Generic Downwards Accumulations

Jeremy Gibbons

School of Computing and Mathematical Sciences,
Oxford Brookes University,
Gipsy Lane, Headington,
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 regular datatype; the resulting definition is co-inductive.

1 Introduction

The notion of *scans* or *accumulations* on lists is well known, and has proved very fruitful for expressing and calculating with programs involving lists [4]. Gibbons [7, 8] generalizes 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 [15, 8], Reingold and Tilford's algorithm for drawing trees tidily [21, 9], and algorithms for query evaluation in structured text [16, 23].

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 [7, 8] is that they are 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

 This paper is a revised version of *Polytypic Downwards Accumulations*, in *LNCS 1422: Mathematics of Program Construction*, ed. Johan Jeuring, 1998.

of *generic* [12,13] or *polytypic* [14] 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 [17] is an early example of generic programming: it allows a single unified definition of operations such as map, fold and unfold, parameterized by the datatype concerned.

Bird *et al.* [1] generalize upwards accumulation to an arbitrary regular datatype, unifying the previous ad hoc definitions. In this paper, we generalize downwards accumulation to an arbitrary regular 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').

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 accumulations on lists and trees. We briefly summarize the theory of datatypes in Section 3. In Section 4 we discuss Bird *et al.*'s generic definition of upwards accumulation. In Section 5 we develop a generic definition of downwards accumulation, and in Section 6 we collect some properties of generic downwards accumulations. Section 7 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, denoted $()$; there is a unique total function $\mathsf{one} :: A \to 1$ for every type A. The function $\mathsf{const} :: A \to B \to A$ ignores its second argument and always returns its first, that is, it satisfies $\mathsf{const}\, a\, b = a$ for all a and b.

1.2 The Pair Calculus

The functors $+$ and $\times$ denote separated sum and cartesian product respectively; $\times$ binds tighter than $+$. The product projections are fst and snd, and the product morphism (sometimes called 'split' or 'fork') $f \vartriangle g$ has type $A \to B \times C$ when $f :: A \to B$ and $g :: A \to C$.

1.3 Lists

The elements of the type $List(A)$ are finite sequences of elements of type A. The empty list is denoted $[\,]$, singleton lists $[a]$, and $+\!\!+$ is list concatenation.

2 Accumulations

To provide motivation and intuition for what follows, we review here the 'monotypic' definitions of leftwards and rightwards accumulations on lists (well-known from functional programming) and upwards and downwards accumulations on binary trees (as presented in [7, 8]). Bird *et al.*'s generalization of upwards accumulation and our generalization of downwards accumulation, when specialized to the same types, reduce essentially to these monotypic definitions.

2.1 Accumulations on Lists

The standard leftwards and rightwards accumulations on lists are defined as follows. Leftwards accumulation distributes information from right to left along a list. It is traditionally, if confusingly, called `scanr`, perhaps because the parentheses collect at the right-hand end of the list.

```
scanr :: (a->b->b) -> b -> List a -> List b
scanr f e []    = [e]
scanr f e (a:x) = f a (head x') : x'   where x' = scanr f e x
```

(Here, '`a:x`' is the non-empty list with head `a` and tail `x`, and the function `head` satisfies `head (a:x) = a`. The notation is essentially that of Haskell [20].) For example,

```
scanr (+) 0 [1,2,3] = [6,5,3,0]
```

(Here, '`(+)`' is the addition operator passed as an argument, and '`[1,2,3]`' a list with three elements.)

Rightwards accumulation, of course, distributes information in the opposite direction:

```
scanl :: (b->a->b) -> b -> List a -> List b
scanl f e []    = [e]
scanl f e (a:x) = e : scanl f (f e a) x
```

For example,

```
scanl (+) 0 [1,2,3] = [0,1,3,6]
```

To make the analogy with trees clearer, we will first adapt the standard list accumulations to operate on a type of non-empty lists:

```
data Plist a = Wrap a | ConsP a (Plist a)
```

Informally, a non-empty list is either a singleton, `Wrap a`, or an element prefixed onto another non-empty list, `ConsP a x`. On this type, the accumulations return a list of the same length as the argument, instead of a list one element longer. As will become clear, leftwards accumulation on lists is a kind of upwards accumulation, so we rename the function to `uaPlist`:

```
uaPlist :: (a->b) -> (a->b->b) -> Plist a -> Plist b
uaPlist f g (Wrap a)    = Wrap (f a)
uaPlist f g (ConsP a x) = ConsP (g a (headPlist x')) x'
                          where x' = uaPlist f g x
```

where `headPlist` returns the first element of a non-empty list:

```
headPlist :: Plist a -> a
headPlist (Wrap a)    = a
headPlist (ConsP a x) = a
```

For example,

```
uaPlist id (+) [1,2,3] = [6,5,3]
```

Rightwards accumulation (which we now call `daPlist`, for 'downwards accumulation on a list') is generalized with an extra parameter, so that it is not the accumulating parameter `e` itself that appears in the result, but some function `f e a` of `e` and the label `a` at this node. The accumulating parameter for the next node is `g e a`, another function of the accumulating parameter and the label at this node.

```
daPlist :: (c->a->b) -> (c->a->c) -> c -> Plist a -> Plist b
daPlist f g e (Wrap a)    = Wrap (f e a)
daPlist f g e (ConsP a x) = ConsP (f e a) (daPlist f g (g e a) x)
```

For example,

```
daPlist (+) (+) 0 [1,2,3] = [1,3,6]
```

A more complicated example, making use of the generalization allowing the accumulating parameter to be of a different type than the result labels, is to compute the 'running averages' of a list:

```
averages = daPlist f g (0,0) where f (s,n) a = (s+a)/(n+1)
                                   g (s,n) a = (s+a , n+1)
```

2.2 Accumulations on Binary Trees

The elements of the datatype of *homogeneous binary trees* are trees with internal and external labels of the same type:

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

Thus, a tree is either a `Leaf` with a label, or a `Bin` with a label and two subtrees.

Upwards accumulation on these trees is fairly straightforward:

```
uaTree :: (a->b) -> (a->b->b->b) -> Tree a -> Tree b
uaTree f g (Leaf a)    = Leaf (f a)
uaTree f g (Bin a t u) = Bin (g a (root t') (root u')) t' u'
                         where t' = uaTree f g t
                               u' = uaTree f g u
```

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

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

The archetypical example of an upwards accumulation is to count the descendants of every node:

```
sizes :: Tree a -> Tree Int
sizes = uaTree one plus  where one a      = 1
                               plus a m n = 1+m+n
```

Downwards accumulation is a little trickier: we need two functions for generating new accumulating parameters, so that left children can be treated differently from right children.

```
daTree :: (c->a->b) -> (c->a->c) -> (c->a->c) -> c -> Tree a -> Tree b
daTree f g h e (Leaf a)    = Leaf (f e a)
daTree f g h e (Bin a t u) = Bin (f e a) (daTree f g h (g e a) t)
                                         (daTree f g h (h e a) u)
```

The archetypical example of a downwards accumulation is to count the ancestors of every node:

```
depths :: Tree a -> Tree Int
depths = daTree label next next 0  where label e a = e+1
                                         next  e a = e+1
```

3 Datatypes

In this section, we briefly review the construction of regular datatypes, in the style of Malcolm [17]. However, we assume a setting of continuous functions between pointed complete partial orders, in contrast to Malcolm's setting of total functions between sets. 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 $\mathsf{out}_F :: T(A) \to F(A, T(A))$, both of which are strict. (Fokkinga and Meijer [6], building on the work of Reynolds [22], show that this determines T up to isomorphism in terms of F.) We call $T(A)$ the *canonical fixpoint* of the functor $F(A, \_)$.

A strict function $\phi :: F(A, B) \to B$ induces a fold (or 'catamorphism') $\mathsf{fold}_F\, \phi :: T(A) \to B$, with the universal property

$$h = \mathsf{fold}_F\, \phi \;\Leftrightarrow\; h \circ \mathsf{in}_F = \phi \circ F(\mathsf{id}, h)$$

A (not necessarily strict) function $\psi :: B \to F(A, B)$ induces an unfold (or 'anamorphism') $\mathsf{unfold}_F\, \psi :: B \to T(A)$, with the universal property

$$h = \mathsf{unfold}_G\, \psi \;\Leftrightarrow\; \mathsf{out}_G \circ h = G(\mathsf{id}, h) \circ \psi$$

The function $\mathsf{map}_F\, f :: T(A) \to T(B)$ for $f :: A \to B$ applies f to every element of its argument; in other words, $\mathsf{map}_F\, f$ is the action $T(f)$ of the functor T on f. It can be defined either as a fold or as an unfold:

$$\begin{aligned} \mathsf{map}_F\, f &= \mathsf{fold}_F\, (\mathsf{in}_F \circ F(f, \mathsf{id})) \\ &= \mathsf{unfold}_F\, (F(f, \mathsf{id}) \circ \mathsf{out}_F) \end{aligned}$$

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

1. Cons lists are constructed from the functor $F(A, X) = 1 + A \times X$:

   ```
   data List a  =  Nil | Cons a (List a)
   ```

   The corresponding fold is

   ```
   foldL :: (Either () (a,b) -> b) -> List a -> b
   foldL phi Nil        = phi (Left ())
   foldL phi (Cons a x) = phi (Right (a, foldL phi x))
   ```

   (Here, the Haskell datatype `Either` is defined by

   ```
   data Either a b = Left a | Right b
   ```

   and corresponds roughly to disjoint sum.) For example, the function `sizeL`, which returns the size (length) of a list, is a fold:

   ```
   sizeL :: List a -> Int
   sizeL = foldL phiSizeL
   phiSizeL (Left ())     = 0
   phiSizeL (Right (a,n)) = 1+n
   ```

   (We will use `phiSizeL` later.)
2. 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))
   ```

   The function `sizeT`, which returns the size of (that is, the number of elements in) 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
   ```

3. Internally-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
```

4. Rose trees [18] are constructed from the functor $F(A, X) = A \times \mathit{List}(X)$:

```
data Rtree a = ConsR a (List (Rtree a))
```

A tree of type `Rtree a` consists of a label of type `a` and a list of children. The corresponding fold is

```
foldR :: ((a, List b) -> b) -> Rtree a -> b
foldR phi (ConsR a ts) = phi (a, map (foldR phi) ts)
```

where `map f xs` applies function `f` to every element of list `xs`. The function `sizeR` is defined as follows:

```
sizeR :: Rtree a -> Int
sizeR = foldR phiSizeR
phiSizeR :: (a, List Int) -> Int
phiSizeR (a,ns) = 1 + sum ns
```

where `sum` sums a list of integers.

□

4 Generic Upwards Accumulations

Bird *et al.* [1] generalize upwards accumulation to an arbitrary regular datatype. We summarize their construction here. It is related to, but not the same as, Meertens' generic definition of the 'predecessors' of a data structure [19]. This section serves partly as motivation and to provide intuition for the definitions in Section 5, as upwards accumulations are simpler than downwards accumulations. More importantly, however, we will use part of their construction in our own. Throughout this section, we assume that F is a bifunctor, and that $T(A)$ is the canonical fixpoint of $F(A, \_)$.

4.1 Labelled Types

The essential idea is that an upwards accumulation of a data structure x computes all the partial results involved in folding x. Each partial result is the fold of some subtree of x, and is stored in the resulting data structure at the root of that subtree.

A consequence of taking this approach is that the result type of an upwards accumulation may be different from the argument type: the result type has a label at every node, whereas the argument type need not. For example, an upwards accumulation on a leaf-labelled binary tree should produce a homogeneous binary tree. Bird *et al.* call the datatype of homogeneous binary trees the *labelled variant* of the type of leaf-labelled binary trees, and give the following general construction for it.

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

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 cons lists is constructed from the functor $F(A, X) = 1 + A \times X$, so the functor G is given by

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

   This induces a labelled type of non-empty lists:

```
data Nlist a  =  ConsN a (Either () (Nlist a))
```

   This type is similar to the datatype `Plist` from Section 2. The unfold for this type is

```
unfoldN :: (b -> (a, Either () b)) -> b -> Nlist a
unfoldN phi b = case phi b of
  (a, Left ())  -> ConsN a (Left ())
  (a, Right b') -> ConsN a (Right (unfoldN phi b'))
```

   (We give folds for the 'ordinary' datatypes and unfolds for the labelled variants, because an accumulation consumes an ordinary datatype and produces a labelled one.)
2. 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 type is similar to the datatype `Tree` of Section 2.) The corresponding unfold is

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

3. The datatype of internally-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.
4. 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

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

and so homogeneous binary trees are their own labelled type. We call a type that is (isomorphic to) its own labelled variant *homogeneous*, and a non-homogeneous type *heterogeneous*. In general, if $F(A,X) = A \times F'(X)$ for some F' independent of A, then $G(A,X) = A \times F(1,X) = A \times 1 \times F'(X) \approx F(A,X)$, and so the generated type is homogeneous. In particular, 'constructing the labelled variant' is an idempotent operation.
5. The datatype of rose trees is constructed from the functor $F(A,X) = A \times \mathit{List}(X)$, so the functor G is given by

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

and so rose trees are homogeneous. The corresponding unfold is

```
unfoldR :: (b -> (a, List b)) -> b -> Rtree a
unfoldR phi b = case phi b of
  (a,bs) -> ConsR a (map (unfoldR phi) bs)
```

(The purpose of these and subsequent examples is to present definitions that the rational programmer might have come up with after some thought, and to show that those definitions are instantiations of our constructions. We use the term 'instantiation' a little loosely: literal instantiations often involve extra superfluous units, which we have suppressed. For example, the literal use of the characterization $G(A,X) = A \times F(1,X)$ gives for cons lists the instantiation $G(A,X) = A \times (1 + 1 \times X)$ and the corresponding type declaration

```
data Nlist a = ConsN a (Either () ((),Nlist a))
```

but in Example 3.1 we suppressed the superfluous `()`.) □

4.2 Upwards Accumulations

The upwards accumulation $\mathsf{ua}_F\,\phi$ constructs a labelled data structure of type $L(B)$ from a data structure of type $T(A)$; the argument ϕ is a suitable argument for fold_F, and the data structure returned by $\mathsf{ua}_F\,\phi\,x$ contains all the partial results obtained in the process of computing $\mathsf{fold}_F\,\phi\,x$.

We will define $\mathsf{ua}_F\,\phi$ as an unfold, namely $\mathsf{ua}_F\,\phi = \mathsf{unfold}_G\,\psi$ for some ψ dependent on ϕ. Type considerations reveal that

$$\begin{aligned}\psi &:: T(A) \to G(B, T(A)) \\ &:: T(A) \to B \times F(1, T(A))\end{aligned}$$

Such a ψ is necessarily of the form $\psi_1 \vartriangle \psi_2$ for some $\psi_1 :: T(A) \to B$ and $\psi_2 :: T(A) \to F(1, T(A))$. We let ψ_1 be simply the corresponding fold:

$$\psi_1 = \mathsf{fold}_F\,\phi$$

and define ψ_2 to destruct the node and discard the root labels:

$$\psi_2 = F(\mathsf{one}, \mathsf{id}) \circ \mathsf{out}_F$$

Thus, we have the following definition.

Definition 4. Upwards accumulation $\mathsf{ua}_F :: (F(A, B) \to B) \to T(A) \to L(B)$ is defined by

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

□

This definition is illustrated in Figure 1. The dotted arrow is the product morphism induced by the two independent parts of the diagram.

$$\begin{tikzcd}
& T(A) \arrow[r, "\mathsf{out}_F"] \arrow[dl, "\mathsf{fold}_F\,\phi"'] \arrow[d, dotted, "\mathsf{fold}_F\,\phi \vartriangle F(\mathsf{one},\mathsf{id}) \circ \mathsf{out}_F"] & F(A, T(A)) \arrow[d, "F(\mathsf{one},\mathsf{id})"] \\
B & B \times F(1, T(A)) \arrow[l, "\mathsf{fst}"'] \arrow[r, "\mathsf{snd}"] \arrow[d, phantom, "="] & F(1, T(A)) \\
& G(B, T(A)) &
\end{tikzcd}$$

Fig. 1. The anatomy of an upwards accumulation

Example 5.

1. The labelled variant of cons lists is non-empty cons lists, so `uaL` is defined in terms of `unfoldN`:

```
uaL :: (Either () (a,b) -> b) -> List a -> Nlist b
uaL phi = unfoldN psi
  where psi Nil        = (foldL phi Nil, Left ())
        psi (Cons a x) = (foldL phi (Cons a x), Right x)
```

For example, the function `sizesL` labels every node of a list with the length of the tail segment starting at that node (returning a list of the form `[n,...,0]`):

```
sizesL :: List a -> Nlist Int
sizesL = uaL phiSizeL
```

where `phiSizeL` is as defined in Example 1.1.

2. The labelled variant of leaf-labelled binary trees is homogeneous binary trees, so we use `unfoldH`:

```
uaT :: (Either a (b,b) -> b) -> Ltree a -> Htree b
uaT phi = unfoldH psi
  where psi (LeafT a)  = (foldT phi (LeafT a), Left ())
        psi (BinT t u) = (foldT phi (BinT t 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
```

3. For internally-labelled binary trees too, the labelled variant is homogeneous binary trees:

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

The corresponding 'sizes' function on these trees is

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

4. Rose trees are homogeneous, so we use `unfoldR`:

```
uaR :: ((a, List b) -> b) -> Rtree a -> Rtree b
uaR phi = unfoldR psi
  where psi (ConsR a ts) = (foldR phi (ConsR a ts), ts)
```

The corresponding 'sizes' function is

```
sizesR :: Rtree a -> Rtree Int
sizesR = uaR phiSizeR
```

□

Of course, taken literally, Definition 4 makes rather an inefficient program: the substructures rooted at each node are folded independently, without exploiting the fact that the results of folding the children of a node can be reused in folding the substructure rooted at the node itself. Fortunately, an immediate consequence of the definition is that the root of the data structure returned by an upwards accumulation is the fold of the original data structure, and it is a straightforward matter to calculate the more efficient characterization described in Theorem 8.

Definition 6. The function $\mathsf{root}_G :: L(A) \to A$ returns the root of a homogeneous data structure:

$$\mathsf{root}_G = \mathsf{fst} \circ \mathsf{out}_G$$

□

Lemma 7.

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

□

Proof. The proof depends on the universal property

$$h = \mathsf{unfold}_G\ \phi \;\Leftrightarrow\; \mathsf{out}_G \circ h = G(\mathsf{id}, h) \circ \phi$$

of unfolds. We have

$$\begin{array}{ll}
& \mathsf{root}_G \circ \mathsf{ua}_F\ \phi \\
= & \quad \{\ \mathsf{root}\ \} \\
& \mathsf{fst} \circ \mathsf{out}_G \circ \mathsf{ua}_F\ \phi \\
= & \quad \{\ \text{universal property of } \mathsf{unfold}\ \} \\
& \mathsf{fst} \circ G(\mathsf{id}, \mathsf{ua}_F\ \phi) \circ (\mathsf{fold}_F\ \phi \mathbin{\triangle} (F(\mathsf{one}, \mathsf{id}) \circ \mathsf{out}_F)) \\
= & \quad \{\ G(f, g) = f \times F(\mathsf{id}, g)\ \} \\
& \mathsf{fst} \circ (\mathsf{id} \times F(\mathsf{id}, \mathsf{ua}_F\ \phi)) \circ (\mathsf{fold}_F\ \phi \mathbin{\triangle} (F(\mathsf{one}, \mathsf{id}) \circ \mathsf{out}_F)) \\
= & \quad \{\ \text{pairs}\ \} \\
& \mathsf{fold}_F\ \phi
\end{array}$$

□

Theorem 8.

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

□

Proof. As well as properties of unfold, this proof depends also on the universal property

$$h = \mathsf{fold}_F\ \phi \;\Leftrightarrow\; h \circ \mathsf{in}_F = \phi \circ F(\mathsf{id}, h)$$

of folds; in particular,

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

Using this property, we can calculate the appropriate ψ:

$$
\begin{aligned}
& \mathsf{ua}_F\ \phi \circ \mathsf{in}_F \\
= & \quad \{\ \text{isomorphism of datatypes: } \mathsf{in}_G \circ \mathsf{out}_G = \mathsf{id}\ \} \\
& \mathsf{in}_G \circ \mathsf{out}_G \circ \mathsf{ua}_F\ \phi \circ \mathsf{in}_F \\
= & \quad \{\ \mathsf{ua} \text{ as an unfold}\ \} \\
& \mathsf{in}_G \circ G(\mathsf{id}, \mathsf{ua}_F\ \phi) \circ (\mathsf{fold}_F\ \phi \vartriangle (F(\mathsf{one}, \mathsf{id}) \circ \mathsf{out}_F)) \circ \mathsf{in}_F \\
= & \quad \{\ G(f,g) = f \times F(\mathsf{id}, g);\ \text{pairs}\ \} \\
& \mathsf{in}_G \circ (\mathsf{fold}_F\ \phi \vartriangle (F(\mathsf{one}, \mathsf{ua}_F\ \phi) \circ \mathsf{out}_F)) \circ \mathsf{in}_F \\
= & \quad \{\ \text{composition distributes backwards over fork;} \\
& \qquad \text{isomorphism of datatypes again}\ \} \\
& \mathsf{in}_G \circ ((\mathsf{fold}_F\ \phi \circ \mathsf{in}_F) \vartriangle F(\mathsf{one}, \mathsf{ua}_F\ \phi)) \\
= & \quad \{\ \text{fold}\ \} \\
& \mathsf{in}_G \circ ((\phi \circ F(\mathsf{id}, \mathsf{fold}_F\ \phi)) \vartriangle F(\mathsf{one}, \mathsf{ua}_F\ \phi)) \\
= & \quad \{\ \text{Lemma 7}\ \} \\
& \mathsf{in}_G \circ ((\phi \circ F(\mathsf{id}, \mathsf{root}_G \circ \mathsf{ua}_F\ \phi)) \vartriangle F(\mathsf{one}, \mathsf{ua}_F\ \phi)) \\
= & \quad \{\ \text{composition distributes backwards over fork}\ \} \\
& \mathsf{in}_G \circ ((\phi \circ F(\mathsf{id}, \mathsf{root}_G)) \vartriangle F(\mathsf{one}, \mathsf{id})) \circ F(\mathsf{id}, \mathsf{ua}_F\ \phi)
\end{aligned}
$$

Therefore

$$
\begin{aligned}
& \mathsf{ua}_F\ \phi = \mathsf{fold}_F\ \psi \\
\Leftarrow & \\
& \psi = \mathsf{in}_G \circ (\phi \circ F(\mathsf{id}, \mathsf{root}_G) \vartriangle F(\mathsf{one}, \mathsf{id}))
\end{aligned}
$$

□

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 9.

1. For cons lists, we have

```
uaL :: (Either () (a,b) -> b) -> List a -> Nlist b
uaL phi = foldL psi
  where
    psi (Left ()) = ConsN (phi (Left ())) (Left ())
    psi (Right (a,x))
                  = ConsN (phi (Right (a, rootN x))) (Right x)
```

   Here, `rootN` returns the 'root' (the first element) of a non-empty list:

```
rootN :: Nlist a -> a
rootN (ConsN a x) = a
```

2. 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))
```

Here, `rootH` returns the root label of a homogeneous binary tree:

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

3. For internally-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))
```

4. For rose trees we have

```
uaR :: ((a, List b) -> b) -> Rtree a -> Rtree b
uaR phi = foldR psi
  where
    psi (a,us) = ConsR (phi (a, map rootR us)) us
```

Here, `rootR` returns the root label of a rose tree:

```
rootR :: Rtree a -> a
rootR (ConsR a ts) = a
```

□

5 Generic Downwards Accumulations

Just as with upwards accumulations, a downwards accumulation on a data structure returns a data structure of a possibly different type, namely the labelled variant of the original datatype. A downwards accumulation differs from an upwards accumulation in that it takes an extra *accumulating parameter* [2] with which to carry contextual information about its ancestors into a subtree. In this section, we will generalize the function `daTree` from Section 2 to a generic downwards accumulation for an arbitrary regular datatype.

Throughout this section, we assume that F is a bifunctor, T is its induced datatype, and L is the labelled variant (induced by bifunctor G).

5.1 The Essential Ideas

We will define a function $\mathsf{da}_F\,\phi$ of type $C \times T(A) \to L(B)$. Here, ϕ is the operation characterizing the particular accumulation, and C is the type of the accumulating parameter; the result is a labelled data structure with labels of type B.

We will write $\mathsf{da}_F\,\phi$ as an unfold, $\mathsf{unfold}_G\,\psi$ for some ψ dependent on ϕ. Type considerations reveal that ψ therefore has type

$$\psi :: C \times T(A) \to G(B, C \times T(A))$$

The only question now is how to define ψ. We will construct separately from a value of type $C \times T(A)$ values of type $G(B, C)$ and $G(1, T(A))$, and combine these two values into a single value of type $G(B, C \times T(A))$.

In order to combine the two values, we suppose a function

$$\mathsf{zip}_G :: G(A, B) \times G(C, D) \to G(A \times C, B \times D)$$

This function will be partial, defined only when both arguments have the same 'shape'. In particular, when G constructs a sum type, these two arguments should be in the same choices of the sum. Zipping together $G(B, C)$ and $G(1, T(A))$ gives $G(B\times 1, C\times T(A))$; discarding the extra 1 with $G(\mathsf{fst}, \mathsf{id})$ yields the required $G(B, C \times T(A))$, the result of ψ.

The second value is the easier to construct. Applying out_F to the second component of the input of type $C \times T(A)$ yields a value of type $C \times F(A, T(A))$. Discarding the C and the A by applying $\mathsf{one} \times F(\mathsf{one}, \mathsf{id})$ produces the required type $1 \times F(1, T(A))$, or equivalently $G(1, T(A))$.

For the first value, we keep the root labels and discard the subtrees from the second component of the argument of type $C \times T(A)$, by applying out_F and then $F(\mathsf{id}, \mathsf{one})$ to the second component. This produces values of type $C\times F(A, T(A))$ and then $C \times F(A, 1)$. Now we suppose that the argument ϕ to the downwards accumulation can complete the job:

$$\begin{aligned} \phi &:: C \times F(A, 1) \to B \times F(1, C) \\ &:: C \times F(A, 1) \to G(B, C) \end{aligned}$$

Thus ϕ takes a pair as an argument and returns a pair as a result. The first component of the argument, of type C, is the accumulating parameter. The second component of the argument, of type $F(A, 1)$, is the data structure with the recursive components stripped away, leaving only whatever labels are attached to this node. The first component of the result, of type B, is the label generated for this node of the result. The second component, of type $F(1, C)$, contains the accumulating parameters for the recursive calls, one for each subcomponent. There is an extra requirement, that the second component of the result have the same 'shape' as the second component of the argument, in order that the subsequent zip can combine the two.

Figure 2 illustrates the process. The dotted arrow is the product morphism induced by the two independent parts of the diagram.

5.2 Shape Preservation

The function ϕ used in a downwards accumulation has to be well-behaved with respect to shape, in a sense that we make precise here. Intuitively, ϕ must produce an appropriate collection of new accumulating parameters for the recursive calls, one accumulating parameter per call. In particular, when F yields a sum type, the accumulating parameters produced, of type $F(1, C)$, must be in the same variant of F as the argument, of type $F(A, 1)$, consumed.

$$\begin{tikzcd}
 & C \times T(A) \arrow[d, "\mathsf{id} \times \mathsf{out}_F"'] & \\
C \times F(A,1) \arrow[d, "\phi"'] & C \times F(A, T(A)) \arrow[l, "\mathsf{id} \times F(\mathsf{id}, \mathsf{one})"'] \arrow[dd, dotted, "{\begin{array}{c}\phi \circ (\mathsf{id} \times F(\mathsf{id}, \mathsf{one})) \\ \triangle \\ \mathsf{one} \times F(\mathsf{one}, \mathsf{id})\end{array}}"'] \arrow[dr, "\mathsf{one} \times F(\mathsf{one}, \mathsf{id})"] & \\
B \times F(1, C) \arrow[d, equal] & & 1 \times F(1, T(A)) \arrow[d, equal] \\
G(B, C) & G(B, C) \times G(1, T(A)) \arrow[l, "\mathsf{fst}"'] \arrow[r, "\mathsf{snd}"] \arrow[d, "\mathsf{zip}_G"'] & G(1, T(A)) \\
 & G(B \times 1, C \times T(A)) \arrow[d, "G(\mathsf{fst}, \mathsf{id})"'] & \\
 & G(B, C \times T(A)) &
\end{tikzcd}$$

Fig. 2. The anatomy of a downwards accumulation

We formalize this as follows. The *shape* of a data structure is obtained by discarding all the data in that data structure, retaining only structure. In particular, the shape of a value x of type $F(A, B)$ is $F(\mathsf{one}, \mathsf{one})\, x$, of type $F(1, 1)$.

Definition 10.

$$\mathsf{shape}_F = F(\mathsf{one}, \mathsf{one})$$

□

We require that if $\phi\,(c, x) = (b, y)$ then $\mathsf{shape}_F\, x = \mathsf{shape}_F\, y$.

5.3 Zipping Two Data Structures

One component of the construction outlined above is a function

$$\mathsf{zip}_G :: G(A, B) \times G(C, D) \to G(A \times C, B \times D)$$

This function should be purely structural, not relying on the values of type A, B, C and D. In other words, it should be polymorphic, or a *natural transformation* [24], satisfying the free theorem

$$G(f \times g, h \times k) \circ \mathsf{zip}_G = \mathsf{zip}_G \circ (G(f, h) \times G(g, k))$$

A function in the opposite direction is easy to define:

$$\begin{aligned}\mathsf{unzip}_G &:: G(A \times C, B \times D) \to G(A,B) \times G(C,D)\\ \mathsf{unzip}_G &= G(\mathsf{fst},\mathsf{fst}) \vartriangle G(\mathsf{snd},\mathsf{snd})\end{aligned}$$

Another requirement of zip is that it should be a post-inverse of unzip:

$$\mathsf{zip}_G \circ \mathsf{unzip}_G = \mathsf{id}$$

The composition the other way round will in general not be the identity, because zip is usually a partial function: only data structures of the same shape can be zipped together. Note that unzip returns a pair of data structures of the same shape.

Hoogendijk [13] has made a thorough study of functions like zip (and generalizations involving functors other than 'pair') in the setting of relations. We merely assume the existence of a (possibly partial) natural transformation zip_G of the required type that is a post-inverse of unzip_G as defined above.

5.4 Putting it Together

To conclude, we have the following.

Definition 11. The function $\phi :: C \times F(A,1) \to B \times F(1,C)$ is *shape-preserving* if, when $\phi\,(c,x) = (b,y)$, the shapes of x and y are the same: $\mathsf{shape}_F\ x = \mathsf{shape}_F\ y$; equivalently, if $\mathsf{shape}_F \circ \mathsf{snd} \circ \phi = \mathsf{shape}_F \circ \mathsf{snd}$. □

Definition 12. For shape-preserving ϕ of type $C \times F(A,1) \to B \times F(1,C)$, the downwards accumulation $\mathsf{da}_F\ \phi :: C \times T(A) \to L(B)$ is defined by

$$\mathsf{da}_F\ \phi = \mathsf{unfold}_G\ \psi$$

where

$$\begin{aligned}\psi &:: C \times T(A) \to G(B, C \times T(A))\\ \psi\,(c,t) &= \mathrm{let}\ u = \mathsf{out}_F\ t\\ &\qquad\quad\ v = \phi\,(c, F(\mathsf{id},\mathsf{one})\,u)\\ &\qquad\quad\ w = ((), F(\mathsf{one},\mathsf{id})\,u)\\ &\quad\ \ \mathrm{in}\ \ G(\mathsf{fst},\mathsf{id})\,(\mathsf{zip}_G\,(v,w))\end{aligned}$$

(The point-free characterization

$$\psi = G(\mathsf{fst},\mathsf{id}) \circ \mathsf{zip}_G \circ ((\phi \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F)$$

where

$$\begin{aligned}\pi_1 &= \mathsf{id} \times F(\mathsf{id},\mathsf{one})\\ \pi_2 &= \mathsf{one} \times F(\mathsf{one},\mathsf{id})\end{aligned}$$

is harder to comprehend, but easier to calculate with.) It can be computed in linear time, if ϕ takes constant time. □

Example 13. We can define a generic 'depths' function by

$$depths_F\, x = \mathsf{da}_F\, \phi\, (0, x)$$

where

$$\phi\,(d, z) = (d + 1, F(\mathsf{one}, \mathsf{const}\,(d + 1))\, z)$$

1. For cons lists we have

```
daL :: ((c, Either () a) -> (b, Either () c)) ->
       (c, List a) -> Nlist b
daL phi = unfoldN psi
  where
    psi :: (c, List a) -> (b, Either () (c, List a))
    psi (c, Nil)      = (b, Left ())
                        where (b, Left ()) = phi (c, Left ())
    psi (c, Cons a x) = (b, Right (cx,x))
                        where (b, Right cx) = phi (c, Right a)
```

   For example, the 'depths' function for lists is

```
depthsL :: List a -> Nlist Int
depthsL x = daL phi (0,x)
  where
    phi :: (Int, Either () a) -> (Int, Either () Int)
    phi (d, Left ()) = (d+1, Left ())
    phi (d, Right a) = (d+1, Right (d+1))
```

   returning a list of the form `[1,...,n+1]` where the original list had `n` elements.
2. For non-empty lists we have

```
daN :: ((c, (a, Either () ())) -> (b, Either () c)) ->
       (c, Nlist a) -> Nlist b
daN phi = unfoldN psi
  where
    psi :: (c, Nlist a) -> (b, Either () (c, Nlist a))
    psi (c, ConsN a (Left ()))
      = (b, Left ())
        where (b,Left()) = phi (c, (a, Left ()))
    psi (c, ConsN a (Right x))
      = (b, Right (cx,x))
        where (b,Right cx) = phi (c, (a, Right()))
```

   (Actually, strictly speaking, the shape-preservation requirement forces an extra unit into the result type of `phi`:

```
daN :: ((c, (a, Either () ())) -> (b, ((), Either () c))) ->
       (c, Nlist a) -> Nlist b
```

   which would simply be discarded later:

```
... where (b,((),Left())) = phi (c, (a, Left ()))
... where (b,((),Right cx)) = phi (c, (a, Right()))
```

The definitions in this example are equivalent, but are closer to what a rational programmer would have written.)
The depths function for non-empty lists is

```
depthsN :: Nlist a -> Nlist Int
depthsN x = daN phi (0,x)
  where
    phi :: (Int, (a, Either () ())) -> (Int, Either () Int)
    phi (d, (a, Left ()))  = (d+1, Left ())
    phi (d, (a, Right ())) = (d+1, Right (d+1))
```

returning a list of the form `[1,...,n]` where the original list had `n` elements.

3. For leaf-labelled binary trees we have

```
daT :: ((c, Either a ()) -> (b, Either () (c,c))) ->
       (c, Ltree a) -> Htree b
daT phi = unfoldH psi
  where
    psi :: (c, Ltree a) -> (b, Either () ((c, Ltree a), (c, Ltree a)))
    psi (c,LeafT a)  = (b, Left ())
                       where (b,Left ()) = phi (c, Left a)
    psi (c,BinT t u) = (b, Right ((ct,t),(cu,u)))
                       where (b,Right (ct,cu)) = phi (c, Right ())
```

The depths function for these trees is

```
depthsT :: Ltree a -> Htree Int
depthsT t = daT phi (0,t)
  where
    phi :: (Int, Either a ()) -> (Int, Either () (Int,Int))
    phi (d, Left a)   = (d+1, Left ())
    phi (d, Right ()) = (d+1, Right (d+1,d+1))
```

4. For internally-labelled binary trees we have

```
daB :: ((c, Either () a) -> (b, Either () (c,c))) ->
       (c, Btree a) -> Htree b
daB phi = unfoldH psi
  where
    psi :: (c, Btree a) -> (b, Either () ((c, Btree a), (c, Btree a)))
    psi (c, Empty)
      = (b, Left ())
        where (b, Left ()) = phi (c, Left ())
    psi (c, BinB a t u)
      = (b, Right ((ct,t),(cu,u)))
        where (b, Right (ct,cu)) = phi (c, Right a)
```

The depths function for these trees is

```
depthsB :: Btree a -> Htree Int
depthsB t = daB phi (0,t)
  where
    phi :: (Int, Either () a) -> (Int, Either () (Int,Int))
    phi (d, Left ()) = (d+1, Left ())
    phi (d, Right a) = (d+1, Right (d+1,d+1))
```

5. For homogeneous binary trees we have

```
daH :: ((c, (a, Either () ())) -> (b, Either () (c,c))) ->
       (c, Htree a) -> Htree b
daH phi = unfoldH psi
  where
    psi :: (c, Htree a) -> (b, Either () ((c, Htree a), (c, Htree a)))
    psi (c, ConsH a (Left ()))
      = (b, Left ())
        where (b, Left ()) = phi (c, (a, Left ()))
    psi (c, ConsH a (Right (t,u)))
      = (b, Right ((ct,t),(cu,u)))
        where (b, Right (ct,cu)) = phi (c, (a, Right ()))
```

The depths function for these trees is

```
depthsH :: Htree a -> Htree Int
depthsH t = daH phi (0,t)
  where
    phi :: (Int, (a, Either () ())) -> (Int, Either () (Int,Int))
    phi (d, (a, Left ()))  = (d+1, Left ())
    phi (d, (a, Right ())) = (d+1, Right (d+1,d+1))
```

6. For rose trees we have

```
daR :: ((c,(a, List ())) -> (b, List c)) -> (c, Rtree a) -> Rtree b
daR phi = unfoldR psi
  where
    psi :: (c, Rtree a) -> (b, List (c, Rtree a))
    psi (c, ConsR a ts)
      = (b, zip cs ts)
        where (b,cs) = phi (c,(a, map (const ()) ts))
```

where `zip :: List a -> List b -> List (a,b)` zips together two lists of the same length to produce a list of pairs. The depths function for rose trees is

```
depthsR :: Rtree a -> Rtree Int
depthsR t = daR phi (0,t)
  where
    phi :: (Int, (a, List ())) -> (Int, List Int)
    phi (d, (a, zs)) = (d+1, map (const (d+1)) zs)
```

□

Example 14. For homogeneous types, we can define a 'paths' function, labelling every node with the list of its ancestors. (For a heterogeneous type, we would need to label each node with a 'heterogeneous list', as different nodes can have different kinds of label. We might also want to record the 'direction' taken at each intermediate node to reach a particular node of the data structure. A characterization of paths incorporating these two refinements was used as the basis for defining downwards accumulations in [10], and is what made that approach more complicated than the one taken here.)

For homogeneous types, there is an isomorphism between $F(A,B)$ and $G(A,B)$. We write $\mathit{fg} :: F(A,B) \to G(A,B)$ for the isomorphism in one direction. Then we can define

$$\mathsf{paths}_F\, x = \mathsf{da}_F\, p\,([\,], x)$$

where

$$\begin{aligned} &p &&:: \mathit{List}(A) \times F(A,1) \to \mathit{List}(A) \times F(1, \mathit{List}(A)) \\ &p\,(x,z) &&= (x \mathbin{+\!\!+} [a], F(\mathsf{one}, \mathsf{const}\,(x \mathbin{+\!\!+} [a]))\,z) \\ & &&\quad\text{where } a = \mathsf{fst}\,(\mathit{fg}\,x) \end{aligned}$$

1. For non-empty lists the paths function is

```
pathsN :: Nlist a -> Nlist [a]
pathsN x = daN phi ([],x)
  where
    phi :: ([a], (a, Either () ())) -> ([a], Either () [a])
    phi (p, (a, Left ()))  = (p++[a], Left ())
    phi (p, (a, Right ())) = (p++[a], Right (p++[a]))
```

   returning the initial segments of the list.
2. For homogeneous binary trees we have

```
pathsH :: Htree a -> Htree [a]
pathsH t = daH phi ([],t)
  where
    phi :: ([a], (a, Either () ())) -> ([a], Either () ([a],[a]))
    phi (p, (a, Left ()))  = (p++[a], Left ())
    phi (p, (a, Right ())) = (p++[a], Right (p++[a],p++[a]))
```

3. For rose trees we have

```
pathsR :: Rtree a -> Rtree [a]
pathsR t = daR phi ([],t)
  where
    phi :: ([a], (a, List ())) -> ([a], List [a])
    phi (p, (a, zs))  = (p++[a], map (const (p++[a])) zs)
```

□

6 Properties of Downwards Accumulations

In this section we collect a few theorems and properties of downwards accumulations. We present a universal property, a fusion law, two special cases of fusion with a map, and an accumulation lemma.

As with earlier sections, throughout this one we assume that F is a bifunctor, T is its induced datatype, and L is its labelled variant (induced by bifunctor G). Moreover, recall from Definition 12 that

$$\mathsf{da}_F\ \phi = \mathsf{unfold}_G\ (G(\mathsf{fst},\mathsf{id}) \circ \mathsf{zip}_G \circ ((\phi \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F))$$

where

$$\begin{aligned} \pi_1 &= \mathsf{id} \times F(\mathsf{id},\mathsf{one}) \\ \pi_2 &= \mathsf{one} \times F(\mathsf{one},\mathsf{id}) \end{aligned}$$

6.1 Universal Property

Recall the universal property of unfolds:

$$\begin{array}{l} \quad h = \mathsf{unfold}_G\ \psi \\ \Leftrightarrow \\ \quad \mathsf{out}_G \circ h = G(\mathsf{id},h) \circ \psi \end{array}$$

Because a downwards accumulation is simply an unfold, we obtain for free the following universal property for downwards accumulations.

Theorem 15.

$$\begin{array}{l} \quad h = \mathsf{da}_F\ \phi \\ \Leftrightarrow \\ \quad \mathsf{out}_G \circ h = G(\mathsf{fst},h) \circ \mathsf{zip}_G \circ ((\phi \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F) \end{array}$$

$\square$

Proof.

$$\begin{array}{l} \quad h = \mathsf{da}_F\ \phi \\ \Leftrightarrow \quad \{\ \mathsf{da}\ \} \\ \quad h = \mathsf{unfold}_G\ (G(\mathsf{fst},\mathsf{id}) \circ \mathsf{zip}_G \circ ((\phi \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F)) \\ \Leftrightarrow \quad \{\ \text{universal property of unfold}\ \} \\ \quad \mathsf{out}_G \circ h = G(\mathsf{id},h) \circ G(\mathsf{fst},\mathsf{id}) \circ \mathsf{zip}_G \circ ((\phi \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F) \\ \Leftrightarrow \quad \{\ \text{functors}\ \} \\ \quad \mathsf{out}_G \circ h = G(\mathsf{fst},h) \circ \mathsf{zip}_G \circ ((\phi \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F) \end{array}$$

$\square$

6.2 Fusion Law

Likewise, corresponding to the fusion law for unfolds there is a fusion law for downwards accumulations. The fusion law for unfolds is as follows.

Lemma 16.

$$
\begin{array}{ll}
 & \mathsf{unfold}_G\, \psi \circ h = \mathsf{unfold}_G\, \psi' \\
\Leftarrow & \\
 & \psi \circ h = G(\mathsf{id}, h) \circ \psi'
\end{array}
$$

□

Proof.

$$
\begin{array}{ll}
 & \mathsf{unfold}_G\, \psi \circ h = \mathsf{unfold}_G\, \psi' \\
\Leftrightarrow & \{ \text{universal property of unfolds} \} \\
 & \mathsf{out}_G \circ \mathsf{unfold}_G\, \psi \circ h = G(\mathsf{id}, \mathsf{unfold}_G\, \psi \circ h) \circ \psi' \\
\Leftrightarrow & \{ \text{unfold evaluation} \} \\
 & G(\mathsf{id}, \mathsf{unfold}_G\, \psi) \circ \psi \circ h = G(\mathsf{id}, \mathsf{unfold}_G\, \psi \circ h) \circ \psi' \\
\Leftarrow & \{ \text{functors; Leibniz} \} \\
 & \psi \circ h = G(\mathsf{id}, h) \circ \psi'
\end{array}
$$

□

Instantiating both unfolds as downwards accumulations gives rise to the following fusion law for downwards accumulations.

Corollary 17.

$$
\begin{array}{ll}
 & \mathsf{da}_F\, \phi \circ h = \mathsf{da}_F\, \phi' \\
\Leftarrow & \\
 & \mathsf{zip}_G \circ ((\phi \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F) \circ h \\
 & \qquad = G(\mathsf{id}, h) \circ \mathsf{zip}_G \circ ((\phi' \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F)
\end{array}
$$

□

6.3 Maps and Downwards Accumulations

Two common special cases of fusion are with a map, either before or after the downwards accumulation.

Theorem 18.

$$
\mathsf{map}_G\, f \circ \mathsf{da}_F\, \phi = \mathsf{da}_F\, (G(f, \mathsf{id}) \circ \phi)
$$

□

Proof.

$$
\begin{array}{ll}
& \mathsf{map}_G\, f \circ \mathsf{da}_F\, \phi = \mathsf{da}_F\, \phi' \\
\Leftrightarrow & \{\ \text{universal property}\ \} \\
& \mathsf{out}_G \circ \mathsf{map}_G\, f \circ \mathsf{da}_F\, \phi \\
& \quad = G(\mathsf{fst}, \mathsf{map}_G\, f \circ \mathsf{da}_F\, \phi) \circ \mathsf{zip}_G \circ ((\phi' \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F) \\
\Leftrightarrow & \{\ \text{evaluation for } \mathsf{map} \text{ and } \mathsf{da}\ \} \\
& G(f, \mathsf{map}_G\, f) \circ G(\mathsf{fst}, \mathsf{da}_F\, \phi) \circ \mathsf{zip}_G \circ ((\phi \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F) \\
& \quad = G(\mathsf{fst}, \mathsf{map}_G\, f \circ \mathsf{da}_F\, \phi) \circ \mathsf{zip}_G \circ ((\phi' \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F) \\
\Leftarrow & \{\ \text{Leibniz}\ \} \\
& G(f \circ \mathsf{fst}, \mathsf{id}) \circ \mathsf{zip}_G \circ (\phi \times \mathsf{id}) = G(\mathsf{fst}, \mathsf{id}) \circ \mathsf{zip}_G \circ (\phi' \times \mathsf{id}) \\
\Leftarrow & \{\ \text{naturality of } \mathsf{zip}\text{; pairs}\ \} \\
& G(f, \mathsf{id}) \circ \phi = \phi'
\end{array}
$$

□

Theorem 19.

$$\mathsf{da}_F\, \phi \circ (\mathsf{id} \times \mathsf{map}_F\, f) = \mathsf{da}_F\, (\phi \circ (\mathsf{id} \times F(f, \mathsf{id})))$$

□

Proof.

$$
\begin{array}{ll}
& \mathsf{da}_F\, \phi \circ (\mathsf{id} \times \mathsf{map}_F\, f) = \mathsf{da}_F\, \phi' \\
\Leftrightarrow & \{\ \text{universal property}\ \} \\
& \mathsf{out}_G \circ \mathsf{da}_F\, \phi \circ (\mathsf{id} \times \mathsf{map}_F\, f) \\
& \quad = G(\mathsf{fst}, \mathsf{da}_F\, \phi \circ (\mathsf{id} \times \mathsf{map}_F\, f)) \circ \mathsf{zip}_G \circ ((\phi' \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F) \\
\Leftrightarrow & \{\ \text{evaluation for } \mathsf{da}\ \} \\
& G(\mathsf{fst}, \mathsf{da}_F\, \phi) \circ \mathsf{zip}_G \circ ((\phi \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F) \circ (\mathsf{id} \times \mathsf{map}_F\, f) \\
& \quad = G(\mathsf{fst}, \mathsf{da}_F\, \phi \circ (\mathsf{id} \times \mathsf{map}_F\, f)) \circ \mathsf{zip}_G \circ ((\phi' \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F) \\
\Leftarrow & \{\ \text{Leibniz; evaluation for } \mathsf{map}\ \} \\
& \mathsf{zip}_G \circ ((\phi \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times (F(f, \mathsf{map}_F\, f) \circ \mathsf{out}_F)) \\
& \quad = G(\mathsf{id}, \mathsf{id} \times \mathsf{map}_F\, f) \circ \mathsf{zip}_G \circ ((\phi' \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F) \\
\Leftarrow & \{\ \text{Leibniz; naturality of } \mathsf{zip}\ \} \\
& ((\phi \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times F(f, \mathsf{map}_F\, f)) \\
& \quad = (\mathsf{id} \times G(\mathsf{id}, \mathsf{map}_F\, f)) \circ ((\phi' \circ \pi_1) \vartriangle \pi_2) \\
\Leftrightarrow & \{\ \text{pairs; relationship between } F \text{ and } G\ \} \\
& \phi \circ \pi_1 \circ (\mathsf{id} \times F(f, \mathsf{map}_F\, f)) = \phi' \circ \pi_1 \\
& \pi_2 \circ (\mathsf{id} \times F(f, \mathsf{map}_F\, f)) = (\mathsf{id} \times F(\mathsf{id}, \mathsf{map}_F\, f)) \circ \pi_2 \\
\Leftarrow & \{\ \pi_1,\ \pi_2\ \} \\
& \phi \circ (\mathsf{id} \times F(f, \mathsf{id})) = \phi' \\
& (\mathsf{id} \times F(\mathsf{id}, \mathsf{map}_F\, f)) \circ \pi_2 = (\mathsf{id} \times F(\mathsf{id}, \mathsf{map}_F\, f)) \circ \pi_2 \\
\Leftarrow & \{\ \text{Leibniz}\ \} \\
& \phi \circ (\mathsf{id} \times F(f, \mathsf{id})) = \phi'
\end{array}
$$

□

6.4 An Accumulation Lemma for Homogeneous Datatypes

We have seen in Example 14 that the 'paths' function on a homogeneous datatype can be expressed as a downwards accumulation. An *accumulation lemma* for downwards accumulations is a kind of converse of this observation: that (some) downwards accumulations can be expressed in terms of paths. Such lemmas have turned out to be very convenient for program calculation [4, 5, 1]. This section presents such a lemma.

Note that we have only a partial converse: not every downwards accumulation (even on homogeneous datatypes) can be expressed in terms of paths. In particular, the 'paths' function discards information distinguishing the different children of a node, so a downwards accumulation that treats a left child differently from a right child cannot be expressed in this way. The approach taken to defining downwards accumulations in [10], on the other hand, involved a polytypic definition of the 'paths' function that maintained all such information. As a consequence, in that paper the accumulation lemma applied to every downwards accumulation (indeed, that was how downwards accumulation were defined); however, the construction was much more complex than the one in this paper.

Definition 20. An *accumulating fold* over lists is defined as follows:

$$\begin{array}{ll} \mathsf{afold} & :: (B \to A \to B) \to B \to List(A) \to B \\ \mathsf{afold}\, f\, e\, [\,] & = e \\ \mathsf{afold}\, f\, e\, (x \mathbin{+\!\!+} [a]) & = f\, (\mathsf{afold}\, f\, e\, x)\, a \end{array}$$

□

The equation

$$\mathsf{afold}\, f\, e\, ([a] \mathbin{+\!\!+} x) = \mathsf{afold}\, f\, (f\, e\, a)\, x$$

is an alternative characterization of the behaviour on non-empty lists. This reveals the motivation for the name 'accumulating fold': the second argument acts as an accumulating parameter [2]. The same function is called `foldl` in Haskell.

Example 21. The length function on lists can be defined by

$$length = \mathsf{afold}\, next\, 0 \qquad \text{where } next\, d\, a = d + 1$$

□

Theorem 22.

$$\mathsf{map}_G\, (\mathsf{afold}\, f\, e)\, (\mathsf{paths}_F\, x) = \mathsf{da}_F\, \phi\, (e, x)$$

where

$$\phi\, (u, z) = (f\, u\, a, F(\mathsf{one}, \mathsf{const}\, (f\, u\, a))\, z) \qquad \text{where } a = \mathsf{fst}\, (fg\, z)$$

□

Example 23. The 'depths' function for a homogeneous datatype can be viewed as computing the lengths of every path:

$$depths_F = \mathsf{map}_G\, length \circ \mathsf{paths}_F$$

Now Theorem 22 concludes that

$$\mathit{depths}_F\, x = \mathsf{da}_F\, \phi\,(0,x)$$

where

$$\phi\,(d,z) = (d+1, F(\mathsf{one}, \mathsf{const}\,(d+1))\,z)$$

—which is just the characterization of depths_F given in Example 13. □

Proof (of Theorem 22). Throughout this proof, we assume that p is as defined in Example 14. We have

$$\begin{array}{cl}
& \mathsf{map}_G\,(\mathsf{afold}\, f\, e)\,(\mathsf{paths}_F\, x) \\
= & \quad \{\ \mathsf{paths}\ \} \\
& \mathsf{map}_G\,(\mathsf{afold}\, f\, e)\,(\mathsf{da}_F\, p\,([\,],x)) \\
= & \quad \{\ \text{fusion (Theorem 18)}\ \} \\
& \mathsf{da}_F\,(G(\mathsf{afold}\, f\, e, \mathsf{id}) \circ p)\,([\,],x)
\end{array}$$

Now we must extract the afold from the accumulation, so we explore the conditions under which we can do so.

$$\begin{array}{cl}
& \mathsf{da}_F\, \phi \circ (\mathsf{afold}\, f\, e \times \mathsf{id}) = \mathsf{da}_F\, \phi' \\
\Leftarrow & \quad \{\ \text{fusion}\ \} \\
& \mathsf{zip}_G \circ ((\phi \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F) \circ (\mathsf{afold}\, f\, e \times \mathsf{id}) \\
& \qquad = G(\mathsf{id}, \mathsf{afold}\, f\, e \times \mathsf{id}) \circ \mathsf{zip}_G \circ ((\phi' \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{id} \times \mathsf{out}_F) \\
\Leftarrow & \quad \{\ \text{naturality of } \mathsf{zip}\text{; pairs; Leibniz}\ \} \\
& ((\phi \circ \pi_1) \vartriangle \pi_2) \circ (\mathsf{afold}\, f\, e \times \mathsf{id}) \\
& \qquad = (G(\mathsf{id}, \mathsf{afold}\, f\, e) \times \mathsf{id}) \circ ((\phi' \circ \pi_1) \vartriangle \pi_2) \\
\Leftrightarrow & \quad \{\ \text{pairs again}\ \} \\
& \phi \circ \pi_1 \circ (\mathsf{afold}\, f\, e \times \mathsf{id}) = G(\mathsf{id}, \mathsf{afold}\, f\, e) \circ \phi' \circ \pi_1 \\
& \pi_2 \circ (\mathsf{afold}\, f\, e \times \mathsf{id}) = \pi_2 \\
\Leftarrow & \quad \{\ \pi_1 \text{ commutes with } h \times \mathsf{id}\text{; } \pi_2 \text{ absorbs } h \times \mathsf{id}\ \} \\
& \phi \circ (\mathsf{afold}\, f\, e \times \mathsf{id}) = G(\mathsf{id}, \mathsf{afold}\, f\, e) \circ \phi'
\end{array}$$

So in order to extract the afold from the accumulation, we must find a ϕ such that

$$\phi \circ (\mathsf{afold}\, f\, e \times \mathsf{id}) = G(\mathsf{id}, \mathsf{afold}\, f\, e) \circ G(\mathsf{afold}\, f\, e, \mathsf{id}) \circ p$$

We start with the right-hand side:

$$\begin{array}{cl}
& (G(\mathsf{id}, \mathsf{afold}\, f\, e) \circ G(\mathsf{afold}\, f\, e, \mathsf{id}) \circ p)\,(x,z) \\
= & \quad \{\ \text{let } a = \mathsf{fst}\,(fg\, z)\ \} \\
& G(\mathsf{afold}\, f\, e, \mathsf{afold}\, f\, e)\,(x \mathbin{+\!\!+} [a], F(\mathsf{one}, \mathsf{const}\,(x \mathbin{+\!\!+} [a]))\,z) \\
= & \quad \{\ \text{relationship between } G \text{ and } F\ \} \\
& (\mathsf{afold}\, f\, e\,(x \mathbin{+\!\!+} [a]), F(\mathsf{one}, \mathsf{const}\,(\mathsf{afold}\, f\, e\,(x \mathbin{+\!\!+} [a])))\,z) \\
= & \quad \{\ \mathsf{afold}\ \} \\
& (f\,(\mathsf{afold}\, f\, e\, x)\, a, F(\mathsf{one}, \mathsf{const}\,(f\,(\mathsf{afold}\, f\, e\, x)\, a))\,z) \\
= & \quad \{\ \text{let } \phi\,(u,z) = (f\, u\, a, F(\mathsf{one}, \mathsf{const}\,(f\, u\, a))\,z) \text{ where } a = \mathsf{fst}\,(fg\, z)\ \} \\
& \phi\,(\mathsf{afold}\, f\, e\, x, z)
\end{array}$$

obtaining the desired ϕ. Therefore,

$$
\begin{aligned}
& \mathsf{map}_G\,(\mathsf{afold}\,f\,e)\,(\mathsf{paths}_F\,x) \\
= & \quad \{\ \text{first calculation above}\ \} \\
& \mathsf{da}_F\,(G(\mathsf{afold}\,f\,e, \mathsf{id}) \circ p)\,([\,], x) \\
= & \quad \{\ \phi \text{ as above; extracting the } \mathsf{afold}\ \} \\
& \mathsf{da}_F\,\phi\,(e, x)
\end{aligned}
$$

□

7 Conclusion

We have shown how to generalize the notion of *downwards accumulation* [7,8] to an arbitrary regular datatype, building on Bird *et al.*'s [1] generalization of upwards accumulation. The definition is as an unfold, further evidence of the usefulness [11] (and under-appreciation, not least on the author's part, given that it has taken so long to discover this characterization) of this higher-order operator.

In an earlier version of this paper [10], we attempted to solve the problem of constructing a generic definition of downwards accumulation by considering the 'paths' to elements of an arbitrary data structure. The path to an element records the ancestors of that element in the data structure; a downwards accumulation consists of a fold mapped over the paths. (Symmetrically, the descendants of an element of a data structure form a 'subtree' of that data structure, and an upwards accumulation consists of a fold mapped over the subtrees of the data structure.) We do something similar but much simpler in Example 14: there we model a path merely as a list of elements, whereas in [10] we had to include also the 'directions' along a path and to consider nodes with different kinds of label.

In retrospect, it appears that starting with paths was the wrong way to approach the problem. Modelling the paths of an arbitrary datatype involves 'linearizing' the type functor for that type, among other reasons to determine the branching degree of a node. This was quite a complicated construction; the approach taken in this paper is much simpler.

Another advantage of the approach presented here is that it leads to a *generic* or *parametric higher-order polymorphic* instead of a *polytypic* or *ad-hoc higher-order polymorphic* construction. By the former we mean a construction based on semantic properties of the type functors concerned, such as Bird *et al.*'s labelling construction $G(A, X) = A \times F(1, X)$. By the latter we mean a construction by induction over the syntactic presentation of the type functor, in the style of Jeuring [14]. (In fact, our characterization in [10] was not even as general as Jeuring's scheme: we required the functor to be polynomial, that is, a sum of products, rather than regular as Jeuring allows.) Hoogendijk [12,13] argues for the inherent superiority of generic over polytypic definitions, but for us there is the added bonus of applicability to regular but non-polynomial datatypes such as rose trees.

Acknowledgements

The members of the Problem Solving Club at Oxford University Computing Laboratory, especially Richard Bird, Jesús Ravelo and Pedro Borges, have made many helpful suggestions and comments; Ross Paterson and the anonymous *Mathematics of Program Construction* and *Science of Computer Programming* referees have also suggested many improvements. The commuting diagrams were drawn using Paul Taylor's LaTeX macros.

References

1. Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.
2. 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 [3].
3. 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.
4. 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.
5. Richard S. Bird, Jeremy Gibbons, and Geraint Jones. Formal derivation of a pattern matching algorithm. *Science of Computer Programming*, 12(2):93–104, July 1989.
6. Maarten M. Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, January 1991.
7. Jeremy Gibbons. *Algebras for Tree Algorithms*. D.Phil. thesis, Programming Research Group, Oxford University, 1991. Available as Technical Monograph PRG-94.
8. 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.
9. Jeremy Gibbons. Deriving tidy drawings of trees. *Journal of Functional Programming*, 6(3):535–562, 1996.
10. Jeremy Gibbons. Polytypic downwards accumulations. In Johan Jeuring, editor, *LNCS 1422: Proceedings of Mathematics of Program Construction*, Marstrand, Sweden, June 1998. Springer-Verlag.
11. Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 273–279, Baltimore, Maryland, September 1998.
12. Paul Hoogendijk. *A Generic Theory of Datatypes*. PhD thesis, TU Eindhoven, 1997.
13. Paul Hoogendijk and Roland Backhouse. When do datatypes commute? In Eugenio Moggi and Guiseppe Rosolini, editors, *LNCS 1290: Category Theory and Computer Science*, pages 242–260. Springer-Verlag, September 1997.

14. Patrick Jansson and Johan Jeuring. PolyP – a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, 1997.
15. Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.
16. Ian A. MacLeod. A query language for retrieving information from hierarchical text structures. *Computer Journal*, 34(3):254–264, 1991.
17. Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
18. Lambert Meertens. First steps towards the theory of rose trees. CWI, Amsterdam; IFIP Working Group 2.1 working paper 592 ROM-25, 1988.
19. Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
20. 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.
21. Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, March 1981.
22. John C. Reynolds. Semantics of the domain of flow diagrams. *Journal of the ACM*, 24(3):484–503, 1977.
23. David Skillicorn. *Foundations of Parallel Programming*. Number 6 in International Series on Parallel Computation. Cambridge University Press, 1994.
24. Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.