Polytypic Downwards Accumulations

Jeremy Gibbons

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

Abstract. A *downwards accumulation* is a higher-order operation that distributes information downwards through a data structure, from the root towards the leaves. The concept was originally introduced in an ad hoc way for just a couple of kinds of tree. We generalize the concept to an arbitrary polynomial datatype; our generalization proceeds via the notion of a *path* in such a datatype.

1 Introduction

The notion of *scans* or *accumulations* on lists is well known, and has proved very fruitful for expressing and manipulating programs involving lists [5]. Gibbons [8, 9] generalized the notion of accumulation to various kinds of tree; that generalization too has proved fruitful, underlying the derivations of a number of tree algorithms, such as the parallel prefix algorithm for prefix sums [9], Reingold and Tilford's algorithm for drawing trees tidily [11], and algorithms for query evaluation in structured text [18].

There are two varieties of accumulation on lists, leftwards and rightwards. Leftwards accumulation labels every node of the list with some function of its *successors*—the tail segment starting at that node—thereby passing information from right to left along the list; rightwards accumulation labels every node with some function of its *predecessors*—the initial segment ending at that node—passing information from left to right. Similarly, there are two varieties of accumulation on trees, upwards and downwards. Upwards accumulation labels every node with some function of its *descendants*—the subtree rooted at that node—thereby passing information up the tree; downwards accumulation labels every node with some function of its *ancestors*—the path from the root to that node—passing information down the tree.

A flaw in the definitions of accumulations on trees from [8, 9] is that they were rather ad hoc. There is no formal relationship between accumulations on different kinds of tree, so each new kind of tree has to be considered from scratch. A recent trend in constructive algorithmics has been the development of theories

To appear in *Mathematics of Program Construction*, Marstrand, Sweden, June 1998.

of *generic* [1, 14] or *polytypic* [15] operations, parameterized by a datatype. (Another name for this kind of abstraction is *higher-order polymorphism*.) A generic program in this sense eliminates the unwanted ad-hockery. The categorical approach to datatypes popularized by Malcolm [16] is an early example of generic programming: it allows a single unified definition of operations such as map and fold, parameterized by the datatype concerned.

Bird *et al* [2] generalize upwards accumulation to an arbitrary polynomial datatype, unifying the previous ad hoc definitions. In this paper, we generalize downwards accumulation to an arbitrary polynomial datatype too. This is a more difficult problem: whereas the descendants of a node in a data structure (some kind of tree) form another data structure of the same type (another tree), the ancestors of a node are in general of a completely different type (a 'path').

Actually, Bird *et al* give a *generic* or *parametric higher-order polymorphic* definition of upwards accumulation, in the style of Hoogendijk and Backhouse, in the sense that their construction is based on *semantic* operations on the type functors involved. In contrast, the definition of downwards accumulation presented here is *polytypic* or *ad-hoc higher-order polymorphic*, in the style of Jeuring, based on *syntactic* properties of the type functor inducing the datatype. This is a problem with our approach: a generic definition would arguably be more elegant, but it is an open question whether a generic definition of downwards accumulation is possible.

We conclude this introductory section with a summary of notation. The remainder of the paper is structured as follows. In Section 2 we recall the monotypic definitions of upwards and downwards accumulations on trees from [8, 9]. We briefly summarize the theory of datatypes in Section 3. In Section 4 we discuss Bird *et al*'s [2] generic definition of upwards accumulations. In Section 5 we develop a polytypic definition of paths in a datatype, which we use in Section 6 to give a polytypic definition of downwards accumulations. This simple definition is inefficient; in Section 7 we show how to make it efficient. Section 8 concludes.

1.1 Functions

The type judgement '$a :: A$' declares that value a is of type A; the type $A \to B$ denotes the type of functions from A to B. Function application is denoted by juxtaposition; the identity function is written id, so that $\mathsf{id}\, a = a$ for every a. The unit type 1 has just one element; there is a unique total function $\mathsf{one} :: A \to 1$ for every type A.

1.2 The Pair Calculus

The functors $+$ and $\times$ denote disjoint sum and cartesian product respectively; $\times$ binds tighter than $+$. The product projections are fst and snd, and the product morphism $f \vartriangle g$ has type $A \to B \times C$ when $f :: A \to B$ and $g :: A \to C$; the sum morphism $f \triangledown g$ has type $A + B \to C$ when $f :: A \to C$ and $g :: B \to C$. We write '$\sum_{i=1}^{n} A_i$' and '$\prod_{i=1}^{n} A_i$' for generalized sum and product; the generalized sum injections are inj_i and the generalized sum morphism is $\bigtriangledown_{i=1}^{n} f_i$.

2 Accumulations on Binary Trees

To provide motivation and intuition for what follows, we review here the 'monotypic' definitions of upwards and downwards accumulations on binary trees, as presented in [8, 9]. Bird *et al*'s generalization of upwards accumulation and our generalization of downwards accumulation, when specialized to the same kind of tree, reduce essentially to these monotypic definitions.

2.1 Homogeneous Binary Trees

We will use as an example the datatype of *homogeneous binary trees*, that is, trees with internal and external labels of the same type. In Haskell, these can be modelled by the type definition

```
data Tree a = Leaf a | Bin a (Tree a) (Tree a)
```

Thus, a tree is either a leaf with a label, or a node with a label and two subtrees.

Two higher-order operations on trees that we shall need in the following are a map and a fold. Again in Haskell, these are defined by

```
mapTree :: (a->b) -> Tree a -> Tree b
mapTree f (Leaf a) = Leaf (f a)
mapTree f (Bin a t u) = Bin (f a) (mapTree f t) (mapTree f u)

foldTree :: (a->b, a->b->b->b) -> Tree a -> b
foldTree (g,h) (Leaf a) = g a
foldTree (g,h) (Bin a t u) = h a (foldTree (g,h) t)
                                 (foldTree (g,h) u)
```

In fact, `mapTree` can be written as a fold:

```
mapTree f = foldTree (g,h) where
              g a = Leaf (f a)
              h a t u = Bin (f a) t u
```

or, shorter but more cryptically,

```
mapTree f = foldTree (Leaf . f, Bin . f)
```

A simpler example of a fold is the function `size`, which returns the size of (that is, the number of elements in) a tree:

```
size :: Tree a -> Int
size = foldTree (g,h) where g a = 1
                            h a m n = 1+m+n
```

2.2 Monotypic Upwards Accumulations

Upwards accumulations are defined in terms of the function `subtrees`, which labels every node with the subtree rooted at that node.

```
subtrees :: Tree a -> Tree (Tree a)
subtrees (Leaf a) = Leaf (Leaf a)
subtrees (Bin a t u) = Bin (Bin a t u) (subtrees t) (subtrees u)
```

Now upwards accumulation is simple to define: it is merely a fold mapped over the subtrees.

```
upTree :: (a->b, a->b->b->b) -> Tree a -> Tree b
upTree (g,h) = mapTree (foldTree (g,h)) . subtrees
```

For example, the function `sizes`, which labels every node with the number of descendants it has, is given by

```
sizes :: Tree a -> Tree Int
sizes = upTree (g,h) where g a = 1
                           h a m n = 1+m+n
```

Of course, this is a very inefficient definition: it does not capitalize on the fact that the folds of the children of a node can be used in computing the fold of the node itself. However, a simple application of *deforestation* [19], together with a little manipulation using the observation that

```
foldTree (g,h) = root . upTree (g,h)
```

where the function `root` returns the root label of a tree:

```
root :: Tree a -> a
root (Leaf a) = a
root (Bin a t u) = a
```

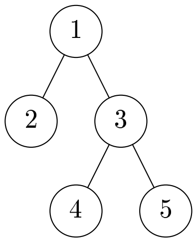
removes this inefficiency:

```
upTree (g,h) (Leaf a) = Leaf (g a)
upTree (g,h) (Bin a t u) = Bin (h a (root t') (root u')) t' u'
                             where t' = upTree (g,h) t
                                   u' = upTree (g,h) u
```

With this improved definition, computing `upTree (g,h)` takes essentially no longer than computing `foldTree (g,h)`.

2.3 Monotypic Downwards Accumulations

As noted earlier, downwards accumulations are more complicated than upwards accumulations because they involve another datatype. Consider the tree



which is represented by the expression

```
Bin 1 (Leaf 2) (Bin 3 (Leaf 4) (Leaf 5))
```

Intuitively, the path from the root to a node of this tree is either a singleton, or consists of a label followed by a left or a right turn to another path. We therefore model paths by the datatype

```
data Path a = Single a | TurnL a (Path a) | TurnR a (Path a)
```

For example, the path to the node labelled 2 is `TurnL 1 (Single 2)`, and the path to the node labelled 4 is `TurnR 1 (TurnL 3 (Single 4))`.

Now, the function `paths` labels every node of the tree with the path from the root to that node. We have

```
paths :: Tree a -> Tree (Path a)
paths (Leaf a) = Leaf (Single a)
paths (Bin a t u) = Bin (Single a) (mapTree (TurnL a) (paths t))
                                   (mapTree (TurnR a) (paths u))
```

Downwards accumulation is a fold mapped over the paths, but this fold has to be a path fold. We therefore define the function `foldPath` by

```
foldPath :: (a->b, a->b->b, a->b->b) -> Path a -> b
foldPath (f,g,h) (Single a) = f a
foldPath (f,g,h) (TurnL a p) = g a (foldPath (f,g,h) p)
foldPath (f,g,h) (TurnR a p) = h a (foldPath (f,g,h) p)
```

For example, the function `pLength`, which returns the length of a path, is given by

```
pLength :: Path a -> Int
pLength = foldPath (f,g,g) where f a = 1
                                 g a n = 1+n
```

Now downwards accumulation can be defined simply by

```
daTree :: (a->b, a->b->b, a->b->b) -> Tree a -> Tree b
daTree (f,g,h) = mapTree (foldPath (f,g,h)) . paths
```

For example, the function `depths`, which labels every node of a tree with its depth, is simply

```
depths :: Tree a -> Tree Int
depths = daTree (f,g,g) where f a = 1
                              g a n = 1+n
```

2.4 Complications

The monotypic example given above is simplistic in three ways, apart from the very fact that it is monotypic and not polytypic.

The first simplification we made was to choose the datatype of homogeneous trees carefully, so that every node has a label at which to store the value computed by the accumulations. In general, not all nodes of a data structure possess such a label, so an accumulation will in general not return a data structure of the same type as its argument. Rather, Bird *et al* show how to define a *labelled variant* of a datatype, adding labels in all the right places, in a generic way. This construction is discussed in Section 4.2.

The second simplification is that we did not distinguish between the path to the node labelled `1` in the tree given above, and the path to the node labelled `1` in the singleton tree `Leaf 1`: the former path ends at an internal node of the original data structure, whereas the latter ends at an external node, although both contain the same 'data'. In fact, the datatype of paths that we construct in Section 5.1 will have two singleton constructors, one for paths ending at internal nodes and one for paths ending at external nodes:

```
data Path a = SingleI a | SingleE a | TurnL a (Path a) | TurnR a (Path a)
```

The third problem is that the function `paths` and downwards accumulation as defined above are inefficient, for the same reason that the first attempt at upwards accumulation was inefficient: we cannot afford the two maps over the children, but we can (sometimes) reuse the value computed at a parent in computing the values at the children. We address this problem in Section 7.

3 Datatypes

In this section, we briefly review the construction of polynomial datatypes, in the style of Malcolm [16]. Given a bifunctor F, the datatype T is the type functor such that the type $T(A)$ is isomorphic to $F(A, T(A))$; the isomorphism is provided by the constructor $\mathsf{in}_F :: F(A, T(A)) \to T(A)$ and the destructor $\mathsf{out}_F :: T(A) \to F(A, T(A))$. We call $T(A)$ the *canonical fixpoint* of the functor $F(A, \_)$. A strict function $\phi :: F(A, B) \to B$ induces a fold $\mathsf{fold}_F\, \phi :: T(A) \to B$, and a function $\psi :: B \to F(A, B)$ induces an unfold $\mathsf{unfold}_F\, \psi :: B \to T(A)$.

Example 1. We will use the following datatypes as running examples throughout the paper.

1. Leaf-labelled binary trees are built from the functor $F(A, X) = A + X \times X$:

   ```
   data Ltree a  =  LeafT a | BinT (Ltree a) (Ltree a)
   ```

   The corresponding fold is

   ```
   foldT :: (Either a (b,b) -> b) -> Ltree a -> b
   foldT phi (LeafT a) = phi (Left a)
   foldT phi (BinT t u) = phi (Right (foldT phi t, foldT phi u))
   ```

For example, the function `sizeT`, which returns the size of a leaf-labelled binary tree, is a fold:

```
sizeT :: Ltree a -> Int
sizeT = foldT phiSizeT
phiSizeT (Left a) = 1
phiSizeT (Right (m,n)) = m+n
```

(We will use `phiSizeT` later.)

2. Branch-labelled binary trees are constructed from the functor $F(A, X) = 1 + A \times X \times X$:

```
data Btree a  =  Empty | BinB a (Btree a) (Btree a)
```

The corresponding fold is

```
foldB :: (Either () (a,b,b) -> b) -> Btree a -> b
foldB phi Empty = phi (Left ())
foldB phi (BinB a t u) =
  phi (Right (a, foldB phi t, foldB phi u))
```

The function `sizeB` is defined as follows:

```
sizeB :: Btree a -> Int
sizeB = foldB phiSizeB
phiSizeB (Left ()) = 0
phiSizeB (Right (a,m,n)) = 1+m+n
```

3. A funny kind of tree can be constructed from the functor $F(A, X) = \mathsf{Int} + A \times X + \mathsf{Bool} \times X \times X$:

```
data Ftree a = TipF Int | MonF a (Ftree a) |
               BinF Bool (Ftree a) (Ftree a)
```

A tree of type `Ftree a` is either a terminal node `TipF` with an integer label, a parent with a label of type `a` and a single child, or a parent with a boolean label and two children. The corresponding fold is

```
foldF :: (Either3 Int (a,b) (Bool,b,b) -> b) -> Ftree a -> b
foldF phi (TipF n) = phi (In1 n)
foldF phi (MonF a x) = phi (In2 (a, foldF phi x))
foldF phi (BinF b x y) = phi (In3 (b, foldF phi x, foldF phi y))
```

where

```
data Either3 a b c = In1 a | In2 b | In3 c
```

The function `sizeF` is defined as follows:

```
sizeF :: Ftree a -> Int
sizeF = foldF phiSizeF
phiSizeF :: Either3 Int (a,Int) (Bool,Int,Int) -> Int
phiSizeF (In1 n) = 0
phiSizeF (In2 (a,n)) = 1+n
phiSizeF (In3 (b,m,n)) = m+n
```

(It is not so obvious what the 'size' of such a bizarre species of tree should be. This definition simply counts the number of occurrences of the type parameter `a`, so that for example the rather large tree

```
BinF True (TipF 5) (BinF False (TipF 6) (TipF 7))
```

still has size 0. There are other reasonable definitions.)

□

4 Polytypic Upwards Accumulations

Bird *et al* [2] generalized upwards accumulations to an arbitrary polynomial datatype, giving *polytypic upwards accumulations*. We summarize their construction here. It is related to, but not the same as, Meertens' generic definition of the 'predecessors' of a data structure [17]. This section serves partly as motivation and 'intuitive support', as upwards accumulations are simpler than downwards accumulations. More importantly, however, we will use part of their construction ourselves in Section 6.

4.1 The Essential Idea

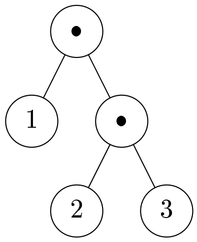
As suggested in Section 2, an upwards accumulations on a tree is related to the subtrees of that tree; in general, an upwards accumulation on a data structure is related to the substructures of that data structure. A substructure of a structure is a subterm of the term representing that structure. We will define a function subs_F, which generates a structure of substructures from a structure: a list of sublists from a list, a tree of subtrees from a tree, and so on.

Upwards accumulations are defined in terms of substructures. The upwards accumulation of a data structure consists of folding every substructure of that structure; in other words, an upwards accumulation is a fold mapped over the substructures.

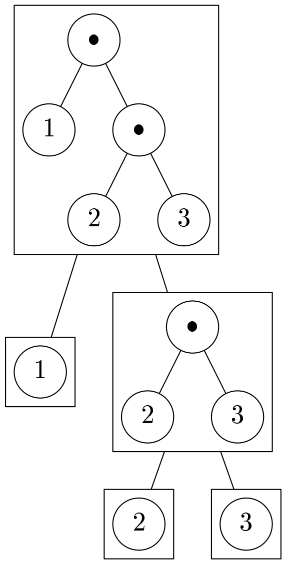
The definitions in this section generalize those of Section 2.2.

4.2 Labelled Types

The function subs_F takes a data structure and returns another data structure, with every node 'labelled' with the corresponding substructure of the original data structure rooted at that node. Therefore, in general, the source and target types of subs_F will not be the same: the target type must have labels at every node, whereas the source type may not. For example, subs_F should take a leaf-labelled binary tree to a homogeneous binary tree of leaf-labelled binary trees. In particular, we want subs_F to take the leaf-labelled tree



to the homogeneous binary tree of leaf-labelled binary trees



Bird *et al* call the datatype of homogeneous binary trees the *labelled variant* of the type of leaf-labelled binary trees, and give a general construction for it, as follows.

Definition 2. Given a datatype $T(A)$, the canonical fixpoint of a functor $F(A, \_)$, the corresponding labelled type $L(A)$ is the canonical fixpoint of the functor $G(A, \_)$, where G is defined by $G(A, X) = A \times F\,(1, X)$. $\square$

Informally, $F\,(1, X)$ is like $F\,(A, X)$, but with all the labels (of type A) removed; thus, using $A \times F\,(1, X)$ ensures that every node carries precisely one label.

Example 3.

1. The datatype of leaf-labelled binary trees is constructed from the functor $F(A, X) = A + X \times X$, so the functor G is given by

$$\begin{aligned} G(A, X) &= A \times F(1, X) \\ &= A \times (1 + X \times X) \end{aligned}$$

   This induces the labelled type of homogeneous binary trees, as desired:

```
data Htree a  =  ConsH a (Either () (Htree a,Htree a))
```

(This is isomorphic to the type `Tree` of Section 2.) The unfold for `Htree` is

```
unfoldH :: (b -> (a, Either () (b,b))) -> b -> Htree a
unfoldH phi b = f (phi b)
  where
    f (a, Left ()) = ConsH a (Left ())
    f (a, Right (b',b'')) = ConsH a (Right (unfoldH phi b',
                                            unfoldH phi b''))
```

2. The datatype of branch-labelled binary trees is constructed from the functor $F(A, X) = 1 + A \times X \times X$, so the functor G is given by

$$\begin{aligned} G(A, X) &= A \times F(1, X) \\ &= A \times (1 + 1 \times X \times X) \\ &\approx A \times (1 + X \times X) \end{aligned}$$

and induces the labelled type of homogeneous binary trees, just as for leaf-labelled binary trees.

3. The datatype of homogeneous binary trees is constructed from the functor $F(A, X) = A \times (1 + X \times X)$, so the functor G is given by

$$\begin{aligned} G(A, X) &= A \times F(1, X) \\ &= A \times (1 \times (1 + X \times X)) \\ &\approx A \times (1 + X \times X) \end{aligned}$$

and so homogeneous binary trees are their own labelled type. In general, if $G(A, X) = A \times F(1, X)$ then $A \times G(1, X) \approx G(A, X)$, that is, 'constructing the labelled variant' is an idempotent operation.

4. The datatype of funny trees is constructed from the functor $F(A, X) = \mathsf{Int} + A \times X + \mathsf{Bool} \times X \times X$, so the functor G is given by

$$\begin{aligned} G(A, X) &= A \times F(1, X) \\ &= A \times (\mathsf{Int} + 1 \times X + \mathsf{Bool} \times X \times X) \\ &\approx A \times (\mathsf{Int} + X + \mathsf{Bool} \times X \times X) \end{aligned}$$

For want of a better name, we call the labelled variant so induced 'odd trees':

```
data Otree a =
  ConsO a (Either3 Int (Otree a) (Bool,Otree a,Otree a))
```

The corresponding unfold is

```
unfoldO :: (b -> (a, Either3 Int b (Bool,b,b))) -> b -> Otree a
unfoldO phi b = f (phi b)
  where
    f (a, In1 n)  = ConsO a (In1 n)
    f (a, In2 b') = ConsO a (In2 (unfoldO phi b'))
    f (a, In3 (bool,b',b''))
                  = ConsO a (In3 (bool, unfoldO phi b',
                                        unfoldO phi b''))
```

□

We define the functions root and kids on labelled types only, as follows.

Definition 4. The functions $\mathsf{root} :: L(A) \to A$ and $\mathsf{kids} :: L(A) \to F(1, L(A))$ are defined by

$$\begin{aligned} \mathsf{root} &= \mathsf{fst} \circ \mathsf{out}_G \\ \mathsf{kids} &= \mathsf{snd} \circ \mathsf{out}_G \end{aligned}$$

(Remember that $G(A, X) = A \times F(1, X)$, so that out_G returns a pair of type $A \times F(1, L(A))$.) □

Example 5.

1. For homogeneous binary trees, we have

```
rootH :: Htree a -> a
rootH (ConsH a x) = a
```

2. For odd trees, we have

```
rootO :: Otree a -> a
rootO (ConsO a x) = a
```

□

4.3 Substructures

The function subs_F has type $T(A) \to L(T(A))$, where T and L are the original and labelled types, as defined above.

Definition 6. The function $\mathsf{subs}_F :: T(A) \to L(T(A))$ is given by

$$\mathsf{subs}_F = \mathsf{unfold}_G\,(\mathsf{id} \vartriangle (F(\mathsf{one}, \mathsf{id}) \circ \mathsf{out}_F))$$

□

Thus, the root of the substructures of a data structure x is x itself:

Theorem 7.

$$\mathsf{root} \circ \mathsf{subs}_F = \mathsf{id}$$

□

Proof. We have

$$\begin{aligned} & \mathsf{root} \circ \mathsf{subs}_F \\ = & \quad \{ \text{ root; } \mathsf{subs}_F \ \} \\ & \mathsf{fst} \circ \mathsf{out}_G \circ \mathsf{unfold}_G\,(\mathsf{id} \vartriangle (F(\mathsf{one}, \mathsf{id}) \circ \mathsf{out}_F)) \\ = & \quad \{ \text{ unfold } \} \\ & \mathsf{fst} \circ G\,(\mathsf{id}, \mathsf{subs}_F) \circ (\mathsf{id} \vartriangle (F(\mathsf{one}, \mathsf{id}) \circ \mathsf{out}_F)) \\ = & \quad \{ \ \mathsf{fst} \circ G\,(f, g) = \mathsf{fst} \circ (f \times F(\mathsf{id}, g)) = f \circ \mathsf{fst} \ \} \\ & \mathsf{fst} \circ (\mathsf{id} \vartriangle (F(\mathsf{one}, \mathsf{id}) \circ \mathsf{out}_F)) \\ = & \quad \{ \text{ pairs } \} \\ & \mathsf{id} \end{aligned}$$

□

Actually, Bird *et al* give an alternative definition of subs_F as a fold:

$$\mathsf{subs}_F = \mathsf{fold}_F\,(\mathsf{in}_G \circ (\mathsf{in}_F \circ F(\mathsf{id}, \mathsf{root})) \vartriangle F(\mathsf{one}, \mathsf{id}))$$

We claim that the definition in terms of an unfold is simpler and more natural. As a general observation, we believe that unfolds are under-appreciated, even amongst functional programming experts, who should be aficionados of higher-order operators [13]. On the other hand, it turns out that our definition as an unfold leads to an inefficient characterization of upwards accumulations, as we shall see shortly; in removing the inefficiency, we end up with exactly Bird *et al*'s definition. Nevertheless, it is better to start from the natural but inefficient characterization and derive from it the less natural but more efficient characterization.

Example 8.

1. For leaf-labelled binary trees, the corresponding labelled type is homogeneous binary trees, so `subsT` is defined in terms of `unfoldH`:

```
subsT :: Ltree a -> Htree (Ltree a)
subsT = unfoldH phi
  where phi (LeafT a) = (LeafT a, Left ())
        phi (BinT t u) = (BinT t u, Right (t, u))
```

2. For branch-labelled binary trees too, the labelled variant is homogeneous binary trees:

```
subsB :: Btree a -> Htree (Btree a)
subsB = unfoldH phi
  where phi Empty = (Empty, Left ())
        phi (BinB a t u) = (BinB a t u, Right (t, u))
```

3. For funny trees, the corresponding labelled type is odd trees, so we use `unfoldO`:

```
subsF :: Ftree a -> Otree (Ftree a)
subsF = unfoldO phi
  where phi (TipF n) = (TipF n, In1 n)
        phi (MonF a x) = (MonF a x, In2 x)
        phi (BinF b x y) = (BinF b x y, In3 (b, x, y))
```

□

4.4 Upwards Accumulations

Given subs_F, upwards accumulation is easy to define: it is simply a fold mapped over the substructures.

Definition 9. Upwards accumulation $\mathsf{ua}_F :: (F(A, B) \to B) \to T(A) \to L(B)$ is defined by

$$\mathsf{ua}_F\ \phi = L\,(\mathsf{fold}_F\ \phi) \circ \mathsf{subs}_F$$

□

Corollary 10.

$$\mathsf{root} \circ \mathsf{ua}_F\ \phi = \mathsf{fold}_F\ \phi$$

Proof. Immediate from Theorem 7 and naturality of root:

$$\mathsf{root} \circ L\, f = f \circ \mathsf{root}$$

□

Of course, taken literally, Definition 9 makes rather an inefficient program: it does not exploit the fact that the results of folding the children of a node can be reused in folding the substructure rooted at the node itself. This efficiency can be removed by *deforestation* [19], one corollary of which states that an unfold followed by a fold can be fused into a single recursion:

Lemma 11 (Deforestation). *Suppose* $h = \mathsf{fold}_F\ \psi \circ \mathsf{unfold}_F\ \phi$. *Then*

$$h = \psi \circ F\,(\mathsf{id}, h) \circ \phi$$

□

Proof. We have

$$\begin{array}{cl}
& h \\
= & \{\ \text{hypothesis}\ \} \\
& \mathsf{fold}_F\ \psi \circ \mathsf{unfold}_F\ \phi \\
= & \{\ \mathsf{in}_F \circ \mathsf{out}_F = \mathsf{id}\ \} \\
& \mathsf{fold}_F\ \psi \circ \mathsf{in}_F \circ \mathsf{out}_F \circ \mathsf{unfold}_F\ \phi \\
= & \{\ \text{folds, unfolds}\ \} \\
& \psi \circ F\,(\mathsf{id}, \mathsf{fold}_F\ \psi) \circ F\,(\mathsf{id}, \mathsf{unfold}_F\ \phi) \circ \phi \\
= & \{\ \text{functors}\ \} \\
& \psi \circ F\,(\mathsf{id}, \mathsf{fold}_F\ \psi \circ \mathsf{unfold}_F\ \phi) \circ \phi \\
= & \{\ \text{hypothesis}\ \} \\
& \psi \circ F\,(\mathsf{id}, h) \circ \phi
\end{array}$$

□

Now we can write $\mathsf{ua}_F\ \phi$ as a fold:

Theorem 12.

$$\mathsf{ua}_F\ \phi = \mathsf{fold}_F\ (\mathsf{in}_G \circ (\phi \circ F(\mathsf{id}, \mathsf{root})) \vartriangle F(\mathsf{one}, \mathsf{id}))$$

□

Proof. To write $\mathsf{ua}_F\ \phi$ as a fold $\mathsf{fold}_F\ \psi$, we have to construct a ψ for which

$$\mathsf{ua}_F\ \phi \circ \mathsf{in}_F = \psi \circ F(\mathsf{id}, \mathsf{ua}_F\ \phi)$$

We have

$$
\begin{array}{cl}
& \mathsf{ua}_F\ \phi \circ \mathsf{in}_F \\
= & \quad \{\ \mathsf{ua}\ \} \\
& L\,(\mathsf{fold}_F\ \phi) \circ \mathsf{subs}_F \circ \mathsf{in}_F \\
= & \quad \{\ L \text{ as a fold; } \mathsf{subs}_F\ \} \\
& \mathsf{fold}_G\,(\mathsf{in}_G \circ G\,(\mathsf{fold}_F\ \phi, \mathsf{id})) \circ \mathsf{unfold}_G\,(\mathsf{id} \vartriangle (F(\mathsf{one}, \mathsf{id}) \circ \mathsf{out}_F)) \circ \mathsf{in}_F \\
= & \quad \{\ \text{Lemma 11}\ \} \\
& \mathsf{in}_G \circ G\,(\mathsf{fold}_F\ \phi, \mathsf{id}) \circ G\,(\mathsf{id}, \mathsf{ua}_F\ \phi) \circ (\mathsf{id} \vartriangle (F\,(\mathsf{one}, \mathsf{id}) \circ \mathsf{out}_F)) \circ \mathsf{in}_F \\
= & \quad \{\ \text{functors}\ \} \\
& \mathsf{in}_G \circ G\,(\mathsf{fold}_F\ \phi, \mathsf{ua}_F\ \phi) \circ (\mathsf{id} \vartriangle (F\,(\mathsf{one}, \mathsf{id}) \circ \mathsf{out}_F)) \circ \mathsf{in}_F \\
= & \quad \{\ \text{pairs; } \mathsf{out}_F \circ \mathsf{in}_F = \mathsf{id}\ \} \\
& \mathsf{in}_G \circ G\,(\mathsf{fold}_F\ \phi, \mathsf{ua}_F\ \phi) \circ (\mathsf{in}_F \vartriangle F\,(\mathsf{one}, \mathsf{id})) \\
= & \quad \{\ G\ \} \\
& \mathsf{in}_G \circ (\mathsf{fold}_F\ \phi \times F\,(\mathsf{id}, \mathsf{ua}_F\ \phi)) \circ (\mathsf{in}_F \vartriangle F\,(\mathsf{one}, \mathsf{id})) \\
= & \quad \{\ \text{pairs}\ \} \\
& \mathsf{in}_G \circ ((\mathsf{fold}_F\ \phi \circ \mathsf{in}_F) \vartriangle (F\,(\mathsf{id}, \mathsf{ua}_F\ \phi) \circ F\,(\mathsf{one}, \mathsf{id}))) \\
= & \quad \{\ \text{fold; functors}\ \} \\
& \mathsf{in}_G \circ ((\phi \circ F\,(\mathsf{id}, \mathsf{fold}_F\ \phi)) \vartriangle F\,(\mathsf{one}, \mathsf{ua}_F\ \phi)) \\
= & \quad \{\ \text{Corollary 10; pairs}\ \} \\
& \mathsf{in}_G \circ ((\phi \circ F\,(\mathsf{id}, \mathsf{root})) \vartriangle F\,(\mathsf{one}, \mathsf{id})) \circ F\,(\mathsf{id}, \mathsf{ua}_F\ \phi)
\end{array}
$$

and so

$$\psi = \mathsf{in}_G \circ ((\phi \circ F\,(\mathsf{id}, \mathsf{root})) \vartriangle F\,(\mathsf{one}, \mathsf{id}))$$

□

With this improved characterization, the upwards accumulation $\mathsf{ua}_F\ \phi$ takes asymptotically no longer to compute than the ordinary fold $\mathsf{fold}_F\ \phi$.

Example 13.

1. For leaf-labelled binary trees, we have

```
uaT :: (Either a (b,b) -> b) -> Ltree a -> Htree b
uaT phi = foldT psi
  where
    psi (Left a) = ConsH (phi (Left a)) (Left ())
    psi (Right (t,u))
      = ConsH (phi (Right (rootH t, rootH u))) (Right (t,u))
```

   For example, the function `sizesT` labels every node of a leaf-labelled binary tree with the size of the subtree rooted there:

```
sizesT :: Ltree a -> Htree Int
sizesT = uaT phiSizeT
```

   where `phiSizeT` is as defined in Example 1.

2. For branch-labelled binary trees, we have

```
uaB :: (Either () (a,b,b) -> b) -> Btree a -> Htree b
uaB phi = foldB psi
  where
    psi (Left ()) = ConsH (phi (Left ())) (Left ())
    psi (Right (a,t,u))
      = ConsH (phi (Right (a, rootH t, rootH u))) (Right (t,u))
```

   The corresponding 'sizes' function on these trees is

```
sizesB :: Btree a -> Htree Int
sizesB = uaB phiSizeB
```

3. For funny trees we have

```
uaF :: (Either3 Int (a,b) (Bool,b,b) -> b) -> Ftree a -> Otree b
uaF phi = foldF psi
  where
    psi (In1 n) = ConsO (phi (In1 n)) (In1 n)
    psi (In2 (a,x)) = ConsO (phi (In2 (a, rootO x))) (In2 x)
    psi (In3 (b,x,y))
      = ConsO (phi (In3 (b, rootO x, rootO y))) (In3 (b,x,y))
```

   The corresponding 'sizes' function is

```
sizesF :: Ftree a -> Otree Int
sizesF = uaF phiSizeF
```

□

5 Polytypic Paths

As described in the previous section, an *upwards* accumulation on a data structure is a fold mapped over the substructures of that structure; for example, upwards accumulation on a binary tree is a tree fold mapped over the subtrees of that tree. A substructure of a data structure of type $T(A)$ is itself of type $T(A)$. Thus, an upwards accumulation replaces every node of a data structure with some function of the descendants of that node. In contrast, a *downwards* accumulation replaces every node of a data structure with some function of the *ancestors* of that node; a downwards accumulation is a fold mapped over the *paths*, where the path to a node in a data structure represents the ancestors of that node. A path in a data structure is in general of a completely different type from the data structure itself; in particular, a path is necessarily a linear structure, each parent having a single child, whereas the data structure itself may be non-linear. So the first thing we need in developing a polytypic definition of downwards accumulations is a polytypic notion of paths.

5.1 Linearization

Recall that a datatype $T(A)$ is built as the canonical fixpoint of the functor $F(A, \_)$ for some bifunctor F. For example, leaf-labelled binary trees are built from the functor F given by

$$F(A, X) = A + X \times X$$

The first step in constructing the appropriate notion of paths is to partition F into a sum of products; that is, we suppose that

$$F(A, X) = \sum_{i=1}^{n} F_i(A, X)$$

where each F_i is a product of literals:

$$F_i(A, X) = \prod_{j=1}^{m_i} F_{i,j}(A, X)$$

where each $F_{i,j}(A, X)$ is either A, X or some constant type such as *Int* or *Bool*. For example, for leaf-labelled binary trees, $n = 2$, $m_1 = 1$ and $m_2 = 2$, and we have

$$\begin{array}{ll} F(A, X) & = F_1(A, X) + F_2(A, X) \\ F_1(A, X) & = F_{1,1}(A, X) \\ F_{1,1}(A, X) & = A \\ F_2(A, X) & = F_{2,1}(A, X) \times F_{2,2}(A, X) \\ F_{2,1}(A, X) & = X \\ F_{2,2}(A, X) & = X \end{array}$$

Now, the $F_i(A, X)$ will in general be 'non-linear', in the sense that they may contain no X or more than one X. Paths, on the other hand, are inherently linear. Therefore, for each bifunctor F_i we construct a multi-functor F_i', linear in each argument except perhaps the first; informally, this multi-functor will be a $(k_i + 1)$-functor where $F_i(A, X)$ contains k_i occurrences of X. For example, for leaf-labelled binary trees we have $k_1 = 0$ (because $F_1(A, X) = A$ contains no occurrences of X) and $k_2 = 2$ (because $F_2(A, X) = X \times X$ contains two occurrences of X), and so we define

$$\begin{array}{ll} F_1'(A) & = A \\ F_2'(A, X_1, X_2) & = X_1 \times X_2 \end{array}$$

Notice that $F_1'(A) = F_1(A, X)$ and $F_2'(A, X, X) = F_2(A, X)$.

We formalize the above as follows.

Definition 14. A functor H is *linear* if it distributes over sum:

$$H(X + Y) \approx H(X) + H(Y)$$

□

In particular, a functor constructing a product of literals is linear when its argument is used precisely once, and so neither $H(X) = A$ nor $H(X) = X \times X$ are linear.

For each bifunctor F_i, we construct the linearization (a $(k_i + 1)$-functor F'_i) of the functor $F_i(A, \_)$, as follows. Abusing the term, we call F'_i 'the linearization of F_i' instead of 'the linearization of $F_i(A, \_)$'.

Definition 15. Suppose bifunctor H is a product of literals. A *linearization* of H is a $(k + 1)$-functor H' such that:

– H' is a generalization of H:

$$H'(A, \underbrace{X, \ldots, X}_{k \text{ times}}) = H(A, X)$$

– H' is linear in all arguments, except perhaps the first:

$$\begin{aligned} &H'(A, X_1, \ldots, X_{j-1}, Y + Z, X_{j+1}, \ldots, X_k) \\ &\qquad \approx H'(A, X_1, \ldots, X_{j-1}, Y, X_{j+1}, \ldots, X_k) + \\ &\qquad\quad\; H'(A, X_1, \ldots, X_{j-1}, Z, X_{j+1}, \ldots, X_k) \end{aligned}$$

for $1 \leq j \leq k$.

□

Note that the linearization is unique up to isomorphism: for $H(A, X) = X \times X$, for example, there are just the two isomorphic linearizations

$$H'(A, X_1, X_2) = X_1 \times X_2 \approx X_2 \times X_1$$

In particular, all linearizations of a functor have the same number of arguments.

To define downwards accumulations in Section 6 we must choose a linearization F'_i of each F_i, but to construct the path type in Section 5.2 we need only the k_i. We call the number k_i the 'degree of branching' of the bifunctor F_i:

Definition 16. Suppose bifunctor H is a product of literals. Then the *degree of branching* of H is the natural number k such that linearizations of H are $(k + 1)$-functors. □

5.2 The Path Type

Consider the leaf-labelled binary tree illustrated on page 9:

```
BinT (LeafT 1) (BinT (LeafT 2) (LeafT 3))
```

This data structure has five nodes, the paths to each being as follows:

- the path to the root node is just 'empty';
- the path to the other internal node is 'right, then empty';
- the path to the node labelled `1` is 'left, then the label `1`';
- the path to the node labelled `2` is 'right, then left, then the label `2`';
- the path to the node labelled `3` is 'right, then right, then the label `3`'.

Thus, a path is essentially a kind of cons list, and that is how we define the type of paths below.

The last element of a path is the label at some node of the data structure. That is, the last element of a path is constructed from information of type $F(A,1)$. For example, for leaf-labelled binary trees $F(A,1) = A+1$; the last element of the path to the root node of our example tree is 'nothing', and the last element of the path to the node labelled `1` is 'the label `1`'.

Every other element of a path corresponds to an internal node of some variant i of the data structure, and is constructed from the label at that node together with a record of which direction to go next. That is, internal elements of a path are constructed from information of type $\sum_{i=1}^{n}(F_i(A,1) \times \{1,\ldots,k_i\})$ (or equivalently, $\sum_{i=1}^{n}\sum_{j=1}^{k_i} F_i(A,1)$) and another path. For example, for leaf-labelled binary trees,

$$\sum_{i=1}^{n}\sum_{j=1}^{k_i} F_i(A,1) = F_2(A,1) + F_2(A,1) \approx 1+1$$

and the internal elements of paths are all either 'left turn' or 'right turn', followed by another path.

We therefore construct the path type as follows.

Definition 17. The path type $P(A)$ is the canonical fixpoint of the bifunctor H defined by

$$H(A,X) = F(A,1) + \sum_{i=1}^{n}\sum_{j=1}^{k_i}(F_i(A,1) \times X)$$

We will use a constructor endp for the last element of a path, and a family of constructors $\mathsf{consp}_{i,j}$ for $1 \le i \le n$ and $1 \le j \le k_i$ for the other elements:

$$\begin{aligned} &\mathsf{endp} && :: F(A,1) \to P(A) \\ &\mathsf{consp}_{i,j} && :: F_i(A,1) \times P(A) \to P(A) \end{aligned}$$

(Note that the path constructed by endp need not be 'empty'; it could be a 'singleton'.) □

Example 18.

1. For leaf-labelled binary trees we have

   $$\begin{aligned} H(A,X) &= (A + 1 \times 1) + (1 \times 1 \times X + 1 \times 1 \times X) \\ &\approx (1+A) + X + X \end{aligned}$$

   Therefore paths for leaf-labelled binary trees could be modelled in Haskell by

   ```
   data PathT a = EndpT (Either () a) |
                  ConspT1 (PathT a) | ConspT2 (PathT a)
   ```

   The fold operator for this type is

```
foldPT :: (Either () a -> b, b -> b, b -> b) -> PathT a -> b
foldPT (f,g,h) (EndpT a) = f a
foldPT (f,g,h) (ConspT1 x) = g (foldPT (f,g,h) x)
foldPT (f,g,h) (ConspT2 x) = h (foldPT (f,g,h) x)
```

For example, the 'length' function on this kind of path is

```
lengthT = foldPT phiLengthT
phiLengthT = (const 1, (1+), (1+))
```

2. For branch-labelled binary trees, we have

$$
\begin{aligned}
F(A,X) &= 1 + A \times X \times X \\
F_1(A,X) &= 1 \\
F_2(A,X) &= A \times X \times X \\
k_1 &= 0 \\
k_2 &= 2 \\
H(A,X) &= (1 + A \times 1 \times 1) + (A \times 1 \times 1 \times X + A \times 1 \times 1 \times X) \\
&\approx (1 + A) + A \times X + A \times X
\end{aligned}
$$

This could be modelled in Haskell by

```
data PathB a = EndpB (Either () a) |
               ConspB1 a (PathB a) | ConspB2 a (PathB a)
```

with a fold operator

```
foldPB :: (Either () a -> b, a->b->b, a->b->b) ->
          PathB a -> b
foldPB (f,g,h) (EndpB a) = f a
foldPB (f,g,h) (ConspB1 a x) = g a (foldPB (f,g,h) x)
foldPB (f,g,h) (ConspB2 a x) = h a (foldPB (f,g,h) x)
```

The corresponding length function is

```
lengthB = foldPB phiLengthB
phiLengthB = (const 1, inc, inc) where inc a n = 1+n
```

3. For funny trees, we have

$$
\begin{aligned}
F(A,X) &= \mathsf{Int} + A \times X + \mathsf{Bool} \times X \times X \\
F_1(A,X) &= \mathsf{Int} \\
F_2(A,X) &= A \times X \\
F_3(A,X) &= \mathsf{Bool} \times X \times X \\
k_1 &= 0 \\
k_2 &= 1 \\
k_3 &= 2 \\
H(A,X) &= (\mathsf{Int} + A \times 1 + \mathsf{Bool} \times 1 \times 1) + A \times 1 \times X + \\
&\qquad \mathsf{Bool} \times 1 \times 1 \times X + \mathsf{Bool} \times 1 \times 1 \times X \\
&\approx (\mathsf{Int} + A + \mathsf{Bool}) + A \times X + \mathsf{Bool} \times X + \mathsf{Bool} \times X
\end{aligned}
$$

This could be modelled in Haskell by

```
data PathF a = EndpF (Either3 Int a Bool) |
               ConspF1 a (PathF a) |
               ConspF2 Bool (PathF a) | ConspF3 Bool (PathF a)
```

with a fold operator

```
foldPF :: (Either3 Int a Bool -> b, a->b->b, Bool->b->b, Bool->b->b) ->
          PathF a -> b
foldPF (f,g,h,j) (EndpF a) = f a
foldPF (f,g,h,j) (ConspF1 a x) = g a (foldPF (f,g,h,j) x)
foldPF (f,g,h,j) (ConspF2 b x) = h b (foldPF (f,g,h,j) x)
foldPF (f,g,h,j) (ConspF3 b x) = j b (foldPF (f,g,h,j) x)
```

The corresponding length function is

```
lengthF = foldPF phiLengthF
phiLengthF = (const 1, inc, inc, inc) where inc a n = 1+n
```

(As with `sizeF`, what constitutes the 'right' definition of `lengthF` is a moot point. This definition counts the number of nodes in a path, regardless of whether or not those nodes carry a label of the type parameter `a`.)

□

6 Polytypic Downwards Accumulations

Having a polytypic notion of paths, we are now able to make a polytypic definition of downwards accumulations. The essential ingredient is a polytypic function paths_F, analogous to the function subs_F from Section 4; the difference is that paths_F labels every node of a data structure with its *ancestors*, a path, whereas subs_F labels every node with its *descendants*, another instance of the same datatype. Thus, the function paths_F has type $T(A) \to L(P(A))$.

The simplest characterization of paths_F is as a fold. The root of $\mathsf{paths}_F\, t$ is a trivial path to the root node of t. This is constructed by applying endp to something of type $F(A,1)$; the latter can be obtained by discarding the children after deconstructing t:

$$\mathsf{root}\,(\mathsf{paths}_F\, t) = \mathsf{endp}\,(F(\mathsf{id},\mathsf{one})\,(\mathsf{out}_F\, t))$$

The children of $\mathsf{paths}_F\, t$ are a little more tricky. We need to construct a value of type $F(1, L(P(A)))$. Suppose that t is of variant i, that is, $t = \mathsf{in}_i\, x$ where $x :: F_i(A, T(A))$. Now, x will contain k_i children, where k_i is the degree of branching of F_i. In order for paths_F to be a fold, the answer has to be composed from $F_i(\mathsf{id}, \mathsf{paths}_F)\, x$ somehow. However, each path in child j (where $1 \le j \le k_i$) must be prefixed with the root label of t and a record that from the root one must choose child j first; in other words, each path in child j should be subjected to $\mathsf{consp}_{i,j}\,a$ where $a = F_i(\mathsf{id},\mathsf{one})\,x$. Therefore, we have, with F_i' the chosen linearization of F_i,

$$\begin{aligned}&\mathsf{kids}\,(\mathsf{paths}_F\,(\mathsf{in}_i\,x))\\&\quad = \mathsf{inj}_i\,(F_i'(\mathsf{one}, L\,(\mathsf{consp}_{i,1}\,a),\ldots,L\,(\mathsf{consp}_{i,k_i}\,a))\,(F_i(\mathsf{id},\mathsf{paths}_F)\,x))\end{aligned}$$

where $a = F_i(\mathsf{id},\mathsf{one})\,x$. We therefore define paths_F as follows.

Definition 19.

$$\mathsf{paths}_F = \mathsf{fold}_F\,\phi$$

where

$$\begin{aligned}\phi\,(\mathsf{in}_i\,x) = \mathsf{in}_G\,(&\mathsf{endp}\,(F(\mathsf{id},\mathsf{one})\,(\mathsf{in}_i\,x)),\\&\mathsf{inj}_i\,(F_i'(\mathsf{one}, L\,(\mathsf{consp}_{i,1}\,a),\ldots,L\,(\mathsf{consp}_{i,k_i}\,a))\,x))\\&\text{where } a = F_i(\mathsf{id},\mathsf{one})\,x\end{aligned}$$

□

Unfortunately, the maps L in this characterization make it very inefficient. To get around this problem, we must define paths_F using unfold. However, we cannot simply define

$$\mathsf{paths}_F = \mathsf{unfold}_G\,\theta$$

for some suitable θ. Consider a substructure y of a data structure x. The substructure of $\mathsf{paths}_F\,x$ that corresponds to y is not constructed from y alone; it depends also on the ancestors of y in x. Therefore, we have to use an extra *accumulating parameter* [3] to carry this contextual information about the ancestors. In fact, we will define

$$\mathsf{paths}_F\,t = \mathsf{unfold}_G\,\theta\,(t,\mathsf{id})$$

where the extra accumulating parameter is a function of type $P(A) \to P(A)$. Intuitively, this accumulating parameter is mapped over the paths:

$$\mathsf{unfold}_G\,\theta\,(t,f) = L\,f\,(\mathsf{paths}_F\,t)$$

From this characterization it is possible to calculate the appropriate definition of θ.

Theorem 20.

$$\mathsf{paths}_F\,t = \mathsf{unfold}_G\,\theta\,(t,\mathsf{id})$$

where

$$\begin{aligned}\theta\,(\mathsf{in}_i\,x, f) = (f\,(\mathsf{endp}\,(\mathsf{inj}_i\,a)),\mathsf{inj}_i\,(F_i'(\mathsf{one},g_1,\ldots,g_{k_i})\,x))&\\ \text{where } a \;\;= F_i\,(\mathsf{id},\mathsf{one})\,x&\\ g_j\,y = (y, f\circ\mathsf{consp}_{i,j}\,a)&\end{aligned}$$

where k_i is the degree of branching of F_i. □

Example 21.

1. For leaf-labelled binary trees, we have

```
pathsT :: Ltree a -> Htree (PathT a)
pathsT t = unfoldH theta (t,id)
  where
    theta (LeafT a, f) = (f (EndpT (Right a)), Left ())
    theta (BinT t u, f) =
      (f (EndpT (Left ())), Right ((t,f.ConspT1), (u,f.ConspT2)))
```

2. For branch-labelled binary trees, we have

```
pathsB :: Btree a -> Htree (PathB a)
pathsB t = unfoldH theta (t,id)
  where
    theta (Empty, f) = (f (EndpB (Left ())), Left ())
    theta (BinB a t u, f) =
      (f (EndpB (Right a)), Right ((t,f.ConspB1 a),
                                   (u,f.ConspB2 a)))
```

3. For funny trees, we have

```
pathsF :: Ftree a -> Otree (PathF a)
pathsF t = unfoldO theta (t,id)
  where
    theta (TipF n, f) = (f (EndpF (In1 n)), In1 n)
    theta (MonF a x, f) =
      (f (EndpF (In2 a)), In2 (x,f.ConspF1 a))
    theta (BinF b x y, f) =
      (f (EndpF (In3 b)), In3 (b, (x,f.ConspF2 b),
                                  (y,f.ConspF3 b)))
```

□

Given paths_F, the downwards accumulation is straightforward to define: it is simply a path fold mapped over the paths:

$$\mathsf{da}_F\ \phi = L\,(\mathsf{fold}_H\ \phi) \circ \mathsf{paths}_F$$

In the next section we shall see that this definition is inefficient, and show how to make it efficient.

7 Polytypic Downwards Accumulations, Quickly

Consider the characterization of downwards accumulations from the previous section:

$$\mathsf{da}_F\ \phi = L\,(\mathsf{fold}_H\ \phi) \circ \mathsf{paths}_F$$

From type information we know that the ϕ here is of the form

$$f \triangledown \mathop{\bigtriangledown}_{i=1}^{n} \mathop{\bigtriangledown}_{j=1}^{k_i} \oplus_{i,j}$$

where $f :: F(A,1) \to B$ and $\oplus_{i,j} :: F_i(A,1) \times B \to B$; we introduce the shorthand $\langle f, \oplus \rangle$ for such a function.

This characterization is inefficient, because it takes no account of the fact that the paths to the children of a node x are closely related to the path to x itself. Rather, it computes the labels of a node and its children independently.

Recall that

$$\mathsf{unfold}_G\, \theta\, (t, f) = L\, f\, (\mathsf{paths}_F\, t)$$

Therefore we have

$$\begin{array}{ll} & \mathsf{da}_F\, \langle f, \oplus \rangle\, t \\ = & \quad \{\ \mathsf{da}\ \} \\ & L\, (\mathsf{fold}_H\, \langle f, \oplus \rangle)\, (\mathsf{paths}_F\, t) \\ = & \quad \{\ \text{equation above}\ \} \\ & \mathsf{unfold}_G\, \theta\, (t, \mathsf{fold}_H\, \langle f, \oplus \rangle) \end{array}$$

But this is no direct help, because the accumulating parameter in the unfold just builds up into a function of the form

$$\mathsf{fold}_H\, \langle f, \oplus \rangle \circ \mathsf{consp}_{i_1,j_1}\, a_1 \circ \cdots \circ \mathsf{consp}_{i_r,j_r}\, a_r$$

We do have

$$\begin{array}{ll} \mathsf{fold}_H\, \langle f, \oplus \rangle \circ \mathsf{endp} & = f \\ \mathsf{fold}_H\, \langle f, \oplus \rangle \circ \mathsf{consp}_{i,j}\, a & = (a \oplus_{i,j}) \circ \mathsf{fold}_H\, \langle f, \oplus \rangle \end{array}$$

so this big accumulating parameter can be simplified to

$$(a_1 \oplus_{i_1,j_1}) \circ \cdots \circ (a_r \oplus_{i_r,j_r}) \circ \mathsf{fold}_H\, \langle f, \oplus \rangle$$

However, in general, it simplifies no further. Thus, applying the accumulating parameter at a node at depth r takes at least r steps, and so the whole downwards accumulation is super-linear. (This is inherent in the definition of downwards accumulation, not an unfortunate artifact of our algorithm for computing it. For example, the function paths_F itself is a downwards accumulation, with $f = \mathsf{endp}$ and $\oplus = \mathsf{consp}$, and paths_F necessarily takes super-linear time to compute because it returns a tree with super-linearly many different subexpressions in it.)

The way around this problem is to assume an efficient representation of functions of the form $(a \oplus_{i,j})$ and of compositions of such functions.

Definition 22. The function $\langle f, \oplus \rangle$ is *efficiently representable* if there is a type C, a function $\mathsf{rep} :: (B \to B) \to C$, a function $\mathsf{abs} :: C \to (B \to B)$ such that

$$\mathsf{abs}\, (\mathsf{rep}\, (a \oplus_{i,j})) = (a \oplus_{i,j})$$

and a $\mathsf{zero} :: C$ and $\otimes :: C \to C \to C$ such that

$$\begin{array}{ll} \mathsf{abs}\, \mathsf{zero} & = \mathsf{id} \\ \mathsf{abs}\, (p \otimes q) & = \mathsf{abs}\, p \circ \mathsf{abs}\, q \end{array}$$

for which rep, abs and $\otimes$ can all be computed in constant time. $\square$

Example 23. For example, if $a \oplus_{i,j} n = g_{i,j}\, a + n$ for every i, j, and the $g_{i,j}$s can be computed in constant time, then $\langle f, \oplus \rangle$ is efficiently representable. The functions $(a \oplus_{i,j})$ and compositions of such functions are all of the form $(m+)$ for some m, and can be represented by the value m itself. Thus, $\mathsf{rep}\, g = g\, 0$, $\mathsf{abs}\, m = (m+)$, $\mathsf{zero} = 0$ and $\otimes = +$.

Downwards accumulations with efficiently representable parameters can be computed efficiently [20].

Theorem 24. Suppose $\langle f, \oplus \rangle$ is efficiently representable. Then

$$\mathsf{da}\, \langle f, \oplus \rangle\, t = \mathsf{unfold}_G\, \theta'\, (t, \mathsf{zero})$$

where

$$\begin{aligned} \theta'\, (\mathsf{in}_i\, x, p) = (\mathsf{abs}\, p\, (f\, (\mathsf{inj}_i\, a)), \mathsf{inj}_i\, (F'_i(\mathsf{one}, g_1, \ldots, g_k)\, x)) \\ \text{where } a \quad = F_i\, (\mathsf{id}, \mathsf{one})\, x \\ g_j\, y = (y, p \otimes \mathsf{rep}\, (a \oplus_{i,j})) \end{aligned}$$

This takes linear time, assuming f and $\oplus$ both take constant time. □

Proof. It is straightforward but tedious to calculate the above definition of θ' from the specification

$$\mathsf{unfold}_G\, \theta'\, (t, p) = \mathsf{unfold}_G\, \theta\, (t, \mathsf{abs}\, p \circ \mathsf{fold}_H\, \langle f, \oplus \rangle)$$

□

Example 25. We assume throughout these examples that `rep :: (b->b)->c`, `abs :: c->(b->b)` and `zero :: c`. To represent the operator $\otimes$ we use the Haskell identifier '`.:. :: c -> c -> c`'.

1. For leaf-labelled binary trees, we have

```
daT :: (Either () a -> b, b->b, b->b) -> Ltree a -> Htree b
daT (f,g,h) t = unfoldH theta' (t,zero)
  where
    theta' (LeafT a, p) = (abs p (f (Right a)), Left ())
    theta' (BinT t u, p)
      = (abs p (f (Left ())),
          Right ((t,p .:. rep g), (u,p .:. rep h)))
```

   For example, the function `depthsT`, which labels every node in the tree with its depth, is a downwards accumulation:

```
depthsT :: Ltree a -> Htree Int
depthsT = daT phiLengthT
```

   where `phiLengthT` is as defined in Example 18.

2. For branch-labelled binary trees, we have

```
daB :: (Either () a -> b, a->b->b, a->b->b) -> Btree a -> Htree b
daB (f,g,h) t = unfoldH theta' (t,zero)
  where
    theta' (Empty, p) = (abs p (f (Left ())), Left ())
    theta' (BinB a t u, p)
      = (abs p (f (Right a)),
         Right ((t,p .:. rep (g a)), (u,p .:. rep (h a))))
```

The corresponding depths function is

```
depthsB :: Btree a -> Htree Int
depthsB = daB phiLengthB
```

3. For funny trees, we have

```
daF :: (Either3 Int a Bool -> b, a->b->b, Bool->b->b, Bool->b->b) ->
       Ftree a -> Otree b
daF (f,g,h,j) t = unfoldO theta' (t,zero)
  where
    theta' (TipF n, p) = (abs p (f (In1 n)), In1 n)
    theta' (MonF a x, p) =
      (abs p (f (In2 a)), In2 (x,p .:. rep (g a)))
    theta' (BinF b x y, p) =
      (abs p (f (In3 b)), In3 (b, (x,p .:. rep (h b)),
                                  (y,p .:. rep (j b))))
```

The corresponding depths function is

```
depthsF :: Ftree a -> Otree Int
depthsF = daF phiLengthF
```

□

8 Conclusion

We have shown how to generalize the notion of *downwards accumulation* [8, 9] to an arbitrary polynomial datatype, building on Bird *et al*'s [2] generalization of upwards accumulation.

Downwards accumulations are much harder to handle than upwards accumulations, because the *ancestors* of a node in a datatype (on which the downwards accumulation depends) form an instance of a completely different datatype, whereas the *descendants* of a node (on which the upwards accumulation depends) form an instance of the same datatype.

On the other hand, Bird *et al*'s construction is *parametric higher-order polymorphic*, whereas ours is merely *ad-hoc higher-order polymorphic*, because our definition is based on the syntactic presentation of the type functors involved; it is an open question whether one can construct a parametric higher-order polymorphic definition of downwards accumulations. In fact, our construction is not even fully polytypic in the sense of Jeuring, in that we deal only with *polynomial* datatypes, whereas Jeuring deals with the more general *regular* datatypes. Another open question is whether our construction can be extended to regular but non-polynomial datatypes.

We have addressed the question of efficiency in computing downwards accumulations by assuming a particular data refinement of the operations concerned. This data refinement allows the order of evaluation of a path fold to be inverted [6, 20], so that it is computed from top to bottom instead of from bottom to top. This is a different approach to the one taken in the original definition of downwards accumulation [8, 9]. There, the order of evaluation is reversed by switching to a dual path datatype—informally, if the path datatypes we construct here are variations on cons lists, then the switch is to variations on snoc lists. This works neatly for the simple monotypic cases considered in the original definitions, but it is rather more complicated in the general case. The reason is that a snoc-style path ending at a node of variant i can be extended downwards only by a 'direction' corresponding to variant i (for example, a path ending at an external node of the data structure clearly cannot be extended at all). In contrast, a cons-style path can be extended upwards in any direction.

Nevertheless, we intend to explore the idea of snoc-style paths further. One motivation for doing so is to see whether they yield a simpler definition of downwards accumulations. A more important reason, however, is the hope of generalizing the *Third Homomorphism Theorem* [12] to our polytypic paths. The Third Homomorphism Theorem, first stated by Bird and proved by Meertens, says that any list function that can be computed both from left to right and from right to left can also be computed as an associative fold; this is much better suited to parallel computation. We have shown [10] that this leads to a very fast parallel algorithm for computing (the original kind of) downwards accumulations. We hope that it will also lead to a fast parallel algorithm for computing polytypic downwards accumulations.

Acknowledgements

I would like to thank the members of the Oxford University Computing Laboratory, especially Richard Bird, Jesús Ravelo and Pedro Borges, for their many helpful suggestions and comments; the MPC referees have also suggested many improvements.

References

1. Roland Backhouse, Henk Doornbos, and Paul Hoogendijk. A class of commuting relators. In *STOP 1992 Summerschool on Constructive Algorithmics*. STOP project, 1992.
2. Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

3. 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 [4].
4. 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.
5. Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.
6. E. A. Boiten. Improving recursive functions by inverting the order of evaluation. *Science of Computer Programming*, 18:139–179, January 1992. Also in [7].
7. Eerke Boiten. *Views of Formal Program Development*. PhD thesis, Department of Informatics, University of Nijmegen, 1992.
8. Jeremy Gibbons. *Algebras for Tree Algorithms*. D. Phil. thesis, Programming Research Group, Oxford University, 1991. Available as Technical Monograph PRG-94.
9. Jeremy Gibbons. Upwards and downwards accumulations on trees. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *LNCS 669: Mathematics of Program Construction*, pages 122–138. Springer-Verlag, 1993. A revised version appears in the Proceedings of the Massey Functional Programming Workshop, 1992.
10. Jeremy Gibbons. Computing downwards accumulations on trees quickly. *Theoretical Computer Science*, 169(1):67–80, 1996. Earlier version appeared in Proceedings of the 16th Australian Computer Science Conference, Brisbane, 1993.
11. 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.
12. Jeremy Gibbons. The Third Homomorphism Theorem. *Journal of Functional Programming*, 6(4):657–665, 1996. Earlier version appeared in C. B. Jay, editor, *Computing: The Australian Theory Seminar*, Sydney, December 1994, p. 62–69.
13. Jeremy Gibbons and Geraint Jones. Against the grain: Linear-time breadth-first tree algorithms. Oxford Brookes University and Oxford University Computing Laboratory, 1998.
14. Paul Hoogendijk. *A Generic Theory of Datatypes*. PhD thesis, TU Eindhoven, 1997.
15. Johan Jeuring and Patrick Jansson. Polytypic programming. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *LNCS 1129: Advanced Functional Programming*. Springer-Verlag, 1996.
16. Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
17. Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992. Also available as Technical Report CS-R9005, CWI, Amsterdam.
18. David B. Skillicorn. Structured parallel computation in structured documents. *Journal of Universal Computer Science*, 3(1), 1997.
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.