Efficient Generalized Folds

Ralf Hinze

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

Abstract. Fold operators capture a common recursion pattern over algebraic datatypes. A fold essentially replaces constructors by functions. However, if the datatype is parameterized, the corresponding fold operates on polymorphic functions which severely limits its applicability. In order to overcome this limitation R. Bird and R. Paterson [4] have proposed so-called generalized folds. We show how to define a variation of these folds by induction on the structure of datatype definitions. Unfortunately, for some datatypes generalized folds are less efficient than one would expect. We identify the source of inefficiency and explain how to remedy this shortcoming. While conceptually simple, our approach places high demands on the type system: it requires polymorphic recursion, rank-2 types, and a strong form of type constructor polymorphism.

1 Introduction

Fold operators are in every functional programmer's toolbox. In essence, a fold operator replaces constructors by functions. As an example, consider the following datatype of strings given in the functional language Haskell [17]:

$$\textbf{data}\ \mathit{String} = \mathit{Nil} \mid \mathit{Cons}\ \mathit{Char}\ \mathit{String}\ .$$

This definition simultaneously introduces a type constructor, called *String*, and two value constructors, called *Nil* and *Cons*, which have the following types:

$$\mathit{Nil} :: \mathit{String} \qquad \mathit{Cons} :: \mathit{Char} \to \mathit{String} \to \mathit{String}\ .$$

The fold operator associated with *String* is given by

$$\begin{aligned} &\mathit{foldS}\ \mathit{nil}\ \mathit{cons}\ \mathit{Nil} &&= \mathit{nil} \\ &\mathit{foldS}\ \mathit{nil}\ \mathit{cons}\ (\mathit{Cons}\ c\ s) &&= \mathit{cons}\ c\ (\mathit{foldS}\ \mathit{nil}\ \mathit{cons}\ s)\ . \end{aligned}$$

Thus, *foldS nil cons* takes a string and replaces *Nil* by *nil* and *Cons* by *cons*. To determine *foldS*'s type we simply take the signatures of the value constructors and systematically replace the type *String* by a type variable, say, *string*.

$$\mathit{foldS} :: (\mathit{string}) \to (\mathit{Char} \to \mathit{string} \to \mathit{string}) \to (\mathit{String} \to \mathit{string})$$

In a similar manner folds can be defined for all non-parametric datatypes and for all regular, parametric datatypes, see [15, 18]. While this scheme also works for general, parametric datatypes, the resulting fold operators are limited in applicability. Again, let us consider an example. The following definitions introduce perfectly balanced, binary leaf trees [7]:

$$\begin{aligned}
&\textbf{data}\ \mathit{Perfect}\ a = \mathit{Zero}\ a \mid \mathit{Succ}\ (\mathit{Perfect}\ (\mathit{Fork}\ a)) \\
&\textbf{data}\ \mathit{Fork}\ a \quad\ = \mathit{Fork}\ a\ a\ .
\end{aligned}$$

A perfect tree of height 2, for instance, is represented by

$$t_2 = \mathit{Succ}\ (\mathit{Succ}\ (\mathit{Zero}\ (\mathit{Fork}\ (\mathit{Fork}\ a_1\ a_2)\ (\mathit{Fork}\ a_3\ a_4))))\ ,$$

where a_1, a_2, a_3, and a_4 are the labels in the leaves. Note that the prefix $\mathit{Succ}^n\ \mathit{Zero}$ encodes the tree's height. The definition of *Perfect* is unusual in that the recursive call of *Perfect* on the right-hand side is not a copy of the declared type on the left-hand side. The type recursion is nested which is why types with this property are called *nested datatypes* [2]. Since *Perfect* and *Fork* are parameterized, the value constructors have polymorphic types:

$$\begin{aligned}
&\mathit{Zero} :: \forall a.a \rightarrow \mathit{Perfect}\ a && \mathit{Fork} :: \forall a.a \rightarrow a \rightarrow \mathit{Fork}\ a\ . \\
&\mathit{Succ} :: \forall a.\mathit{Perfect}\ (\mathit{Fork}\ a) \rightarrow \mathit{Perfect}\ a
\end{aligned}$$

Now, *Perfect*'s fold operator is given by (*sfold* stands for simple fold)

$$\begin{aligned}
&\mathit{sfoldP}\ \mathit{zero}\ \mathit{succ}\ (\mathit{Zero}\ a) = \mathit{zero}\ a \\
&\mathit{sfoldP}\ \mathit{zero}\ \mathit{succ}\ (\mathit{Succ}\ a) = \mathit{succ}\ (\mathit{sfoldP}\ \mathit{zero}\ \mathit{succ}\ a)\ .
\end{aligned}$$

As was to be expected, *sfoldP zero succ* takes a perfect tree and replaces *Zero* by *zero* and *Succ* by *succ*. For instance, *sfoldP zero succ* maps t_2 to

$$\mathit{succ}\ (\mathit{succ}\ (\mathit{zero}\ (\mathit{Fork}\ (\mathit{Fork}\ a_1\ a_2)\ (\mathit{Fork}\ a_3\ a_4))))\ .$$

To determine *sfoldP*'s type we take again the signatures of the value constructors and replace the type constructor *Perfect* by a type variable, say, *perfect*, which is implicitly universally qualified.

$$\begin{aligned}
\mathit{sfoldP}\ ::\ &(\forall a.a \rightarrow \mathit{perfect}\ a) \\
\rightarrow\ &(\forall a.\mathit{perfect}\ (\mathit{Fork}\ a) \rightarrow \mathit{perfect}\ a) \\
\rightarrow\ &(\forall a.\mathit{Perfect}\ a \rightarrow \mathit{perfect}\ a)
\end{aligned}$$

Remark. Note that *sfoldP* operates on polymorphic functions. Its type is a so-called *rank-2 type* [13] which is not legal Haskell.[1] Furthermore, note that *sfoldP* makes use of *polymorphic recursion* [16], since the recursive call operates at a different type. Since type inference is undecidable in the presence of polymorphic recursion [5], Haskell requires that an explicit type signature is provided. □

[1] A suitable extension, however, has been implemented in the Glasgow Haskell Compiler GHC [19] and in the Haskell interpreter Hugs [12], both of which accept the definition if we eliminate the quantifier in the result type changing the type signature to $(\forall a.a \rightarrow \mathit{perfect}\ a) \rightarrow (\forall a.\mathit{perfect}\ (\mathit{Fork}\ a) \rightarrow \mathit{perfect}\ a) \rightarrow (\mathit{Perfect}\ b \rightarrow \mathit{perfect}\ b)$.

To illustrate the use of *sfoldP* define

$$
\begin{aligned}
&\mathit{wrap} && :: a \to \mathit{List}\ a\\
&\mathit{wrap}\ a && = [\,a\,]\\
&\mathit{unforks} && :: \mathit{List}\ (\mathit{Fork}\ a) \to \mathit{List}\ a\\
&\mathit{unforks}\ [\,] && = [\,]\\
&\mathit{unforks}\ (\mathit{Fork}\ a_1\ a_2 : \mathit{ns}) && = a_1 : a_2 : \mathit{unforks}\ \mathit{ns}\ .
\end{aligned}
$$

Using *wrap* and *unforks* we can flatten a perfect tree into a list.

$$
\begin{aligned}
&\mathit{flatten} :: \mathit{Perfect}\ a \to \mathit{List}\ a\\
&\mathit{flatten} = \mathit{sfoldP}\ \mathit{wrap}\ \mathit{unforks}
\end{aligned}
$$

Note that *flatten* implements an iterative algorithm: the perfect tree is first converted into a singleton list, the trees in this list are then repeatedly split into subtrees until only singleton trees, that is, elements remain. Interestingly, the usual recursive method, which works by flattening the subtrees and appending the results, cannot be defined using *sfoldP*. The reason is simply that *sfoldP* replaces the constructors *Zero* and *Succ* but not the constructor *Fork*. For much the same reason *sfoldP* cannot be used to sum a perfect tree of integers.

These observations suggest to provide a fold on perfect trees which takes three arguments, *zero*, *succ*, and *fork*, and which maps t_2 to

$$
\mathit{succ}\ (\mathit{succ}\ (\mathit{zero}\ (\mathit{fork}\ (\mathit{fork}\ a_1\ a_2)\ (\mathit{fork}\ a_3\ a_4))))\ .
$$

To define this fold we require the so-called *mapping function* for perfect trees.

$$
\begin{aligned}
&\mathit{mapP} && :: (a \to b) \to (\mathit{Perfect}\ a \to \mathit{Perfect}\ b)\\
&\mathit{mapP}\ \varphi\ (\mathit{Zero}\ a) && = (\mathit{Zero} \cdot \varphi)\ a\\
&\mathit{mapP}\ \varphi\ (\mathit{Succ}\ a) && = (\mathit{Succ} \cdot \mathit{mapP}\ (\mathit{mapF}\ \varphi))\ a\\
&\mathit{mapF} && :: (a \to b) \to (\mathit{Fork}\ a \to \mathit{Fork}\ b)\\
&\mathit{mapF}\ \varphi\ (\mathit{Fork}\ a_1\ a_2) && = \mathit{Fork}\ (\varphi\ a_1)\ (\varphi\ a_2)
\end{aligned}
$$

Mapping functions can be defined completely generically for all parameterized datatypes, see Section 2. In general, a mapping function applies a given function to each element of type a in a given structure of type $F\ a$, returning a new structure with the same shape but different elements. The combination of a parameterized type and a mapping function is often referred to as a *functor*. Now, the new fold on perfect trees is given by

$$
\begin{aligned}
&\mathit{foldP} && :: (\forall a. a \to \mathit{perfect}\ a)\\
& && \to (\forall a. \mathit{perfect}\ (\mathit{fork}\ a) \to \mathit{perfect}\ a)\\
& && \to (\forall a. a \to a \to \mathit{fork}\ a)\\
& && \to (\forall a. \mathit{Perfect}\ a \to \mathit{perfect}\ a)\\
&\mathit{foldP}\ \mathit{zero}\ \mathit{succ}\ \mathit{fork}\ (\mathit{Zero}\ a) && = \mathit{zero}\ a\\
&\mathit{foldP}\ \mathit{zero}\ \mathit{succ}\ \mathit{fork}\ (\mathit{Succ}\ a) && = (\mathit{succ} \cdot \mathit{foldP}\ \mathit{zero}\ \mathit{succ}\ \mathit{fork}\\
& && \qquad \cdot\ \mathit{mapP}\ (\mathit{foldF}\ \mathit{fork}))\ a\\
&\mathit{foldF} && :: (\forall a. a \to a \to \mathit{fork}\ a)\\
& && \to (\forall a. \mathit{Fork}\ a \to \mathit{fork}\ a)\\
&\mathit{foldF}\ \mathit{fork}\ (\mathit{Fork}\ a_1\ a_2) && = \mathit{fork}\ a_1\ a_2\ .
\end{aligned}
$$

Consider the second equation of *foldP*. The fold of $a :: Perfect\ (Fork\ a)$ proceeds as follows: first the fold function on nodes is mapped on *Perfect* resulting in an element of type $Perfect\ (fork\ a)$, the recursive call to *foldP* yields an element of type $perfect\ (fork\ a)$, which *succ* finally maps to $perfect\ a$. Note that the type signature of *foldP* is derived from the constructors' types by replacing both *Perfect* and *Fork* by type variables. Now, using *foldP* we may implement, for instance, a recursive tree traversal.

$$rflatten = foldP\ id\ id\ (+\!\!+) \cdot mapP\ wrap$$

This idiom, a fold after a map $(foldP\ zero\ succ\ fork \cdot mapP\ \varphi)$, captures both iterative and recursive algorithms on perfect trees: *zero* and *succ* constitute the iterative part and *fork* and φ the recursive part.

While *foldP* is quite flexible, at least one problem remains: it cannot be used to sum a perfect tree of integers. The reason is that the type of *foldP* is too restricted: since *sum* has type $Perfect\ Int \to Int$, the only sensible choice for *perfect* is $perfect = \lambda s.Int$. Fixing *perfect* this way the first argument of *foldP* has type $\forall a.a \to Int$. However, a function of type $\forall a.a \to t$, where a does not appear in t, must necessarily ignore its argument.

Interestingly, this problem can be solved by generalizing the types of the folds — leaving the definitions unchanged. The central idea is to define functions of type $\forall a.F\ (x\ a) \to f\ a$ instead of $\forall a.F\ a \to f\ a$. The additional type constructor x provides the greater freedom of manœuvre required for many applications. We obtain the following *generalized folds* (Section 2 explains how the type signature is obtained)

$$\begin{aligned}
&gfoldP && :: (\forall a.x\ a \to perfect\ a)\\
& && \to (\forall a.perfect\ (fork\ a) \to perfect\ a)\\
& && \to (\forall a.x\ a \to x\ a \to x\ (fork\ a))\\
& && \to (\forall a.Perfect\ (x\ a) \to perfect\ a)\\
&gfoldP\ zero\ succ\ fork\ (Zero\ a) && = zero\ a\\
&gfoldP\ zero\ succ\ fork\ (Succ\ a) && = (succ \cdot gfoldP\ zero\ succ\ fork\\
& && \qquad \cdot mapP\ (kfoldF\ fork))\ a\\
&kfoldF && :: (\forall a.x\ a \to x\ a \to x\ (fork\ a))\\
& && \to (\forall a.Fork\ (x\ a) \to x\ (fork\ a))\\
&kfoldF\ fork\ (Fork\ a_1\ a_2) && = fork\ a_1\ a_2\ .
\end{aligned}$$

Again, let us examine how the fold of $a :: Perfect\ (Fork\ (x\ a))$ proceeds. The call $mapP\ (kfoldF\ fork)$ yields an element of type $Perfect\ (x\ (fork\ a))$, which the recursive invocation of *gfoldP* takes to $perfect\ (fork\ a)$, which in turn *succ* maps to $perfect\ a$.

Using *gfoldP* we can finally solve the problem of summing up a perfect tree.

$$\begin{aligned}
&sum :: Perfect\ Int \to Int\\
&sum = gfoldP\ id\ id\ (+)
\end{aligned}$$

Remark. In order to typecheck *sum* a strong form of *type constructor polymorphism* is required. Note that the call $gfoldP\ id\ id\ (+)$ must be of type $Perfect\ Int \rightarrow Int$. In order to unify this type with *gfoldP*'s declared type, $Perfect\ (x\ a) \rightarrow perfect\ a$, we must equate x and $perfect$ with $\lambda s.Int$. Unfortunately, the standard Haskell typechecker does not use higher-order unification but a kinded first-order unification [11]. The first-order unification of Int and $x\ a$, however, fails. To define *sum* in Haskell we are forced to define a type specialized variant of *gfoldP*. □

The rest of this paper is organized as follows. Section 2 shows how to define mapping functions and generalized folds generically for all parameterized datatypes. Section 3 presents several applications of generalized folds. Section 4 identifies a source of inefficiency in the definition of generalized folds and explains how to remedy this shortcoming. Our generalized folds are inspired by the folds of R. Bird and R. Paterson [4]. Section 5 explains the differences of the approaches and reviews related work.

2 Generalized folds

To define mapping functions and generalized folds generically for all parameterized datatypes we employ the framework of *polytypic programming* [6]. We will introduce the necessary background as we go along. For reasons of space we confine ourselves to datatypes parameterized by a single type variable; the extension to multiple parameters is straightforward. For a start, it is convenient to rewrite datatype definitions as *functor equations*:

$$\begin{aligned} Perfect &= Zero\ Id + Succ\ (Perfect \cdot Fork) \\ Fork &= Fork\ (Id \times Id)\ , \end{aligned}$$

where Id is the identity functor, $Id = \lambda s.s$, and $F_1 \cdot F_2$ denotes functor composition, $F_1 \cdot F_2 = \lambda s.F_1\ (F_2\ s)$. Sum and product are defined pointwise: $C_1\ F_1 + \cdots + C_n\ F_n = \lambda s.C_1\ (F_1\ s) + \cdots + C_n\ (F_n\ s)$ and $F_1 \times \cdots \times F_n = \lambda s.F_1\ s \times \cdots \times F_n\ s$. We agree upon that '$\cdot$' binds more tightly than '$\times$', which in turn takes precedence over '$+$'. As another example, consider the ubiquitous datatype of polymorphic lists.

$$\mathbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

The corresponding functor equation is

$$List = Nil\ (K\ 1) + Cons\ (Id \times List)\ .$$

Here, $K\ t$ denotes the constant functor $\lambda s.t$ and 1 is the unit type. In general, a functor equation has the form $f = F$ where f is a functor variable and F a functor expression. Functor expressions are formed according to the grammar

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

where T is the set of primitive types such as 1, Int, etc and C is the set of constructor names. The translation of datatype definitions into functor equations is straightforward. Note that the resulting equations have the general form

$$f = C_1\ (F_{11} \times \cdots \times F_{1k_1}) + \cdots + C_n\ (F_{n1} \times \cdots \times F_{nk_n})\ ,$$

where each F_{ij} is either $K\ T$ for some T, Id, or a composition of functor variables $f_1 \cdot \cdots \cdot f_n$.

In the sequel we require the following general combining forms on sums, $C_1\ F_1 + \cdots + C_n\ F_n$, and on products, $F_1 \times \cdots \times F_n$.

$$\begin{aligned}
(\varphi_1 \triangledown \cdots \triangledown \varphi_n)\ (C_i\ a_i) &= \varphi_i\ a_i \\
\varphi_1 + \cdots + \varphi_n &= (C_1 \cdot \varphi_1) \triangledown \cdots \triangledown (C_n \cdot \varphi_n) \\
(\varphi_1 \times \cdots \times \varphi_n)\ (a_1, \ldots, a_n) &= (\varphi_1\ a_1, \ldots, \varphi_n\ a_n)
\end{aligned}$$

The n-ary operator $(\triangledown)$ performs a case analysis; $(+)$ and $(\times)$ establish the type constructors of the same name as n-ary functors. For unary sums we write $\varphi_1 \cdot unC_1$ instead of $\varphi_1 \triangledown \cdots \triangledown \varphi_n$ and $C_1 \cdot \varphi_1 \cdot unC_1$ instead of $\varphi_1 + \cdots + \varphi_n$.

Now, in order to synthesize the mapping function for a given datatype or for a system of mutually recursive datatypes, we simply take the corresponding system of functor equations and transform it into a system of function definitions:

$$\begin{array}{c} f_1 = F_1 \\ \vdots \\ f_n = F_n \end{array} \quad \Longrightarrow \quad \begin{array}{c} map\langle f_1\rangle = map\langle F_1\rangle \\ \vdots \\ map\langle f_n\rangle = map\langle F_n\rangle \end{array}\ ,$$

where $map\langle F\rangle$ is given by

$$\begin{aligned}
map\langle K\ T\rangle\ \varphi &= id \\
map\langle Id\rangle\ \varphi &= \varphi \\
map\langle C_1\ F_1 + \cdots + C_n\ F_n\rangle\ \varphi &= map\langle F_1\rangle\ \varphi + \cdots + map\langle F_n\rangle\ \varphi \\
map\langle F_1 \times \cdots \times F_n\rangle\ \varphi &= map\langle F_1\rangle\ \varphi \times \cdots \times map\langle F_n\rangle\ \varphi \\
map\langle F_1 \cdot F_2\rangle\ \varphi &= map\langle F_1\rangle\ (map\langle F_2\rangle\ \varphi)\ .
\end{aligned}$$

Note that $map\langle F\rangle$ is defined inductively on the structure of functor expressions. For $Perfect$ and $Fork$ we obtain the following mapping functions.

$$\begin{aligned}
Perfect &= Zero\ Id + Succ\ (Perfect \cdot Fork) \\
Fork &= Fork\ (Id \times Id)
\end{aligned}$$

$$\Downarrow$$

$$\begin{aligned}
map\langle Perfect\rangle\ \varphi &= \varphi + map\langle Perfect\rangle\ (map\langle Fork\rangle\ \varphi) \\
map\langle Fork\rangle\ \varphi &= Fork \cdot (\varphi \times \varphi) \cdot unFork
\end{aligned}$$

If we strip off the abstract clothing, we find that these definitions correspond exactly to the ones given in the introduction to this paper.

By now we have the necessary prerequisites at hand to define generalized folds in a systematic way. In general, a generalized fold takes a structure and replaces *each* constructor within that structure by a corresponding function. By contrast, a simple fold only replaces the constructors listed in the respective datatype definition. As in Section 1 we will introduce generalized folds in two steps. First, we present a simply typed variant where each fold has type $\forall a.F\ a \rightarrow f\ a$. Second, we explain the modifications necessary to generalize the types to $\forall a.F\ (x\ a) \rightarrow f\ a$. To simplify the presentation let us assume that the functions that replace the constructors are fixed and given in advance. We denote the function that replaces the constructor C by $\bar{C}$. It should be stressed, however, that this is merely a technical convenience, which relieves us from making the passing of the $\bar{C}$'s explicit. Now, the simply typed fold is defined as follows.

$$\begin{array}{lcl} f_1 = F_1 & & \mathit{fold}\langle f_1\rangle = \mathit{fold}\langle F_1\rangle \\ \vdots & \Longrightarrow & \vdots \\ f_n = F_n & & \mathit{fold}\langle f_n\rangle = \mathit{fold}\langle F_n\rangle \ , \end{array}$$

where $\mathit{fold}\langle F\rangle$ is given by

$$\begin{array}{ll} \mathit{fold}\langle K\ T\rangle & = \mathit{id} \\ \mathit{fold}\langle \mathit{Id}\rangle & = \mathit{id} \\ \mathit{fold}\langle C_1\ F_1 + \cdots + C_n\ F_n\rangle & = \bar{C}_1 \cdot \mathit{fold}\langle F_1\rangle \mathbin{\triangledown} \cdots \mathbin{\triangledown} \bar{C}_n \cdot \mathit{fold}\langle F_n\rangle \\ \mathit{fold}\langle F_1 \times \cdots \times F_n\rangle & = \mathit{fold}\langle F_1\rangle \times \cdots \times \mathit{fold}\langle F_n\rangle \\ \mathit{fold}\langle F_1 \cdot F_2\rangle & = \mathit{fold}\langle F_1\rangle \cdot \mathit{map}\langle F_1\rangle\ (\mathit{fold}\langle F_2\rangle) \ . \end{array}$$

The third line captures the essential of a fold: the constructors $C_1, \ldots, C_n$ are replaced by the corresponding functions $\bar{C}_1, \ldots, \bar{C}_n$. The remaining equations preserve the structure of the folded value. Consequently, if the constructors are replaced by themselves, $\mathit{fold}\langle F\rangle$ amounts to the identity function. Now, for *Perfect* and *Fork* we obtain the following equations.

$$\begin{array}{ll} \mathit{Perfect} & = \mathit{Zero}\ \mathit{Id} + \mathit{Succ}\ (\mathit{Perfect} \cdot \mathit{Fork}) \\ \mathit{Fork} & = \mathit{Fork}\ (\mathit{Id} \times \mathit{Id}) \end{array}$$

$$\Downarrow$$

$$\begin{array}{ll} \mathit{fold}\langle \mathit{Perfect}\rangle & = \mathit{zero} \mathbin{\triangledown} \mathit{succ} \cdot \mathit{fold}\langle \mathit{Perfect}\rangle \cdot \mathit{map}\langle \mathit{Perfect}\rangle\ (\mathit{fold}\langle \mathit{Fork}\rangle) \\ \mathit{fold}\langle \mathit{Fork}\rangle & = \mathit{fork} \cdot \mathit{unFork} \end{array}$$

If we strip off the abstract clothing and if we explicate the passing of the parameters *zero*, *succ*, and *fork*, we obtain the definitions given in the introduction.

Turning to the types of the folds let us abbreviate $\forall a.F_1\ a \rightarrow F_2\ a$ by $F_1 \dot{\rightarrow} F_2$. Using this shorthand the types of the constructors read

$$\begin{array}{llll} \mathit{Zero} :: \mathit{Id} \dot{\rightarrow} \mathit{Perfect} & & \mathit{Fork} :: \mathit{Id} \times \mathit{Id} \dot{\rightarrow} \mathit{Fork} \ . \\ \mathit{Succ} :: \mathit{Perfect} \cdot \mathit{Fork} \dot{\rightarrow} \mathit{Perfect} & & \end{array}$$

Recall from the introduction that the types of the corresponding functions $\bar{C}$ are obtained by replacing the type constructors by type variables. Denote by $\bar{F}$ the functor expression thus obtained. For our running example we obtain

$$\begin{array}{llll} \mathit{zero} :: \mathit{Id} \dot{\rightarrow} \mathit{perfect} & & \mathit{fork} :: \mathit{Id} \times \mathit{Id} \dot{\rightarrow} \mathit{fork} \ . \\ \mathit{succ} :: \mathit{perfect} \cdot \mathit{fork} \dot{\rightarrow} \mathit{perfect} & & \end{array}$$

Finally, the signatures of the folds are given by

$$\mathit{fold}\langle \mathit{Perfect}\rangle :: \mathit{Perfect} \dot{\to} \mathit{perfect} \qquad \mathit{fold}\langle \mathit{Fork}\rangle :: \mathit{Fork} \dot{\to} \mathit{fork} \enspace .$$

In general, $\mathit{fold}\langle F\rangle :: F \dot{\to} \bar{F}$ replaces each value constructor $C :: E \dot{\to} F$ by $\bar{C} :: \bar{E} \dot{\to} \bar{F}$.

We have already discussed that the type of $\mathit{fold}\langle F\rangle$ is too restricted for many applications and that this limitation can be overcome by generalizing its type to $F \cdot x \dot{\to} \bar{F}$. Interestingly this generalization only induces changes at the type level. The definition of $\mathit{fold}\langle F\rangle$ remains essentially the same. Let us 'evaluate' the composition $F \cdot x$. Using the fact that functor composition distributes leftward through sums and products,

$$\begin{aligned} (C_1\ F_1 + \cdots + C_n\ F_n) \cdot x &= C_1\ (F_1 \cdot x) + \cdots + C_n\ (F_n \cdot x) \\ (F_1 \times \cdots \times F_n) \cdot x &= (F_1 \cdot x) \times \cdots \times (F_n \cdot x) \end{aligned}$$

we can push the functor x to the arguments of the constructors. Recall that the argument type of a constructor equals either $K\ T$, Id, or $f_1 \cdot f_2 \cdot \cdots \cdot f_n$. The first two cases are simple: we have $K\ T \cdot x = K\ T$ and $\mathit{Id} \cdot x = x$. The main question is how to fold an argument of type $f_1 \cdot f_2 \cdot \cdots \cdot f_n \cdot x$. There are several choices. For instance, we could apply gfold to f_n. In this case the remaining functors can be reduced using fold. Conversely, if we apply gfold to f_1, the functor x must be propagated through $f_2, \ldots, f_n$. This would entail the definition of a second fold operation of type $F \cdot x \dot{\to} x \cdot \bar{F}$, which will be called kfold. Rather arbitrarily, we decide for the second alternative. Now, the two generalized folds are given by

$$\begin{aligned} \mathit{gfold}\langle F_1 \cdot F_2\rangle &= \mathit{gfold}\langle F_1\rangle \cdot \mathit{map}\langle F_1\rangle\ (\mathit{kfold}\langle F_2\rangle) \\ \mathit{kfold}\langle F_1 \cdot F_2\rangle &= \mathit{kfold}\langle F_1\rangle \cdot \mathit{map}\langle F_1\rangle\ (\mathit{kfold}\langle F_2\rangle) \enspace . \end{aligned}$$

The remaining equations are as in the definition of $\mathit{fold}\langle F\rangle$. Thus, the code is the same for each variant. Only the functions $\bar{C}$ have different types. Consider the generalized fold for our running example.

$$\begin{aligned} &\mathit{gfold}\langle \mathit{Perfect}\rangle :: \mathit{Perfect} \cdot x \dot{\to} \mathit{perfect} \\ &\mathit{gfold}\langle \mathit{Perfect}\rangle = \mathit{zero} \mathbin{\triangledown} \mathit{succ} \cdot \mathit{gfold}\langle \mathit{Perfect}\rangle \cdot \mathit{map}\langle \mathit{Perfect}\rangle\ (\mathit{kfold}\langle \mathit{Fork}\rangle) \\ &\mathit{kfold}\langle \mathit{Fork}\rangle :: \mathit{Fork} \cdot x \dot{\to} x \cdot \mathit{fork} \\ &\mathit{kfold}\langle \mathit{Fork}\rangle = \mathit{fork} \cdot \mathit{unFork} \end{aligned}$$

The types of zero, succ, and fork are given by

$$\begin{aligned} &\mathit{zero} :: x \dot{\to} \mathit{perfect} &\qquad& \mathit{fork} :: x \times x \dot{\to} x \cdot \mathit{fork} \enspace . \\ &\mathit{succ} :: \mathit{perfect} \cdot \mathit{fork} \dot{\to} \mathit{perfect} \end{aligned}$$

In general, $\mathit{gfold}\langle F\rangle :: F \cdot x \dot{\to} \bar{F}$ replaces each value constructor $C :: E \to F$ by $\bar{C} :: x \circ \bar{E} \dot{\to} \bar{F}$ and $\mathit{kfold}\langle F\rangle :: F \cdot x \dot{\to} x \cdot \bar{F}$ replaces each value constructor $C :: E \to F$ by $\bar{C} :: x \bullet \bar{E} \dot{\to} x \cdot \bar{F}$. The operations $x \circ F$ and $x \bullet F$ are given by

F	$x \circ F$	$x \bullet F$
$K\ T$	$K\ T$	$K\ T$
Id	x	x
$f_1 \cdot \cdots \cdot f_n$	$f_1 \cdot \cdots \cdot f_n$	$x \cdot f_1 \cdot \cdots \cdot f_n$

.

On products $x \circ F$ and $x \bullet F$ are defined pointwise.

3 Examples

Let us consider some examples in more detail.

Polymorphic lists. The generalized fold on polymorphic lists is given by

$$\begin{aligned} gfold\langle List\rangle &:: List \cdot x \dot{\to} list \\ gfold\langle List\rangle &= nil \mathbin{\triangledown} cons \cdot (id \times gfold\langle List\rangle) \ . \end{aligned}$$

The functions *nil* and *cons* have the following types.

$$nil :: K\ 1 \dot{\to} list \qquad cons :: x \times list \dot{\to} list$$

From these data we can easily derive the Haskell definition of $gfold\langle List\rangle$.

$$\begin{aligned} &gfoldL && :: (\forall a.list\ a) \\ & && \to (\forall a.x\ a \to list\ a \to list\ a) \\ & && \to (\forall a.List\ (x\ a) \to list\ a) \\ &gfoldL\ nil\ cons\ Nil && = nil \\ &gfoldL\ nil\ cons\ (Cons\ a\ as) && = cons\ a\ (gfoldL\ nil\ cons\ as) \end{aligned}$$

This definition looks exactly like the standard fold on lists, usually called *foldr*. And, in fact, if we specialize the type of *gfoldL* setting $x = K\ t$ and $list = K\ u$, we obtain *foldr*'s signature: $(u) \to (t \to u \to u) \to (List\ t \to u)$. Conversely, each application of *gfoldL* can be implemented in terms of *foldr*. Thus, for polymorphic lists folds and generalized folds coincide. This is quite a pleasant finding and it holds for all regular datatypes.

Bushy lists. As a more intricate example consider the type of bushy lists [2].

$$\mathbf{data}\ Bush\ a = Nil \mid Cons\ a\ (Bush\ (Bush\ a))$$

The type is similar to *List* except that, at each step down the list, entries are 'bushed': the n-th entry of a list of type $Bush\ a$ has type $Bush^n\ a$. Despite this bewildering form of type recursion the derivation of the generalized fold proceeds entirely mechanically.

$$Bush = Nil\ (K\ 1) + Cons\ (Id \times Bush \cdot Bush)$$
$$\Downarrow$$
$$\begin{aligned} gfold\langle Bush\rangle &:: Bush \cdot x \dot{\to} bush \\ gfold\langle Bush\rangle &= gnil \mathbin{\triangledown} gcons \cdot (id \times gfold\langle Bush\rangle \cdot map\langle Bush\rangle\ (kfold\langle Bush\rangle)) \\ kfold\langle Bush\rangle &:: Bush \cdot x \dot{\to} x \cdot bush \\ kfold\langle Bush\rangle &= knil \mathbin{\triangledown} kcons \cdot (id \times kfold\langle Bush\rangle \cdot map\langle Bush\rangle\ (kfold\langle Bush\rangle)) \end{aligned}$$

Since *Bush* is a *non-linear* nested datatype [4] — the recursive call of *Bush* contains another call to *Bush* — we require both $gfold\langle Bush\rangle$ and $kfold\langle Bush\rangle$. Consequently, two sets of functions must be supplied.

$$\begin{array}{ll} gnil \;::\; K\ 1 \dot{\to} bush & knil \;::\; K\ 1 \dot{\to} x \cdot bush \\ gcons :: x \times bush \cdot bush \dot{\to} bush & kcons :: x \times x \cdot bush \cdot bush \dot{\to} x \cdot bush \end{array}$$

The corresponding Haskell definitions, which are listed in Figure 1, make the passing of *gnil*, *gcons*, *knil*, and *kcons* explicit. Fortunately, the complicated

$$\begin{array}{ll}
gfoldB & ::\ (\forall a.bush\ a) \\
 & \to (\forall a.x\ a \to bush\ (bush\ a) \to bush\ a) \\
 & \to (\forall a.x\ (bush\ a)) \\
 & \to (\forall a.x\ a \to x\ (bush\ (bush\ a)) \to x\ (bush\ a)) \\
 & \to (\forall a.Bush\ (x\ a) \to bush\ a) \\
gfoldB\ gn\ gc\ kn\ kc\ Nil & =\ gn \\
gfoldB\ gn\ gc\ kn\ kc\ (Cons\ a\ as) & =\ gc\ a\ (gfoldB\ gn\ gc\ kn\ kc\ (mapB\ (kfoldB\ kn\ kc)\ as)) \\
kfoldB & ::\ (\forall a.x\ (bush\ a)) \\
 & \to (\forall a.x\ a \to x\ (bush\ (bush\ a)) \to x\ (bush\ a)) \\
 & \to (\forall a.Bush\ (x\ a) \to x\ (bush\ a)) \\
kfoldB\ kn\ kc\ Nil & =\ kn \\
kfoldB\ kn\ kc\ (Cons\ a\ as) & =\ kc\ a\ (kfoldB\ kn\ kc\ (mapB\ (kfoldB\ kn\ kc)\ as))
\end{array}$$

Fig. 1. Generalized fold for *Bush*.

gfoldB is not needed for most applications. For instance, flattening a bush or summing up a bush can be accomplished using the simpler *kfoldB*.

$$\begin{array}{l} flatten :: Bush\ a \to List\ a \\ flatten = kfoldB\ [\,]\ \sigma\ \textbf{where}\ \sigma\ a\ aas = a : concat\ aas \\ sum \quad :: Bush\ Int \to Int \\ sum \quad = kfoldB\ 0\ (+) \end{array}$$

Note that $fold\langle F\rangle$ is a type instance of $kfold\langle F\rangle$ with x specialized to *Id*. In the first example *kfoldB* is used as such (that is, $x = Id$ and $bush = List$). In the second example x is instantiated to the constant functor $K\ Int$. In general, if we instantiate x in $kfold\langle F\rangle$'s type to $K\ t$, we obtain the signature of a *reduction* or *crush* [14]. A reduction is a function of type $F\ t \to t$, which collapses a structure of values of type t into a single value of type t.

De Bruijn notation. In the de Bruijn encoding of λ-terms a bound variable is represented by a natural number, indicating the 'distance' from the occurrence to its binding place. De Bruijn terms can be defined using a nested type [3].

$$\begin{array}{l} \textbf{data}\ Term\ a = Var\ a \mid App\ (Term\ a)\ (Term\ a) \mid Lam\ (Term\ (Incr\ a)) \\ \textbf{data}\ Incr\ a \;\; = Zero \mid Succ\ a \end{array}$$

The corresponding functor equations are

$$\begin{aligned} Term &= Var\ Id + App\ (Term \times Term) + Lam\ (Term \cdot Incr) \\ Incr &= Zero\ (K\ 1) + Succ\ Id \ . \end{aligned}$$

The generalized fold on *Term* has the following form.

$$\begin{aligned} gfold\langle Term\rangle &:: Term \cdot x \dot{\to} term \\ gfold\langle Term\rangle &= var \,\nabla\, app \cdot (gfold\langle Term\rangle \times gfold\langle Term\rangle) \\ &\qquad \nabla\, lam \cdot (gfold\langle Term\rangle \cdot map\langle Term\rangle\ (kfold\langle Incr\rangle)) \\ kfold\langle Incr\rangle &:: Incr \cdot x \dot{\to} x \cdot incr \\ kfold\langle Incr\rangle &= zero \,\nabla\, succ \end{aligned}$$

The functions *var*, *app*, *lam*, *zero*, and *succ* have the following types.

$$\begin{aligned} var &:: x \dot{\to} term & zero &:: K\ 1 \dot{\to} x \cdot incr \\ app &:: term \times term \dot{\to} term & succ &:: x \dot{\to} x \cdot incr \ . \\ lam &:: term \cdot incr \dot{\to} term \end{aligned}$$

The derivation of the corresponding Haskell program is left as an exercise to the reader. The functor *Term* is also a monad, with *Var* as its unit and *join* given by the following application of *gfoldT*.

$$\begin{aligned} join &:: Term\ (Term\ a) \to Term\ a \\ join &= gfoldT\ id\ App\ Lam\ (Var\ Zero)\ (mapT\ Succ) \end{aligned}$$

Using *join* one can, for instance, define β-reduction of de Bruijn terms.

$$\begin{aligned} &reduce && :: Term\ a \to Term\ a \\ &reduce\ (App\ (Lam\ s)\ t) && = join\ (mapT\ (subst\ t)\ s) \\ &subst && :: Term\ a \to Incr\ a \to Term\ a \\ &subst\ t\ Zero && = t \\ &subst\ t\ (Succ\ a) && = Var\ a \end{aligned}$$

For an explanation of *join* and *reduce* the interested reader is referred to [3].

4 Efficient generalized folds

Since fold operators replace constructors by functions, one would intuitively expect that $fold\langle F\rangle$ and friends take time proportional to the size of the input (plus the time to evaluate the function applications). Unfortunately, this is not the case as the following example shows. Assume that we want to determine the free variables of a de Bruijn term. This function is implemented by

$$\begin{aligned} free &:: Term\ a \to List\ a \\ free &= gfoldT\ wrap\ (+\!\!+)\ concat\ [\,]\ wrap \ . \end{aligned}$$

Now, consider the evaluation of $\mathit{free}\ (\mathit{Lam}^n\ (\mathit{Var}\ (\mathit{Succ}^n\ a)))$: in each recursive step a subexpression of the form $\mathit{Lam}^{i+1}\ (\mathit{Var}\ (\mathit{Succ}^{i+1}\ x))$ is mapped to $\mathit{concat}\ (\mathit{Lam}^i\ (\mathit{Var}\ (\mathit{Succ}^i\ (\mathit{wrap}\ x))))$. Since the call $\mathit{map}\langle \mathit{Term}\rangle\ (\mathit{kfold}\langle \mathit{Incr}\rangle)$, which replaces Succ by wrap, takes linear time, we have a total running time of $n + (n-1) + \cdots + 2 + 1 = \Theta(n^2)$. This is bad news. Since fold operators are fundamental combining forms, such a lack of efficiency is clearly unacceptable.

Fortunately, it is relatively easy to improve the efficiency of $\mathit{fold}\langle F\rangle$ and friends. Consider the definition of $\mathit{fold}\langle F_1 \cdot F_2\rangle$.

$$\mathit{fold}\langle F_1 \cdot F_2\rangle = \mathit{fold}\langle F_1\rangle \cdot \mathit{map}\langle F_1\rangle\ (\mathit{fold}\langle F_2\rangle)$$

The example above shows that the call to $\mathit{map}\langle F_1\rangle$ causes the efficiency problem. This observation suggests to define a function which combines a fold with a map.

$$\mathit{FOLD}\langle F\rangle\ \varphi = \mathit{fold}\langle F\rangle \cdot \mathit{map}\langle F\rangle\ \varphi\ \ .$$

The derivation of $\mathit{FOLD}\langle F\rangle$ is a nice exercise in program calculation and proceeds almost mechanically. For the two base cases we have

$$\begin{aligned}
&\mathit{FOLD}\langle K\ t\rangle\ \varphi = \mathit{fold}\langle K\ t\rangle \cdot \mathit{map}\langle K\ t\rangle\ \varphi = \mathit{id} \cdot \mathit{id} = \mathit{id}\ \ , \\
&\mathit{FOLD}\langle \mathit{Id}\rangle\ \varphi = \mathit{fold}\langle \mathit{Id}\rangle \cdot \mathit{map}\langle \mathit{Id}\rangle\ \varphi = \mathit{id} \cdot \varphi = \varphi\ \ .
\end{aligned}$$

The derivation for sums and products is left as an exercise to the reader. For the composition of functors we obtain

$$\begin{aligned}
\mathit{FOLD}\langle F_1 \cdot F_2\rangle\ \varphi &= \mathit{fold}\langle F_1 \cdot F_2\rangle \cdot \mathit{map}\langle F_1 \cdot F_2\rangle\ \varphi \\
&= \mathit{fold}\langle F_1\rangle \cdot \mathit{map}\langle F_1\rangle\ (\mathit{fold}\langle F_2\rangle) \cdot \mathit{map}\langle F_1\rangle\ (\mathit{map}\langle F_2\rangle\ \varphi) \\
&= \mathit{fold}\langle F_1\rangle \cdot \mathit{map}\langle F_1\rangle\ (\mathit{fold}\langle F_2\rangle \cdot \mathit{map}\langle F_2\rangle\ \varphi) \\
&= \mathit{FOLD}\langle F_1\rangle\ (\mathit{FOLD}\langle F_2\rangle\ \varphi)\ \ ,
\end{aligned}$$

which shows that we have succeeded in eliminating the call to map. To summarize

$$\begin{aligned}
&\mathit{FOLD}\langle K\ T\rangle\ \varphi && = \mathit{id} \\
&\mathit{FOLD}\langle \mathit{Id}\rangle\ \varphi && = \varphi \\
&\mathit{FOLD}\langle C_1\ F_1 + \cdots + C_n\ F_n\rangle\ \varphi && = \bar{C}_1 \cdot \mathit{FOLD}\langle F_1\rangle\ \varphi\ \triangledown \cdots \triangledown\ \bar{C}_n \cdot \mathit{FOLD}\langle F_n\rangle\ \varphi \\
&\mathit{FOLD}\langle F_1 \times \cdots \times F_n\rangle\ \varphi && = \mathit{FOLD}\langle F_1\rangle\ \varphi \times \cdots \times \mathit{FOLD}\langle F_n\rangle\ \varphi \\
&\mathit{FOLD}\langle F_1 \cdot F_2\rangle\ \varphi && = \mathit{FOLD}\langle F_1\rangle\ (\mathit{FOLD}\langle F_2\rangle\ \varphi)\ \ .
\end{aligned}$$

The derivation of $\mathit{GFOLD}\langle F\rangle$ and $\mathit{KFOLD}\langle F\rangle$ proceeds in an analogous fashion. Only the equations for $F_1 \cdot F_2$ are different.

$$\begin{aligned}
\mathit{GFOLD}\langle F_1 \cdot F_2\rangle\ \varphi &= \mathit{GFOLD}\langle F_1\rangle\ (\mathit{KFOLD}\langle F_2\rangle\ \varphi) \\
\mathit{KFOLD}\langle F_1 \cdot F_2\rangle\ \varphi &= \mathit{KFOLD}\langle F_1\rangle\ (\mathit{KFOLD}\langle F_2\rangle\ \varphi)
\end{aligned}$$

Let us determine the types of the new folds. The central insight is that $\mathit{FOLD}\langle F\rangle$ and friends are *higher-order folds*, ie they take folds to folds. To see why this is the case note that we can easily derive the following properties from the specification.

$$\begin{aligned} \mathit{FOLD}\langle F\rangle\ (\mathit{fold}\langle g\rangle) &= \mathit{fold}\langle F\cdot g\rangle \\ \mathit{GFOLD}\langle F\rangle\ (\mathit{kfold}\langle g\rangle) &= \mathit{gfold}\langle F\cdot g\rangle \\ \mathit{KFOLD}\langle F\rangle\ (\mathit{kfold}\langle g\rangle) &= \mathit{kfold}\langle F\cdot g\rangle \end{aligned}$$

These properties motivate in turn the following type signatures.

$$\begin{aligned} &\mathit{FOLD}\langle F\rangle &&:: (g \stackrel{.}{\to} \bar{g}) \to (F\cdot g \stackrel{.}{\to} \bar{F}\cdot\bar{g}) \\ &\mathit{GFOLD}\langle F\rangle &&:: (g\cdot x \stackrel{.}{\to} x\cdot\bar{g}) \to (F\cdot g\cdot x \stackrel{.}{\to} \bar{F}\cdot\bar{g}) \\ &\mathit{KFOLD}\langle F\rangle &&:: (g\cdot x \stackrel{.}{\to} x\cdot\bar{g}) \to (F\cdot g\cdot x \stackrel{.}{\to} x\cdot\bar{F}\cdot\bar{g}) \end{aligned}$$

Note that the type variables g, $\bar{g}$, and x are universally quantified in each case. In order to check that the definition of $\mathit{GFOLD}\langle F\rangle$ is well-typed we propagate the functor $g\cdot x$ to the arguments of the constructors. Consider, for instance, the constructor $C :: E \stackrel{.}{\to} F$ with $E = H_1 \times \cdots \times H_k$. Each argument type is either $K\ T\cdot g\cdot x$, $\mathit{Id}\cdot g\cdot x$, or $f_1\cdot\cdots\cdot f_n\cdot g\cdot x$. Now, in the second case, for instance, $\mathit{GFOLD}\langle \mathit{Id}\rangle\ \varphi = \varphi$ takes $g\cdot x$ to $x\cdot\bar{g}$. The remaining cases are listed below.

H_i	$H_i\cdot g\cdot x$		$(x\circ\bar{H}_i)\cdot\bar{g}$
$K\ T$	$K\ T\cdot g\cdot x = K\ T\cdot\bar{g}$	$\longmapsto$	$K\ T\cdot\bar{g}$
Id	$g\cdot x$	$\longmapsto$	$x\cdot\bar{g}$
$f_1\cdot\cdots\cdot f_n$	$f_1\cdot\cdots\cdot f_n\cdot g\cdot x$	$\longmapsto$	$\bar{f}_1\cdot\cdots\cdot\bar{f}_n\cdot\bar{g}$

Consequently, the arguments of the function $\bar{C} :: x\circ\bar{E} \stackrel{.}{\to} \bar{F}$ have type $(x\circ\bar{E})\cdot\bar{g}$ which $\bar{C}$ then takes to $\bar{F}\cdot\bar{g}$ as desired. The first-order folds can be easily defined in terms of the higher-order folds: we have, for instance, $\mathit{gfold}\langle F\rangle = \mathit{GFOLD}\langle F\rangle\ \mathit{id}$. On the type level g and $\bar{g}$ are instantiated to Id and id has type $x \stackrel{.}{\to} x$.

Let us finally consider an example. The higher-order fold for perfect trees is

$$\begin{aligned} &\mathit{GFOLD}\langle \mathit{Perfect}\rangle &&:: (g\cdot x \stackrel{.}{\to} x\cdot\bar{g}) \to (\mathit{Perfect}\cdot g\cdot x \stackrel{.}{\to} \mathit{perfect}\cdot\bar{g}) \\ &\mathit{GFOLD}\langle \mathit{Perfect}\rangle\ \varphi &&= \mathit{zero}\cdot\varphi\ \nabla\ \mathit{succ}\cdot \mathit{GFOLD}\langle \mathit{Perfect}\rangle\ (\mathit{KFOLD}\langle \mathit{Fork}\rangle\ \varphi) \\ &\mathit{KFOLD}\langle \mathit{Fork}\rangle &&:: (g\cdot x \stackrel{.}{\to} x\cdot\bar{g}) \to (\mathit{Fork}\cdot g\cdot x \stackrel{.}{\to} x\cdot \mathit{fork}\cdot\bar{g}) \\ &\mathit{KFOLD}\langle \mathit{Fork}\rangle\ \varphi &&= \mathit{fork}\cdot(\varphi\times\varphi)\cdot \mathit{unFork} \end{aligned}$$

The corresponding Haskell definition is listed in Figure 2. The definition of *gFOLDP* is interesting in that it makes use of polymorphic recursion on the *level of type constructors*, that is, it combines polymorphic recursion and type constructor polymorphism. The recursive call has type

$$(\mathit{Fork}\cdot g\cdot x \stackrel{.}{\to} x\cdot \mathit{fork}\cdot\bar{g}) \to (\mathit{Perfect}\cdot \mathit{Fork}\cdot g\cdot x \stackrel{.}{\to} \mathit{perfect}\cdot \mathit{fork}\cdot\bar{g})\ ,$$

which is a substitution instance of the declared type (instantiating g to $\mathit{Fork}\cdot g$ and $\bar{g}$ to $\mathit{fork}\cdot\bar{g}$). Previous uses of type constructor polymorphism could be circumvented in Haskell by defining type specialized versions of the polymorphic function. For *gFOLDP* this trick is not applicable. As matters stand higher-order folds for nested types cannot be typed in Haskell. This problem could be solved, for instance, by extending Haskell with syntax for explicit type abstractions and type applications.

$$
\begin{array}{lcl}
\mathit{gFOLDP} & :: & (\forall a.x\ a \rightarrow \mathit{perfect}\ a) \\
 & \rightarrow & (\forall a.\mathit{perfect}\ (\mathit{fork}\ a) \rightarrow \mathit{perfect}\ a) \\
 & \rightarrow & (\forall a.x\ a \rightarrow x\ a \rightarrow x\ (\mathit{fork}\ a)) \\
 & \rightarrow & (\forall a.g\ (x\ a) \rightarrow x\ (\bar{g}\ a)) \\
 & \rightarrow & (\forall a.\mathit{Perfect}\ (g\ (x\ a)) \rightarrow \mathit{perfect}\ (\bar{g}\ a)) \\
\mathit{gFOLDP}\ z\ s\ f\ \varphi\ (\mathit{Zero}\ a) & = & (z \cdot \varphi)\ a \\
\mathit{gFOLDP}\ z\ s\ f\ \varphi\ (\mathit{Succ}\ a) & = & (s \cdot \mathit{gFOLDP}\ z\ s\ f\ (\mathit{kFOLDF}\ f\ \varphi))\ a \\
\mathit{kFOLDF} & :: & (\forall a.x\ a \rightarrow x\ a \rightarrow x\ (\mathit{fork}\ a)) \\
 & \rightarrow & (\forall a.g\ (x\ a) \rightarrow x\ (\bar{g}\ a)) \\
 & \rightarrow & (\forall a.\mathit{Node}\ (g\ (x\ a)) \rightarrow x\ (\mathit{fork}\ (\bar{g}\ a))) \\
\mathit{kFOLDF}\ f\ \varphi\ (\mathit{Fork}\ a_1\ a_2) & = & f\ (\varphi\ a_1)\ (\varphi\ a_2)
\end{array}
$$

Fig. 2. Generalized higher-order fold for *Perfect*.

5 Related work

The fold operator for list, usually called *foldr*, has its origins in the reduction operator of APL [9] and in the insertion operator of FP [1]. An excellent tutorial on *foldr* giving further references is G. Hutton's paper [8]. By modeling datatypes as fixpoints of functors folds were generalized from lists to regular datatypes [15]. Folds for nested datatypes (which we call simple folds) first appeared in [2] where a functorial semantics for nests is presented. It was noted that these folds are limited in applicability but no solution to this problem was given. R. Bird and R. Paterson [4] subsequently introduced generalized folds in order to overcome this limitation. The differences to our approach are best explained by means of an example. Consider R. Bird and R. Paterson's fold for perfect trees.

$$
\begin{array}{lcl}
\mathit{gfoldP} & :: & (\forall a.x\ a \rightarrow \mathit{perfect}\ a) \\
 & \rightarrow & (\forall a.\mathit{perfect}\ (\mathit{Fork}\ a) \rightarrow \mathit{perfect}\ a) \\
 & \rightarrow & (\forall a.\mathit{Fork}\ (x\ a) \rightarrow x\ (\mathit{Fork}\ a)) \\
 & \rightarrow & (\forall a.\mathit{Perfect}\ (x\ a) \rightarrow \mathit{perfect}\ a) \\
\mathit{gfoldP}\ \mathit{zero}\ \mathit{succ}\ k\ (\mathit{Zero}\ a) & = & \mathit{zero}\ a \\
\mathit{gfoldP}\ \mathit{zero}\ \mathit{succ}\ k\ (\mathit{Succ}\ a) & = & (\mathit{succ} \cdot \mathit{gfoldP}\ \mathit{zero}\ \mathit{succ}\ k \cdot \mathit{mapP}\ k)\ a
\end{array}
$$

The parameter k corresponds roughly to the subsidiary fold *kfoldF fork* in the definition in Section 1. The type of k is, however, more restricted: it has type $\mathit{Fork} \cdot x \dot{\rightarrow} x \cdot \mathit{Fork}$ as opposed to $\mathit{Fork} \cdot x \dot{\rightarrow} x \cdot \mathit{fork}$ for *kfoldF fork*. In other words, a generalized fold in the sense of R. Bird and R. Paterson replaces only the constructors listed in the datatype definition. The user has to supply additional arguments that transport the functor x to the recursive calls. To determine the number and the types of these arguments the functor equations must be rewritten into a special fixpoint form. We feel that our definition of $\mathit{gfold}\langle F\rangle$, which is inductive on the structure of datatype definitions, is easier to grasp. Moreover, our approach is amenable to the improvements described in Section 4. R. Bird's and R. Paterson's folds suffer from the same problem; whether they can be improved in a similar fashion is, however, not clear. Nonetheless, it should be stressed that [4] was our main source of inspiration. A different approach to generalized folds has recently been proposed by M. Jones and P. Blampied [10]. In essence, their folds replace a single constructor by a family of functions. For instance, the constructor $Succ :: \forall a. Perfect\ (Fork\ a) \to Perfect\ a$ is replaced by functions of the family $succ_i :: Perfect\ (Fork\ (Fork^i\ a)) \to Perfect\ (Fork^i\ a)$. It remains to be seen whether this additional flexibility is of practical importance.

Acknowledgement

I would like to thank Patrik Jansson and three anonymous referees for their numerous and valuable comments.

References

1. John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8), August 1978.
2. 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.
3. Richard Bird and Ross Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, January 1999.
4. Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.
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. *Discrete Mathematics and Theoretical Computer Science*, 3(4):159–180, September 1999.
7. Ralf Hinze. Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 2000. To appear.
8. Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.
9. Kenneth E. Iverson. *A programming Language*. John Wiley & Sons, 1962.
10. Mark B Jones and Paul Blampied. A pragmatic approach to maps and folds for parameterized datatypes. unpublished, 1999.
11. 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.
12. M.P. Jones and J.C. Peterson. *Hugs 98 User Manual*, May 1999. Available from `http://www.haskell.org/hugs`.
13. Daniel Leivant. Polymorphic type inference. In *Proc. 10th Symposium on Principles of Programming Languages*, 1983.
14. 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.

15. 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.
16. 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.
17. Simon Peyton Jones and John Hughes, editors. *Haskell 98 — A Non-strict, Purely Functional Language*, February 1999.
Available from `http://www.haskell.org/definition/`.
18. 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.
19. The GHC Team. *The Glasgow Haskell Compiler User's Guide, Version 4.04*, September 1999.
Available from `http://www.haskell.org/ghc/documentation.html`.