EFFICIENT GENERALIZED FOLDS

RALF HINZE

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

July, 2000

(Pick the slides at `.../~ralf/talks.html#T13`.)

Efficient generalized folds for *nested data types* in 4 steps:

1. Simple folds: replacing constructors by functions.

2. Folds: replacing *all* constructors by functions.

3. Generalized folds: folds with a more general type.

4. Efficient generalized folds: eliminating map.

A nested data type

Okasaki's binary random-access lists:

$$\textbf{data}\ \mathit{Sequ}\ a = \mathit{End} \mid \mathit{Zero}\ (\mathit{Sequ}\ (\mathit{Tree}\ a)) \mid \mathit{One}\ a\ (\mathit{Sequ}\ (\mathit{Tree}\ a))$$
$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Node}\ a\ a.$$

The sequence $a\ b\ c\ d\ e$, for instance, is represented by

$$\mathit{One}\ a\ (\mathit{Zero}\ (\mathit{One}\ (\mathit{Node}\ (\mathit{Node}\ b\ c)\ (\mathit{Node}\ d\ e))\ \mathit{End})).$$

NB. A binary random-access list is a sequence of perfect binary leaf trees stored in increasing order of height.

Step 1

Since $Sequ$ and $Tree$ are parameterized types, the value constructors have polymorphic types:

$$\begin{array}{ll} End & :: \forall a. Sequ\ a \\ Zero & :: \forall a. Sequ\ (Tree\ a) \to Sequ\ a \\ One & :: \forall a. a \to Sequ\ (Tree\ a) \to Sequ\ a \\ Node & :: \forall a. a \to a \to Tree\ a. \end{array}$$

Replacing $Seq u$ by a type variable, say, $sequ$ yields the algebra of $Sequ$:

$$
\begin{array}{l}
\mathbf{data}\ AlgebraS\ sequ \\
\quad = \ AlgS\{ \begin{array}[t]{lll} end & :: & \forall a. sequ\ a, \\ zero & :: & \forall a. sequ\ (Tree\ a) \rightarrow sequ\ a, \\ one & :: & \forall a. a \rightarrow sequ\ (Tree\ a) \rightarrow sequ\ a\}. \end{array}
\end{array}
$$

The simple fold is then given by

$$
\begin{array}{ll}
sfoldS & :: AlgebraS\ sequ \rightarrow (Sequ\ a \rightarrow sequ\ a) \\
sfoldS\ \alpha\ End & = \alpha.end \\
sfoldS\ \alpha\ (Zero\ s) & = \alpha.zero\ (sfoldS\ \alpha\ s) \\
sfoldS\ \alpha\ (One\ a\ s) & = \alpha.one\ a\ (sfoldS\ \alpha\ s).
\end{array}
$$

For instance, $sfoldS\ \alpha$ maps

$$One\ a\ (Zero\ (One\ (Node\ (Node\ b\ c)\ (Node\ d\ e))\ End))$$

to

$$\alpha.one\ a\ (\alpha.zero\ (\alpha.one\ (Node\ (Node\ b\ c)\ (Node\ d\ e))\ \alpha.end)).$$

Using $sfoldS$ we can convert a random-access list to a list ($sequ\ a = [\,a\,]$):

$$\begin{aligned}
&toList \;::\; Sequ\ a \to [\,a\,]\\
&toList \;=\; sfoldS\ \sigma\\
&\quad \mathbf{where}\ \sigma = AlgS\{\, end \;= [\,],\\
&\qquad\qquad\qquad\quad zero = \lambda ts \to untrees\ ts,\\
&\qquad\qquad\qquad\quad one \;= \lambda a\ ts \to a : untrees\ ts\,\}
\end{aligned}$$

where

$$\begin{aligned}
&untrees && :: [\,Tree\ a\,] \to [\,a\,]\\
&untrees\ [\,] && = [\,]\\
&untrees\ (Node\ a_1\ a_2 : ts) && = a_1 : a_2 : untrees\ ts.
\end{aligned}$$

NB. $toList$ implements an iterative algorithm: a list of trees is repeatedly split into subtrees until only singleton trees, that is, elements remain.

$$
\begin{aligned}
&\alpha.one\ a\ (\alpha.zero\ (\alpha.one\ (Node\ (Node\ b\ c)\ (Node\ d\ e))\ \alpha.end))\\
&\Longrightarrow \alpha.one\ a\ (\alpha.zero\ (\alpha.one\ (Node\ (Node\ b\ c)\ (Node\ d\ e))\ [\,]))\\
&\Longrightarrow \alpha.one\ a\ (\alpha.zero\ (Node\ (Node\ b\ c)\ (Node\ d\ e) : [\,]))\\
&\Longrightarrow \alpha.one\ a\ (Node\ b\ c : Node\ d\ e : [\,])\\
&\Longrightarrow a : b : c : d : e : [\,]
\end{aligned}
$$

Since $sfoldS$ replaces End, $Zero$ and One ***but not*** $Node$ we cannot implement the usual recursive method that works by flattening the subtrees and appending the results.

This observation suggests to provide a fold that maps

$$One\ a\ (Zero\ (One\ (Node\ (Node\ b\ c)\ (Node\ d\ e))\ End))$$

to

$$\alpha.one\ a\ (\alpha.zero\ (\alpha.one\ (\alpha.node\ (\alpha.node\ b\ c)\ (\alpha.node\ d\ e))\ \alpha.end)).$$

Step 2

The algebra of $Sequ$ now contains the algebra of $Tree$ as a 'subalgebra':

$$
\begin{array}{l}
\textbf{data}\ AlgebraT\ tree \\
\quad = \ AlgT\{ node :: \forall a . a \to a \to tree\ a \} \\
\textbf{data}\ AlgebraS\ sequ\ tree \\
\quad = \ AlgS\{ \begin{array}[t]{ll} algT & :: AlgebraT\ tree, \\ end & :: \forall a . sequ\ a, \\ zero & :: \forall a . sequ\ (tree\ a) \to sequ\ a, \\ one & :: \forall a . a \to sequ\ (tree\ a) \to sequ\ a \}. \end{array}
\end{array}
$$

To define the new fold we require the so-called *mapping function* for binary random-access lists.

$$
\begin{aligned}
&mapS && :: (a \to b) \to (Sequ\ a \to Sequ\ b)\\
&mapS\ \varphi\ End && = End\\
&mapS\ \varphi\ (Zero\ s) && = Zero\ (mapS\ (mapT\ \varphi)\ s)\\
&mapS\ \varphi\ (One\ a\ s) && = One\ (\varphi\ a)\ (mapS\ (mapT\ \varphi)\ s)
\end{aligned}
$$

$$
\begin{aligned}
&mapT && :: (a \to b) \to (Tree\ a \to Tree\ b)\\
&mapT\ \varphi\ (Node\ a_1\ a_2) && = Node\ (\varphi\ a_1)\ (\varphi\ a_2)
\end{aligned}
$$

The fold is now given by

$$
\begin{array}{ll}
\mathit{foldS} & :: \mathit{AlgebraS}\ \mathit{sequ}\ \mathit{tree} \to (\mathit{Sequ}\ a \to \mathit{sequ}\ a) \\
\mathit{foldS}\ \alpha\ \mathit{End} & = \alpha.\mathit{end} \\
\mathit{foldS}\ \alpha\ (\mathit{Zero}\ s) & = \alpha.\mathit{zero}\ (\mathit{foldS}\ \alpha\ (\mathit{mapS}\ (\mathit{foldT}\ \alpha.\mathit{algT})\ s)) \\
\mathit{foldS}\ \alpha\ (\mathit{One}\ a\ s) & = \alpha.\mathit{one}\ a\ (\mathit{foldS}\ \alpha\ (\mathit{mapS}\ (\mathit{foldT}\ \alpha.\mathit{algT})\ s))
\end{array}
$$

$$
\begin{array}{ll}
\mathit{foldT} & :: \mathit{AlgebraT}\ \mathit{tree} \to (\mathit{Tree}\ a \to \mathit{tree}\ a) \\
\mathit{foldT}\ \alpha\ (\mathit{Node}\ a_1\ a_2) & = \alpha.\mathit{node}\ a_1\ a_2.
\end{array}
$$

Alas, we still cannot 'recursively flatten' a random-access list.

Why? The only sensible choices for the type variables sequ and tree are $\mathit{sequ}\ a = [\,a\,]$ and $\mathit{tree}\ a = [\,a\,]$.

$$
\begin{array}{lll}
\mathit{toList}' & :: & \mathit{Sequ}\ a \to [\,a\,] \\
\mathit{toList}' & = & \mathit{foldS}\ \sigma \\
\multicolumn{3}{l}{\quad \textbf{where}\ \sigma = \mathit{AlgS}\{\,\begin{array}[t]{@{}l@{\ }l}\mathit{algT} & = \tau, \\ \mathit{end} & = [\,], \\ \mathit{zero} & = \lambda \mathit{xs} \to \mathit{concat}\ \mathit{xs}, \\ \mathit{one} & = \lambda a\ \mathit{xs} \to a : \mathit{concat}\ \mathit{xs}\,\}\end{array}} \\
\multicolumn{3}{l}{\qquad\qquad\quad \tau = \mathit{AlgT}\{\,\mathit{node} = \lambda a_1\ a_2 \to [\,a_1, a_2\,]\,\}}
\end{array}
$$

NB. This is the iterative solution in disguise.

Step 3

The central idea, due to Bird and Paterson, is to define a generalized fold

$$gfoldS\ \alpha :: Sequ\ (x\ a) \rightarrow sequ\ a.$$

How do we fold $Zero\ s :: (Sequ \cdot x)\ a$ where $s :: (Sequ \cdot Tree \cdot x)\ a$?

$$\begin{array}{ll}
Sequ \cdot Tree & Sequ \cdot Tree \cdot x \\
\quad \downarrow mapS\ (foldT\ \alpha.algT) & \quad \downarrow mapS\ (kfoldT\ \alpha.algT) \\
Sequ \cdot tree & Sequ \cdot x \cdot tree \\
\quad \downarrow foldS\ \alpha & \quad \downarrow gfoldS\ \alpha \\
sequ \cdot tree & sequ \cdot tree \\
\quad \downarrow \alpha.zero & \quad \downarrow \alpha.zero \\
sequ & sequ
\end{array}$$

Now we have two sorts of algebras:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{KAlgebraT}\ x\ \mathit{tree} \\
\ \ =\ \ \mathit{KAlgT}\{\mathit{node} :: \forall a.x\ a \to x\ a \to x\ (\mathit{tree}\ a)\} \\
\mathbf{data}\ \mathit{GAlgebraS}\ x\ \mathit{sequ}\ \mathit{tree} \\
\ \ =\ \ \mathit{GAlgS}\{ \begin{array}[t]{ll} \mathit{algT} & :: \mathit{KAlgebraT}\ x\ \mathit{tree}, \\ \mathit{end} & :: \forall a.\mathit{sequ}\ a, \\ \mathit{zero} & :: \forall a.\mathit{sequ}\ (\mathit{tree}\ a) \to \mathit{sequ}\ a, \\ \mathit{one} & :: \forall a.x\ a \to \mathit{sequ}\ (\mathit{tree}\ a) \to \mathit{sequ}\ a\}. \end{array}
\end{array}
$$

In general, K-algebras are for folds of type $\mathit{kfoldF}\ \alpha :: F\ (x\ a) \to x\ (f\ a)$ and G-algebras are for folds of type $\mathit{gfoldF}\ \alpha :: F\ (x\ a) \to f\ a$.

However, the code is essentially the same:

$$
\begin{aligned}
&gfoldS && :: GAlgebraS\ x\ sequ\ tree \\
& && \quad \rightarrow (Sequ\ (x\ a) \rightarrow sequ\ a) \\
&gfoldS\ \alpha\ End && = \alpha.end \\
&gfoldS\ \alpha\ (Zero\ s) && = \alpha.zero\ (gfoldS\ \alpha\ (mapS\ (kfoldT\ \alpha.algT)\ s)) \\
&gfoldS\ \alpha\ (One\ a\ s) && = \alpha.one\ a\ (gfoldS\ \alpha\ (mapS\ (kfoldT\ \alpha.algT)\ s))
\end{aligned}
$$

$$
\begin{aligned}
&kfoldT && :: KAlgebraT\ x\ tree \\
& && \quad \rightarrow (Tree\ (x\ a) \rightarrow x\ (tree\ a)) \\
&kfoldT\ \alpha\ (Node\ a_1\ a_2) && = \alpha.node\ a_1\ a_2.
\end{aligned}
$$

Setting $x\ a = [a]$, $sequ\ a = [a]$ and $tree\ a = a$ we can finally 'recursively flatten' a random-access list:

$$
\begin{array}{lcl}
toList'' & :: & Sequ\ a \to [a] \\
toList'' & = & gfoldS\ \sigma \cdot mapS\ (\lambda a \to [a]) \\
\end{array}
$$

$$
\begin{array}{rl}
\textbf{where}\ \sigma = GAlgS\{ & algT = \tau, \\
 & end \ \ = [\,], \\
 & zero = id, \\
 & one \ \ = (+\!\!+)\} \\
\tau = KAlgT\{ & node = (+\!\!+)\}.
\end{array}
$$

NB. Haskell cannot infer $tree\ a = a$. We must use $tree\ a = Id\ a$ where

$$\mathbf{data}\ Id\ a = Id\{\,unId :: a\,\}$$

and we must add explicit coercions:

$$\begin{array}{lcl} toList'' & :: & Sequ\ a \to [\,a\,] \\ toList'' & = & gfoldS\ \sigma \cdot mapS\ (\lambda a \to [\,a\,]) \\ \multicolumn{3}{l}{\quad \mathbf{where}\ \sigma = GAlgS\{\,algT = \tau,} \\ \multicolumn{3}{l}{\qquad\qquad\qquad\qquad\quad end = [\,],} \\ \multicolumn{3}{l}{\qquad\qquad\qquad\qquad\quad zero = \lambda xs \to map\ unId\ xs,} \\ \multicolumn{3}{l}{\qquad\qquad\qquad\qquad\quad one = \lambda xs_1\ xs_2 \to xs_1 +\!\!+\ map\ unId\ xs_2\}} \\ \multicolumn{3}{l}{\qquad\quad \tau = KAlgT\{\,node = \lambda xs_1\ xs_2 \to map\ Id\ (xs_1 +\!\!+\ xs_2)\,\}.} \end{array}$$

Step 4

Alas, generalized folds are inefficient: for some data types we experience a quadratic running time for linear time algorithms.

This is because the mapping functions take linear time.

$$\boxed{gfold\langle F_1 \cdot F_2\rangle = gfold\langle F_1\rangle \cdot map\langle F_1\rangle\ (gfold\langle F_2\rangle)}$$

We can eliminate the call to map by defining a higher-order fold that combines a fold with a map:

$$\boxed{GFold\langle F\rangle\ \varphi = gfold\langle F\rangle \cdot map\langle F\rangle\ \varphi}$$

so that $GFold\langle F_1\rangle\ (kfold\langle F_2\rangle) = gfold\langle F_1 \cdot F_2\rangle$.

We obtain the following efficient generalized fold:

$$
\begin{array}{ll}
gFoldS & :: GAlgebraS\ x\ sequ\ tree \\
& \quad \to (\forall a.g\ (x\ a) \to x\ (\bar{g}\ a)) \\
& \quad \to (\forall a.Sequ\ (g\ (x\ a)) \to sequ\ (\bar{g}\ a)) \\
gFoldS\ \alpha\ \varphi\ End & = \alpha.end \\
gFoldS\ \alpha\ \varphi\ (Zero\ s) & = \alpha.zero\ (gFoldS\ \alpha\ (kFoldT\ \alpha.algT\ \varphi)\ s) \\
gFoldS\ \alpha\ \varphi\ (One\ a\ s) & = \alpha.one\ (\varphi\ a)\ (gFoldS\ \alpha\ (kFoldT\ \alpha.algT\ \varphi)\ s)
\end{array}
$$

$$
\begin{array}{ll}
kFoldT & :: KAlgebraT\ x\ tree \\
& \quad \to (\forall a.g\ (x\ a) \to x\ (\bar{g}\ a)) \\
& \quad \to (\forall a.Tree\ (g\ (x\ a)) \to x\ (tree\ (\bar{g}\ a))) \\
kFoldT\ \alpha\ \varphi\ (Node\ a_1\ a_2) & = \alpha.node\ (\varphi\ a_1)\ (\varphi\ a_2).
\end{array}
$$

NB. Haskell cannot type check the definition of $gFoldS$ as it supports only a limited form of type constructor polymorphism.

Worse still, since $gFoldS$ makes use of polymorphic recursion on the *level of type constructors*, we cannot even define a type specialized version.

Summary

- Generalized folds generalize folds to nested data types. They can be defined by induction on the structure of data type declarations.

- Efficient generalized folds take time proportional to the size of the input (plus the time to evaluate the function applications).

- Unfortunately, Haskell's kinded first-order unification is a hindrance to implementing these folds.

Appendix: the generalized folds of Bird and Paterson

$$
\begin{array}{l}
\textbf{data}\ \mathit{AlgebraS}\ x\ \mathit{sequ} \\
\quad = \ \mathit{AlgS}\{\begin{array}[t]{ll}
k & :: \forall a.\mathit{Tree}\ (x\ a) \rightarrow x\ (\mathit{Tree}\ a), \\
\mathit{end} & :: \forall a.\mathit{sequ}\ a, \\
\mathit{zero} & :: \forall a.\mathit{sequ}\ (\mathit{Tree}\ a) \rightarrow \mathit{sequ}\ a, \\
\mathit{one} & :: \forall a.x\ a \rightarrow \mathit{sequ}\ (\mathit{Tree}\ a) \rightarrow \mathit{sequ}\ a\}.
\end{array}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{gfoldS} & :: \mathit{AlgebraS}\ x\ \mathit{sequ} \rightarrow (\mathit{Sequ}\ (x\ a) \rightarrow \mathit{sequ}\ a) \\
\mathit{gfoldS}\ \alpha\ \mathit{End} & = \alpha.\mathit{end} \\
\mathit{gfoldS}\ \alpha\ (\mathit{Zero}\ s) & = \alpha.\mathit{zero}\ (\mathit{gfoldS}\ \alpha\ (\mathit{mapS}\ \alpha.k\ s)) \\
\mathit{gfoldS}\ \alpha\ (\mathit{One}\ a\ s) & = \alpha.\mathit{one}\ a\ (\mathit{gfoldS}\ \alpha\ (\mathit{mapS}\ \alpha.k\ s))
\end{array}
$$