The Under-Appreciated Unfold

Jeremy Gibbons
School of Computing and Math. Sciences
Oxford Brookes University
Gipsy Lane, Headington,
Oxford OX3 0BP, UK.
Email: jgibbons@brookes.ac.uk

Geraint Jones
Oxford University Computing Lab
Wolfson Building, Parks Road
Oxford OX1 3QD, UK.
Email: geraint@comlab.ox.ac.uk

Abstract

Folds are appreciated by functional programmers; the benefits of encapsulating common patterns of computation as higher-order operators are well-known and well understood. Their dual, *unfolds*, are nearly as well-known, but not nearly as well appreciated. We believe they deserve better. To illustrate, we present (indeed, we calculate) a number of algorithms for computing the *breadth-first traversal* of a tree. We specify breadth-first traversal in terms of *level-order traversal*, which we characterize first as a fold. The presentation as a fold is simple, but it is inefficient, and removing the inefficiency makes it no longer a fold. We calculate a characterization as an unfold from the characterization as a fold; this unfold is equally clear, but more efficient. We also calculate a characterization of breadth-first traversal directly as an unfold; this turns out to be the 'standard' queue-based algorithm.

Keywords: Program calculation, functional programming, fold, unfold, anamorphism, co-induction, traversal, breadth-first, level-order.

1 Introduction

Folds are appreciated by functional programmers; the benefits of encapsulating common patterns of computation as higher-order operators are well-known and well understood. Their dual, *unfolds*, have been explored by Hagino [8] and Malcolm [12], and popularized at this conference by Meijer *et al* [14]. Unfolds are now nearly as well-known as folds, but they are not nearly as well appreciated. (For example, they merit just half a page in [4], and have disappeared altogether in [3]. Co-inductive types warrant a few pages in [18], but apart from that there are no other mentions in the nine functional programming textbooks on our shelves.) We believe unfolds deserve a much higher profile.

To illustrate this claim, we present (indeed, we *calculate*) a number of algorithms for computing the *breadth-first traversal* of a tree in a pure functional language. This is a thorny problem for functional programmers, in contrast to the more natural depth-first traversal: depth-first traversal runs with the grain of the tree, but breadth-first traversal runs against the grain. Nevertheless, we can construct a simple and elegant characterization of breadth-first traversal in terms of *level-order traversal*, which we specify with a fold. Unfortunately, this characterization is inefficient, and in order to remove the inefficiency we must resort to a 'mere' recursive definition (or to a higher-order fold).

In contrast, from the fold characterization of level-order traversal we can calculate an unfold characterization. The unfold characterization is equally clear, but apparently less obvious, even to experienced functional programmers. (We have been talking about this topic to various audiences for five years, but have only recently discovered the unfold characterization.) Moreover, the unfold characterization is efficient, taking linear time. Best of all, it is easier to manipulate; in particular, it leads easily to a *deforested* program with no unnecessary data structures.

Taking a different route, we can also calculate a characterization of breadth-first traversal directly as an unfold. This turns out to be the 'standard' queue-based algorithm, which, with a little extra work to make the queue operations efficient, also takes linear time.

The remainder of this paper is structured as follows. In Section 2, we briefly present our notation. In Section 3, we define breadth-first traversal, in terms of level-order traversal. In Section 4, we present the characterization of level-order traversal as a fold, and show why it is inefficient; we then calculate the efficient characterization that ceases to be a (first-order) fold. In Section 5, we calculate the characterization of level-order traversal as an unfold from the characterization as a fold, and show why it is linear. Finally, in Section 6 we calculate as an unfold the standard queue-based algorithm for breadth-first traversal.

2 Notation

We will be using Haskell notation [17], but the translation into nearly any modern functional language is straightforward. (Not even laziness is required; all our unfolds will in fact generate finite results, given finite arguments.)

2.1 Folds over lists

We will be using two kinds of *fold* on lists: the normal 'fold right',

```
foldr op e []     = e
foldr op e (a:x) = a `op` foldr op e x
```

Submitted to the Third ACM SIGPLAN International Conference on Functional Programming, Baltimore, Maryland, September 1998.

(here, the function `op` is converted into a binary operator `` `op` `` by writing it in backquotes) and a restricted version for non-empty lists, determined by

```
foldr1 op (x++[a]) = foldr op a x
```

The two are related by the property that, for non-empty `x`,

```
foldr op e x = foldr1 op x
```

when `e` is a right unit of `op`.

The normal fold enjoys the following *universal property*, essentially saying that the definition of `foldr`, treated as an equation in the 'unknown' `foldr op e`, has a unique (strict) solution. In other words, for strict `h`,

```
h = foldr op e  ≡  h [] = e   ∧
                   h (a:x) = a `op` h x
```

A number of *promotion properties* are simple consequences of the universal property:

fold-map promotion: if `f` is strict and

```
f (a `op` b) = f a `op2` f b
```

then

```
f . foldr op e = foldr op2 (f e) . map f
f . foldr1 op  = foldr1 op2 . map f
```

fold-join promotion: if

```
h = foldr op e . map f
```

where `op` is associative with identity `e`, then

```
h (xs ++ ys) = h xs `op` h ys
```

fold-concat promotion: if

```
h = foldr op e . map f
```

where `op` is associative with identity `e`, then

```
h . concat = foldr op e . map h
```

2.2 Unfolds over lists

We also use *unfolds* [8, 12, 14], a dual to folds. The standard construction of unfolds [12] gives the characterization

```
unfold :: (b -> Either () (a,b)) -> b -> [a]
unfold pfg x = case pfg x of
  Left ()     -> []
  Right (a,y) -> a : unfold pfg y
```

but we will find it more convenient to use the equivalent characterization

```
unfold :: (b->Bool) -> (b->a) -> (b->b) -> b -> [a]
unfold p f g x
  | p x       = []
  | otherwise = f x : unfold p f g (g x)
```

Unfolds too enjoy a *universal property*, saying that the above definition, considered as an equation in the unknown `unfold p f g`, has a unique solution:

```
h = unfold p f g
```

precisely when

- `h x = []` iff `p x` holds, and
- for `x` such that `p x` does not hold (so `h x` is not empty),

  ```
  head (h x) = f x
  tail (h x) = h (g x)
  ```

We work in the setting of continuous functions between pointed complete partial orders, as advocated by Meijer *et al* [14], instead of the setting of total functions between sets originally used by Hagino [8] and Malcolm [12], in order better to match the semantics of most functional programming languages. In particular, because of the treatment of infinite data structures, with sets and total functions, the data structures generated by unfolds are different from the data structures consumed by folds, so folds and unfolds cannot be composed; with cpos, the two kinds of data structure are the same.

2.3 Trees

Our trees are represented by the datatype

```
data Tree a = Nd a [Tree a]
```

of *rose trees* [13]. That is, a tree of type `Tree a` consists of a root label of type `a` and a list of children, each again of type `Tree a`. We define the two deconstructors `root` and `kids`:

```
root :: Tree a -> a
root (Nd a ts) = a

kids :: Tree a -> [Tree a]
kids (Nd a ts) = ts
```

Actually, we carry out most calculations on *forests*, lists of trees; it turns out simpler that way. We use `Forest a` as a synonym for the type `[Tree a]`.

2.4 Folds over trees

The datatypes of trees and forests are mutually recursive, so folds over trees and forests are too. We define the two folds as follows:

```
foldt :: (a->b->b) -> ([b]->b) -> Tree a -> b
foldt f g (Nd a ts) = f a (foldf f g ts)

foldf :: (a->b->b) -> ([b]->b) -> Forest a -> b
foldf f g ts = g (map (foldt f g) ts)
```

For example, the function `sumt`, which sums a tree of numbers, is given by

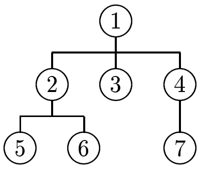
```
sumt = foldt (+) sum
```

(where `sum` sums a list of numbers), and the corresponding function `sumf` on forests by

```
sumf = foldf (+) sum
```

3 Breadth-first traversal

A *tree traversal* is an operation that, given a tree, computes a list consisting of all the elements of the tree in some order. One example of a tree traversal is *preorder traversal*, in which every parent appears in the traversal before any of its children, and siblings appear in left-to-right order. For example, the preorder traversal of the tree



is the list `[1,2,5,6,3,4,7]`.

Preorder traversal, *postorder traversal* (in which a parent appears *after* all its children) and *inorder traversal* (which only makes sense on binary trees, and in which a parent appears *between* its two children) are all examples of *depth-first* traversals. They are easy to implement in a pure functional language, because they naturally follow the structure of the tree—that is, they can be expressed as *folds* over trees. For example, preorder traversal of a tree is given by

```
preordert = foldt (:) concat
```

In contrast, *breadth-first traversal* goes against the structure of the tree. The breadth-first traversal of a tree consists first of the root (the only element at depth 1), then of all the elements at depth 2, and so on. For example, the breadth-first traversal of the tree above is `[1,2,3,4,5,6,7]`. It is not nearly so obvious how to implement breadth-first traversal efficiently in a pure functional language. In particular, breadth-first traversal is not a fold, because the traversal of a forest cannot be constructed from the traversals of the trees in that forest. The standard implementation of breadth-first traversal in an imperative language involves queues, which are awkward to express functionally because they require fast access to both ends of a list. In contrast, depth-first traversals are based on stacks, which 'come for free' with recursive programs.

It *is* possible to express the standard queue-based algorithm efficiently in a pure functional language; indeed, we do so in Section 6. However, this algorithm is unsatisfactory in a functional language, for two reasons. For one thing, some effort is required to implement queues with constant-time operations, which is necessary to get a linear-time program. For another, the queue-based algorithm really describes a process rather than a value, and so is rather low-level; in a declarative language, we would prefer a more declarative 'specification' of the problem (even if we then choose a more operational implementation).

We find this more declarative characterization of breadth-first traversal in the notion of *level-order traversal* [6, 7] of a tree. This gives the elements on each level of the tree, as a list of lists (although, strictly speaking, this is not our definition a *traversal*). For example, the level-order traversal of the tree above is the list of lists

```
[ [1], [2,3,4], [5,6,7] ]
```

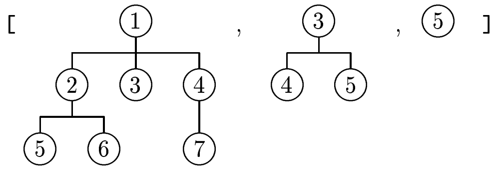
Given the level-order traversal, the breadth-first traversal is easy to construct: simply concatenate the levels.

4 Traversal as a fold

In this section, we will present a characterization of level-order traversal as a fold over forests. That is, we will define the two related functions

```
levelf :: Forest a -> [[a]]
levelt :: Tree a -> [[a]]
```

to compute the level-order traversals of forests and trees, respectively. For example, the levelorder traversal of the forest



is the list of lists

```
[ [1,3,5], [2,3,4,4,5], [5,6,7] ]
```

Given the levelorder traversal of a forest, the breadth-first traversal is formed by concatenating the levels:

```
bftf :: Forest a -> [a]
bftf =  concat . levelf

bftt :: Tree a -> [a]
bftt =  concat . levelt
```

Now, the level-order traversal of a forest is found by 'gluing together' the level-order traversals of the trees in that forest. Two lists of lists can be 'glued together' in the appropriate way by the function `lzc` (standing for 'long zip with concatenate'). 'Long zip with' is related to the standard function `zipWith`, but it returns a list as long as its *longer* argument, whereas `zipWith` returns a list as long as its shorter argument. Formally, we have

```
lzc :: [[a]] -> [[a]] -> [[a]]
lzc = lzw (++)
```

where

```
lzw :: (a->a->a) -> [a] -> [a] -> [a]
lzw op xs ys
  | null xs   = ys
  | null ys   = xs
  | otherwise = (head xs `op` head ys) :
                  lzw op (tail xs) (tail ys)
```

(Note that `lzw op` is associative when `op` is.)

Now we can define the two functions `levelf` and `levelt`, returning the level-order traversal of a forest and a tree respectively, by

```
levelf = foldf f g
levelt = foldt f g
  where f a xss = [a] : xss
        g = foldr lzc []
```

(In passing, we observe that `lzw' f = uncurry (lzw f)` is an unfold:

```
lzw' op = unfold p f g where
  p (xs,ys) = null xs && null ys
  f (xs,ys)
    | null ys   = head xs
    | null xs   = head ys
    | otherwise = head xs `op` head ys
  g (xs,ys) = (tail' xs, tail' ys)
  tail' zs
    | null zs   = []
    | otherwise = tail zs
```

However, this uncurried version is inconvenient to use, because the standard definition of `foldr` requires a curried operator. Moreover, this version is less efficient than the direct recursion, because it takes time proportional to the length of the result, whereas the direct recursion only traverses the shorter argument.)

4.1 Traversal as a fold in linear time

This characterization of `levelf` (and hence of `bftf`) does not take linear time, even using the efficient long zip `lzw`. Consider for example the forest

```
ts = [ Nd 1 [t,u], Nd 2 [v,w] ]
```

where `t`, `u`, `v` and `w` are four trees. Unfolding the definitions, we have

```
levelf ts
  = lzc ([1] : lzc (levelt t)
                   (lzc (levelt u) []))
        (lzc ([2] : lzc (levelt v)
                        (lzc (levelt w) [])))
             []
```

Note that `levelt t` and `levelt u` must be traversed once each to compute `levelf [t,u]`, and then traversed again to compute `levelf ts`. In a complete binary tree of depth `d`, the level-order traversals of the deepest trees will be retraversed `d-1` times; the whole algorithm takes time proportional to the size of the forest times its depth.

The standard technique of introducing an *accumulating parameter* [1] can be used to remove this inefficiency. We introduce two auxilliary functions `levelf'` and `levelt'`, defined by

```
levelf' ts xss = lzc (levelf ts) xss
levelt' t xss  = lzc (levelt t) xss
```

This is a generalization, because

```
levelf ts = levelf' ts []
levelt t  = levelt' t []
```

Now, we have

```
   levelf' [] xss
=    { levelf' }
   lzc (levelf []) xss
=    { levelf, lzc }
   xss
```

and

```
   levelf' (t:ts) xss
=    { levelf' }
   lzc (levelf (t:ts)) xss
=    { levelf }
   lzc (lzc (levelt t) (levelf ts)) xss
=    { lzc is associative }
   lzc (levelt t) (lzc (levelf ts) xss)
=    { levelt', levelf' }
   levelt' t (levelf' ts xss)
```

For `levelt'`, we have to consider separately the two cases whether or not `xss` is empty. When `xss` is empty, we have

```
   levelt' (Nd a ts) xss
=    { levelt' }
   lzc (levelt (Nd a ts)) xss
=    { lzc; xss is empty }
   levelt (Nd a ts)
=    { levelt }
   [a] : levelf ts
=    { levelf'; xss is empty }
   [a] : levelf' ts xss
```

and when `xss` is non-empty, we have

```
   levelt' (Nd a ts) xss
=    { levelt' }
   lzc (levelt (Nd a ts)) xss
=    { levelt }
   lzc ([a]:levelf ts) xss
=    { lzc; xss is non-empty }
   (a:head xss) : (lzc (levelf ts) (tail xss))
=    { levelf' }
   (a:head xss) : (levelf' ts (tail xss))
```

Hence we can define

```
levelf' ts xss = foldr levelt' xss ts
levelt' (Nd a ts) xss = (a:ys) : levelf' ts yss
  where
    (ys,yss) | null xss  = ([],[])
             | otherwise = (head xss,tail xss)
```

which takes linear time. (The efficient long zip `lzw` is necessary here; with the unfold version `lzw'` the program is still quadratic in the worst case.)

Unfortunately, this efficient characterization of level-order traversal is no longer a fold: the traversal of a forest is not constructed from the independent traversals of the trees in that forest, but rather, the trees must be considered from right to left, the traversal of one being used as a starting point for constructing the 'traversal' of the next. This is sad, because we have to resort to expressing the recursion directly, losing the benefits of higher-order operators. Apart from being more difficult to read, this direct recursion is no longer suitable for parallel evaluation, because the accumulating parameter is 'single-threaded' throughout the computation.

It is possible to regain a characterization as a fold, but taking linear time, by abstracting from the accumulating parameter and constructing instead a function *between* lists of lists, in a *continuation-based* [20] or *higher-order fold* [5] style:

```
levelf'' = foldf f g
levelt'' = foldt f g
  where f a hss = (a:) : hss
        g = foldr (lzw (.)) []
```

but this is even more complicated, and moreover it requires higher-order languages features, and so cannot be used in more traditional languages.

5 Traversal as an unfold

In this section, we calculate a characterization of `levelf` as an unfold. We have to find `p`, `f` and `g` such that

```
levelf = unfold p f g
```

Since a non-empty forest has a non-empty traversal,

```
levelf ts = []  ≡  null ts
```

which determines `p`; it remains only to consider non-empty forests.

```
   head . levelf
=    { levelf }
   head . foldr lzc [] . map levelt
=    { foldr lzc [] = foldr1 lzc
       on non-empty lists }
   head . foldr1 lzc . map levelt
=    { fold-map promotion: for non-empty xs, ys,
       head (lzw f xs ys) = f (head xs) (head ys) }
   foldr1 (++) . map head . map levelt
=    { head . levelt = wrap . root, where wrap a = [a] }
   foldr1 (++) . map wrap . map root
=    { foldr1 (++) . map wrap = id
       on non-empty lists }
   map root
```

and

```
   tail . levelf
=    { levelf }
   tail . foldr lzc [] . map levelt
=    { foldr lzc [] = foldr1 lzc
       on non-empty lists }
   tail . foldr1 lzc . map levelt
=    { fold-map promotion: for non-empty xs, ys,
       tail (lzw f xs ys) = lzw f (tail xs) (tail ys) }
   foldr1 lzc . map tail . map levelt
=    { tail . levelt = levelf . kids }
   foldr1 lzc . map levelf . map kids
=    { foldr op e = foldr1 op on non-empty lists }
   foldr lzc [] . map levelf . map kids
=    { fold-concat promotion }
   levelf . concat . map kids
```

Therefore, `levelf` is an unfold as well as a fold:

```
levelf
  = unfold null (map root) (concat . map kids)
```

This gives us another linear-time algorithm for level-order traversal; this algorithm is no more complicated (indeed, it is arguably simpler) than the characterization as a fold, but it is more efficient. Moreover, we will see subsequently that it is also amenable to manipulation; to conclude this section, we will use *deforestation* [19] to eliminate the intermediate list of lists constructed during the breadth-first traversal.

5.1 Deforestation

As hinted at above, one of the benefits that accrues from expressing `levelf` as an unfold is that `bftf` is then a *hylomorphism*, that is, an unfold followed by a fold. Hylomorphisms proceed in two stages, the first producing a data structure and the second consuming it. With lazy evaluation, the intermediate complex data structure need never exist as a whole—the producer and consumer phases operate concurrently—but it is still advantageous to fuse the two phases into one, to reduce the amount of heap space turned over. This transformation is known as *deforestation* [19], and is now a standard technique; indeed, it can even be performed mechanically [10, 16].

In detail, we will use deforestation on functions of the form

```
h = foldr op e . unfold p f g
```

Consider first the case that `p` holds of the argument;

```
   h x
=    { h }
   foldr op e (unfold p f g x)
=    { assumption: p x holds; unfold }
   foldr op e []
=    { foldr }
   e
```

When `p x` does not hold, we have

```
   h x
=    { h }
   foldr op e (unfold p f g x)
=    { assumption: p x does not hold; unfold }
   foldr op e (f x : unfold p f g (g x))
=    { foldr }
   f x `op` foldr op e (unfold p f g (g x))
=    { h }
   f x `op` h (g x)
```

Therefore

```
h x
  | p x       = e
  | otherwise = f x `op` h (g x)
```

Applied to breadth-first traversal, deforestation gives the program

```
bftf ts
  | null ts   = []
  | otherwise = map root ts ++
                  bftf (concat (map kids ts))
```

(In fact, the version generated automatically by HYLO [16] also deforests away the `++` and the `concat . map kids`.) This program was shown to us by Bernhard Möller [15]. It is certainly elegant, but it is rather low-level; in particular, it uses recursion directly rather than encapsulating it with a higher-order operator. It is gratifying to find that this program arises as a compiler optimization from our more abstract characterization.

6 Traversal using a queue

It turns out that the standard queue-based traversal algorithm arises from expressing `bftf` directly as an unfold, starting from the characterization of level-order traversal as a fold. The calculation depends crucially on the following property of `lzw`: if `op` is associative, then

```
foldr op e (lzw op (x:xs) ys)
  = x `op` foldr op e (lzw op ys xs)
```

For example,

```
   concat (lzc [xs1,xs2] [ys1,ys2,ys3])
=    { lzc }
   concat [xs1 ++ ys1, xs2 ++ ys2, ys3]
=    { concat }
   (xs1 ++ ys1) ++ (xs2 ++ ys2) ++ ys3
=    { associativity }
   xs1 ++ (ys1 ++ xs2) ++ ys2 ++ ys3
=    { concat, lzc }
   xs1 ++ concat (lzc [ys1,ys2,ys3] [xs2])
```

The proof of this property, by induction on `xs`, is straightforward and is omitted.

Returning to traversal, clearly we have

```
bftf ts = []  ≡  null ts
```

For a non-null forest, we have

```
   bftf (Nd a us : ts)
=    { bftf, levelf }
   concat (foldr lzc []
     (map levelt (Nd a us : ts)))
=    { map, foldr }
   concat (lzc (levelt (Nd a us))
     (foldr lzc [] (map levelt ts)))
=    { levelf }
   concat (lzc (levelt (Nd a us)) (levelf ts))
=    { levelt }
   concat (lzc ([a] : levelf us) (levelf ts))
=    { crucial property }
   [a] ++ concat (lzc (levelf ts) (levelf us))
=    { fold-join promotion: levelf (ts ++ us) =
         lzc (levelf ts) (levelf us) }
   [a] ++ concat (levelf (ts ++ us))
=    { ++; bftf }
   a : bftf (ts ++ us)
```

Therefore, we have

```
bftf = unfold null f g
  where f (Nd a us : ts) = a
        g (Nd a us : ts) = ts ++ us
```

—the standard queue-based traversal algorithm. Again, it is rather low-level, and it is gratifying to be able to derive it from a more abstract specification.

Of course, this program is not linear-time: appending the children `us` of the first tree to the end of the queue `ts` takes time proportional to `length ts`, which grows linearly in the size of the tree, so the program is quadratic. To make it take linear time, we could use a clever data structure that allows queue operations in constant amortized time (for example, [9]), but here the simpler accumulating parameter technique [1] of using two lists, the second reversed, suffices. That is, the idea is to introduce a function `bftf'` such that

```
bftf' (ts,vs) = bftf (ts ++ reverse vs)
```

where `reverse` reverses a list; then

```
bftf ts = bftf' (ts,[])
```

and straightforward calculations lead to the characterization

```
bftf' :: (Forest a,Forest a) -> [a]
bftf' ([],[]) = []
bftf' ([],vs) = bftf' (reverse vs,[])
bftf' (Nd a us : ts,vs)
            = a : bftf' (ts,reverse us ++ vs)
```

of `bftf'`. In fact, `bftf'` is an unfold, too:

```
bftf' = unfold p f g
  where
    p (ts,vs)     = null ts && null vs
    f ([],vs)     = f (reverse vs,[])
    f (t:ts, vs) = root t
    g ([],vs)     = g (reverse vs,[])
    g (t:ts, vs) = (ts, reverse (kids t) ++ vs)
```

Expressing `bftf'` in this way entails two reversals of the second list, one for `f` and one for `g`, when the first list runs out. This is an artifact of our choice of characterization of `unfold`; the standard characterization would entail just one reversal.

Acknowledgements

This paper arose from a discussion late in 1992 on the Usenet newsgroup `comp.lang.functional`, on how to perform breadth-first traversal in a functional language. We wrote an earlier paper [11] giving derivations of our solutions, and also giving a rather complicated derivation of an algorithm for breadth-first *labelling*, a kind of inverse of breadth-first traversal originally posed to us by Joe Fasel: given is a tree `t` and a list `x`, and the problem is to construct a tree of the same shape as `t` but with breadth-first traversal `x`.

In response to that earlier paper, Bernhard Möller [15] showed us the elegant traversal algorithm presented here in Section 5.1. This led to the much simpler derivations presented here, and indirectly to our obervation that unfolds are greatly underappreciated, even in the functional programming community (not least by ourselves at the time).

We are grateful to the Problem Solving Club at Oxford University Computing Laboratory, and to attendees at the IFIP Working Group 2.1 meetings in Winnipeg and in Oxford, for their help in polishing the derivations presented here; also, we thank Zhenjiang Hu and Yoshiyuki Onoue for their experiments with the HYLO system.

References

[1] Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984. See also [2].

[2] Richard S. Bird. Addendum to "The promotion and accumulation strategies in transformational programming". *ACM Transactions on Programming Languages and Systems*, 7(3):490–492, July 1985.

[3] Richard S. Bird. *Introduction to Functional Programming Using Haskell.* Prentice-Hall, 1998.

[4] Richard S. Bird and Philip L. Wadler. *An Introduction to Functional Programming.* Prentice-Hall, 1988.

[5] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions. In *23rd ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 284–294, St Petersburg Beach, Florida, 1996.

[6] Jeremy Gibbons. *Algebras for Tree Algorithms.* D. Phil. thesis, Programming Research Group, Oxford University, 1991. Available as Technical Monograph PRG-94.

[7] Jeremy Gibbons. Deriving tidy drawings of trees. *Journal of Functional Programming*, 6(3):535–562, 1996. Earlier version appears as Technical Report No. 82, Department of Computer Science, University of Auckland.

[8] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *LNCS 283: Category Theory and Computer Science*, pages 140–157. Springer-Verlag, September 1987.

[9] Rob Hoogerwoord. A symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513, 1992.

[10] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *International Conference on Functional Programming.* ACM/SIGPLAN, 1996.

[11] Geraint Jones and Jeremy Gibbons. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Computer Science Report No. 71, Dept of Computer Science, University of Auckland, May 1993. Also IFIP Working Group 2.1 working paper 705 WIN-2.

[12] Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[13] Lambert Meertens. First steps towards the theory of rose trees. CWI, Amsterdam; IFIP Working Group 2.1 working paper 592 ROM-25, 1988.

[14] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *LNCS 523: Functional Programming Languages and Computer Architecture*, pages 124–144. Springer-Verlag, 1991.

[15] Bernhard Möller. Personal communication, May 1993.

[16] Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system HYLO. In Richard Bird and Lambert Meertens, editors, *Algorithmic Languages and Calculi*, pages 76–106. Chapman and Hall, 1997.

[17] John Peterson, Kevin Hammond, Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnsson, Mark Jones, Erik Meijer, Simon Peyton Jones, Alastair Reid, and Philip Wadler. The Haskell 1.4 report. `http://www.haskell.org/report/`, April 1997.

[18] Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[19] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[20] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.