) &= (f_1 \circledcirc g_1) \otimes (f_2 \circledcirc g_2) \quad \text{if } m \text{ is commutative} && (17)\end{aligned}$$

¶ We tacitly assume that computations do not diverge. Otherwise this property does not hold, take, for instance, $h_1\ a_1 = \mathit{Nothing}$ and $h_2\ a_2 = \bot$.

Note that equations (13), (14), and (15) are the monad laws stated in terms of the monadic composition ($\circledcirc$). These equations furthermore show that procedures (for a fixed m) form the morphisms of a category, the so-called *Kleisli category*, where the identity is *return* and the composition is ($\circledcirc$). The last equation only holds for so-called *commutative monads* that satisfy

$$\mathbf{do}\,\{a_1 \leftarrow m_1; a_2 \leftarrow m_2; \mathit{return}\ (a_1, a_2)\} \quad = \quad \mathbf{do}\,\{a_2 \leftarrow m_2; a_1 \leftarrow m_1; \mathit{return}\ (a_1, a_2)\}\ .$$

For instance, the identity monad, the *Maybe* monad, and reader monads all have this property. On the other hand, the list monad, the state monads, and the *IO* monad are not commutative. This implies that $(\times)$ is not a bifunctor in the corresponding Kleisli category.

Now for the definition of $\mathit{zip}\langle t\rangle$. The first case falls back on the equality of values. Thus, $\mathit{eq}\langle t\rangle$ and $\mathit{zip}\langle f\rangle$ are mutually recursive functions, which is not surprising since $F^{(0)}$ and $F^{(1)}$ are mutually recursive as well (we tacitly identify nullary functors, $F^{(0)}$, and types, T).

$$\begin{array}{lcl}
\mathit{zip}\langle f\rangle & :: & f\ a \times f\ b \rightarrow \mathit{Maybe}\ (f\ (a \times b)) \\
\mathit{zip}\langle \mathit{Kt}\rangle & = & (\mathit{eq}\langle t\rangle \rightarrow \mathit{return} \circ \mathit{outl}, \mathit{const}\ \mathit{Nothing}) \\
\mathit{zip}\langle \mathit{Id}\rangle & = & \mathit{return} \\
\mathit{zip}\langle f+g\rangle & = & (\mathit{zip}\langle f\rangle \oplus \mathit{zip}\langle g\rangle) \circledcirc \mathit{couple} \\
\mathit{zip}\langle f\times g\rangle & = & (\mathit{zip}\langle f\rangle \otimes \mathit{zip}\langle g\rangle) \circ \mathit{transpose} \\
\mathit{zip}\langle f\cdot g\rangle & = & \mathit{mapM}\langle f\rangle\ (\mathit{zip}\langle g\rangle) \circledcirc \mathit{zip}\langle f\rangle
\end{array}$$

The last case is again the most interesting one. To zip two values of type $f\ (g\ a)$ and $f\ (g\ b)$ we first apply $\mathit{zip}\langle f\rangle$, which yields a value of type $\mathit{Maybe}\ (f\ (g\ a \times g\ b))$. To obtain a value of the desired type, $\mathit{Maybe}\ (f\ (g\ (a \times b)))$, we then map $\mathit{zip}\langle g\rangle$ on f. However, since we are working within a monad we cannot use the 'ordinary' map operation as defined in Sec. 3. Instead we must employ a *monadic map* [7], which is defined as follows (note that $\mathit{mapM}\langle f\rangle$ is termed *mapl* in [18]).

$$\begin{array}{lcl}
\mathit{mapM}\langle f\rangle & :: & (\mathit{Monad}\ m) \Rightarrow (a \rightarrow m\ b) \rightarrow (f\ a \rightarrow m\ (f\ b)) \\
\mathit{mapM}\langle \mathit{Kt}\rangle\ \varphi & = & \mathit{return} \\
\mathit{mapM}\langle \mathit{Id}\rangle\ \varphi & = & \varphi \\
\mathit{mapM}\langle f+g\rangle\ \varphi & = & \mathit{mapM}\langle f\rangle\ \varphi \oplus \mathit{mapM}\langle g\rangle\ \varphi \\
\mathit{mapM}\langle f\times g\rangle\ \varphi & = & \mathit{mapM}\langle f\rangle\ \varphi \otimes \mathit{mapM}\langle g\rangle\ \varphi \\
\mathit{mapM}\langle f\cdot g\rangle\ \varphi & = & \mathit{mapM}\langle f\rangle\ (\mathit{mapM}\langle g\rangle\ \varphi)
\end{array}$$

Abstractly speaking, $\mathit{mapM}\langle f\rangle$ defines the action on arrows in the Kleisli category induced by m. If we specialize m to *Id*, the identity monad, we obtain the 'ordinary' *map* operation. Since $\mathit{mapM}\langle f\rangle$ is the morphism part of a functor, it satisfies the following two functorial laws.

$$\begin{array}{rcll}
\mathit{mapM}\langle f\rangle\ \mathit{return} & = & \mathit{return} & \\
\mathit{mapM}\langle f\rangle\ (\varphi \circledcirc \psi) & = & \mathit{mapM}\langle f\rangle\ \varphi \circledcirc \mathit{mapM}\langle f\rangle\ \psi & \text{if } m \text{ is commutative}
\end{array}$$

Since the proof of the last law employs equation (17), it is restricted to commutative monads, as well. The monadic map is surprisingly versatile. It may be used, for example, to thread a monad through a structure, see [11].

$$\begin{array}{lcl}
\mathit{thread}\langle f\rangle & :: & (\mathit{Monad}\ m) \Rightarrow f\ (m\ a) \rightarrow m\ (f\ a) \\
\mathit{thread}\langle f\rangle & = & \mathit{mapM}\langle f\rangle\ \mathit{id}
\end{array}$$

What about the running time of $zip\langle f\rangle$? From the last equation of $zip\langle f\rangle$ we are lead to suspect that $zip\langle f\rangle$ suffers from the same problems as $reduce\langle f\rangle$. And this indeed is the case. Consider the specialization of $zip\langle f\rangle$ to *Tower* (note that $zip\langle Maybe\rangle = couple$).

$$zip\langle Tower\rangle \quad = \quad (return \oplus mapM\langle Tower\rangle\ couple \circledcirc zip\langle Tower\rangle) \circledcirc couple$$

It is not hard to see that zipping $(Recurse^n\ (End\ (Just^n a)), Recurse^n\ (End\ (Just^n b)))$ takes $\Theta(n^2)$ time. However, we can easily improve the efficiency by combining a monadic map with a zip.

$$\begin{aligned} zipWith\langle f\rangle\ \varphi \quad &= \quad mapM\langle f\rangle\ \varphi \circledcirc zip\langle f\rangle \\ zip\langle f\rangle \quad &= \quad zipWith\langle f\rangle\ return \end{aligned}$$

The derivation of $zipWith\langle f\rangle$ is again a simple exercise in program calculation. It pays off that we have defined $zip\langle f\rangle$ in terms of the admittedly abstract operators on procedures.

$$\begin{aligned}
&zipWith\langle Kt\rangle\ \varphi = mapM\langle Kt\rangle\ \varphi \circledcirc zip\langle Kt\rangle = return \circledcirc zip\langle Kt\rangle = zip\langle Kt\rangle \\
&zipWith\langle Id\rangle\ \varphi = mapM\langle Id\rangle\ \varphi \circledcirc zip\langle Id\rangle = \varphi \circledcirc return = \varphi \\
&zipWith\langle f+g\rangle\ \varphi \quad = \quad mapM\langle f+g\rangle\ \varphi \circledcirc zip\langle f+g\rangle \\
&\qquad = \quad (mapM\langle f\rangle\ \varphi \oplus mapM\langle g\rangle\ \varphi) \circledcirc (zip\langle f\rangle \oplus zip\langle g\rangle) \circledcirc couple \\
&\qquad = \quad (mapM\langle f\rangle\ \varphi \circledcirc zip\langle f\rangle \oplus mapM\langle g\rangle\ \varphi \circledcirc zip\langle g\rangle) \circledcirc couple \\
&\qquad = \quad (zipWith\langle f\rangle\ \varphi \oplus zipWith\langle g\rangle\ \varphi) \circledcirc couple
\end{aligned}$$

The derivation for products and compositions relies on the commutativity of *Maybe*.

$$\begin{aligned}
zipWith\langle f\times g\rangle\ \varphi \quad &= \quad mapM\langle f\times g\rangle\ \varphi \circledcirc zip\langle f\times g\rangle \\
&= \quad (mapM\langle f\rangle\ \varphi \otimes mapM\langle g\rangle\ \varphi) \circledcirc (zip\langle f\rangle \otimes zip\langle g\rangle) \circ transpose \\
&= \quad (mapM\langle f\rangle\ \varphi \circledcirc zip\langle f\rangle \otimes mapM\langle g\rangle\ \varphi \circledcirc zip\langle g\rangle) \circ transpose \\
&= \quad (zipWith\langle f\rangle\ \varphi \otimes zipWith\langle g\rangle\ \varphi) \circ transpose \\
zipWith\langle f\cdot g\rangle\ \varphi \quad &= \quad mapM\langle f\cdot g\rangle\ \varphi \circledcirc zip\langle f\cdot g\rangle \\
&= \quad mapM\langle f\rangle\ (mapM\langle g\rangle\ \varphi) \circledcirc mapM\langle f\rangle\ (zip\langle g\rangle) \circledcirc zip\langle f\rangle \\
&= \quad mapM\langle f\rangle\ (mapM\langle g\rangle\ \varphi \circledcirc zip\langle g\rangle) \circledcirc zip\langle f\rangle \\
&= \quad zipWith\langle f\rangle\ (zipWith\langle g\rangle\ \varphi)
\end{aligned}$$

Putting things together we obtain

$$\begin{aligned}
zip\langle f\rangle \quad &= \quad zipWith\langle f\rangle\ return \\
zipWith\langle f\rangle \quad &:: \quad (a\times b \to Maybe\ c) \to (f\ a \times f\ b \to Maybe\ (f\ c)) \\
zipWith\langle Kt\rangle\ \varphi \quad &= \quad (eq\langle t\rangle \to return \circ outl, const\ Nothing) \\
zipWith\langle Id\rangle\ \varphi \quad &= \quad \varphi \\
zipWith\langle f+g\rangle\ \varphi \quad &= \quad (zipWith\langle f\rangle\ \varphi \oplus zipWith\langle g\rangle\ \varphi) \circledcirc couple \\
zipWith\langle f\times g\rangle\ \varphi \quad &= \quad (zipWith\langle f\rangle\ \varphi \otimes zipWith\langle g\rangle\ \varphi) \circ transpose \\
zipWith\langle f\cdot g\rangle\ \varphi \quad &= \quad zipWith\langle f\rangle\ (zipWith\langle g\rangle\ \varphi)\ .
\end{aligned}$$

Interestingly, the two definitions of $zip\langle f\rangle$ are equivalent for arbitrary monads, not just for *Maybe*. To see why this is the case note that $zip\langle f\rangle$ and $zipWith\langle f\rangle$ are related by

$$zip\langle f\cdot g\rangle \quad = \quad zipWith\langle f\rangle\ (zip\langle g\rangle)\ .$$

Setting $g = Id$ yields the desired result.

5.2 Polybifunctorial functions

The definition of polybifunctorial functions contains little surprise. The main difference to polyfunctorial functions is that we must consider two projection functors, *Fst* and *Snd*, instead of one. Here is the definition of $bimap\langle f\rangle$, which describes the action of a bifunctor on arrows.

$$
\begin{array}{lcl}
bimap\langle f\rangle & :: & (a_1 \to b_1) \to (a_2 \to b_2) \to (f\ a_1\ a_2 \to f\ b_1\ b_2) \\
bimap\langle Kt\rangle\ \varphi_1\ \varphi_2 & = & id \\
bimap\langle Fst\rangle\ \varphi_1\ \varphi_2 & = & \varphi_1 \\
bimap\langle Snd\rangle\ \varphi_1\ \varphi_2 & = & \varphi_2 \\
bimap\langle f+g\rangle\ \varphi_1\ \varphi_2 & = & bimap\langle f\rangle\ \varphi_1\ \varphi_2 + bimap\langle g\rangle\ \varphi_1\ \varphi_2 \\
bimap\langle f\times g\rangle\ \varphi_1\ \varphi_2 & = & bimap\langle f\rangle\ \varphi_1\ \varphi_2 \times bimap\langle g\rangle\ \varphi_1\ \varphi_2 \\
bimap\langle f\cdot g\rangle\ \varphi_1\ \varphi_2 & = & map\langle f\rangle\ (bimap\langle g\rangle\ \varphi_1\ \varphi_2) \\
bimap\langle f\cdot(g_1,g_2)\rangle\ \varphi_1\ \varphi_2 & = & bimap\langle f\rangle\ (bimap\langle g_1\rangle\ \varphi_1\ \varphi_2)\ (bimap\langle g_2\rangle\ \varphi_1\ \varphi_2)
\end{array}
$$

Note the interplay of $map\langle f\rangle$ and $bimap\langle f\rangle$. For reasons of symmetry the following equation should be added to the definition of $map\langle f\rangle$.

$$map\langle f\cdot(g_1,g_2)\rangle\ \varphi \quad = \quad bimap\langle f\rangle\ (map\langle g_1\rangle\ \varphi)\ (map\langle g_2\rangle\ \varphi)$$

The example of $map\langle f\rangle$ illustrates a problem inherent with all polytypic definitions. If an unary functor, say, f is defined in terms of a ternary functor, then $map\langle f\rangle$ is undefined. The reason is simply that the relevant case $f\cdot(g_1,g_2,g_3)$ is missing in $map\langle f\rangle$'s definition. Moreover, since $f\cdot(g_1,\ldots,g_k)$ is an element of $F^{(n)}$ for all $k \geqslant 1$, we cannot define $map\langle f\rangle$ exhaustively. Clearly, further research is required here.

The generalization of polytypic reductions to binary functors is left as an exercise to the reader. Note that, in general, a reduction is a function of type $f\ a\ \ldots\ a \to a$.

6 Related and future work

The impetus for writing this article came while reading the article "Nested Datatypes" by R. Bird and L. Meertens [3]. 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 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 generic programmer, it also removes some of the redundancy present in the classical approach. In the polytypic programming language PolyP [10], 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 [4] introduced so-called *generalised folds*, which overcome some of the problems mentioned in the introduction to this article. 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 (ie folds and unfolds) [16]. 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 that maps a type functor to its base functor. Inventing a notation we define $F' = B$ for $F = \mu B$ (the operation $(\text{-})'$ 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

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

Ironically, hylomorphisms can be defined without any additions since their type does not involve type functors.

$$\begin{aligned}
hylo\langle f\rangle &:: (f\ a \to a) \to (b \to f\ b) \to (b \to a)\\
hylo\langle f\rangle\ \varphi\ \psi &= \varphi \circ map\langle f\rangle\ (hylo\langle f\rangle\ \varphi\ \psi) \circ \psi
\end{aligned}$$

Polytypic functions are values that depend on types. For that reason they cannot be expressed in languages such as Haskell or Standard ML. The question naturally arises as to whether polytypic definitions can be easily added to languages that incorporate *dependent types* such as Cayenne [2]. Now, since polytypic functions perform pattern matching on types, this would entail the addition of a **typecase** [1]. A **typecase** was, however, left out of Cayenne by design and the consequences of adding such a construct to the language are unclear (personal communication with L. Augustsson). The transformational approach taken here—to specialize a polytypic function for each given instance—appears to be simpler and also more efficient since the repeated matching of the type argument (which is statically known at compile time) is avoided.

Directions for future work suggest themselves. It remains to broaden the approach to include exponentials and higher-order polymorphism [13]. The former extension should be fairly straightforward, see [22] or [17]. The latter extension is far more challenging. To illustrate, consider the following generalization of rose trees.

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

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 [20]. 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 that 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 $(\forall a.f\ a \to g\ a) \to (\forall a.GRose\ f\ a \to GRose\ g\ a)$. 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 that 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 $(\forall a.\forall b.(a \to b) \to (f\ a \to f\ b)) \to (\forall a.\forall b.(a \to b) \to (GRose\ f\ a \to GRose\ f\ b))$. 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.

Acknowledgements

Thanks are due to Richard Bird for his helpful comments on an earlier draft of this article. I am also grateful to Philip Wadler and two anonymous referees for suggesting numerous improvements regarding contents and presentation.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[2] Lennart Augustsson. Cayenne – a language with dependent types. *SIGPLAN Notices*, 34(1):239–250, January 1999.

[3] Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, June 1998.

[4] Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 1999. To appear.

[5] Richard S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computer Science*. Springer-Verlag, 1988.

[6] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, March 1983.

[7] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Technical Report Memoranda Informatica 94-28, University of Twente, June 1994.

[8] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[9] Ralf Hinze. Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 1999. To appear.

[10] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97, Paris, France*, pages 470–482. ACM-Press, January 1997.

[11] Patrik Jansson and Johan Jeuring. PolyLib—A library of polytypic functions. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden*. Department of Computing Science, Chalmers University of Technology and Göteborg University, June 1998.

[12] Johan Jeuring and Patrik Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text 2nd International School on Advanced Functional Programming, Olympia, WA, USA*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer-Verlag, 1996.

[13] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer-Verlag, 1995.

[14] Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–280, 1990.

[15] Lambert Meertens. Calculate polytypically! In H. Kuchen and S.D. Swierstra, editors, *Proceedings 8th International Symposium on Programming Languages: Implementations, Logics, and Programs, PLILP'96, Aachen, Germany*, volume 1140 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, September 1996.

[16] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

[17] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Conference Record 7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 International Conference on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA*, pages 324–333. ACM-Press, June 1995.

[18] Erik Meijer and Johan Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *1st International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, volume 925 of *Lecture Notes in Computer Science*, pages 228–266. Springer-Verlag, Berlin, 1995.

[19] Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming, 6th Colloquium, Toulouse, France*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228, 1984.

[20] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[21] Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/onlinereport/`. February 1999.

[22] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings 6th ACM SIGPLAN/SIGARCH International Conference on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark*, pages 233–242. ACM-Press, June 1993.

[23] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78. ACM-Press, June 1990.

[24] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages, Sante Fe, New Mexico*, pages 1–14, January 1992.

[25] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming, Proceedings of the Båstad Spring School*, number 925 in Lecture Notes in Computer Science. Springer-Verlag, May 1995.