Accumulating Attributes

for Doaitse Swierstra, on his retirement

Jeremy Gibbons

Department of Computer Science, University of Oxford
http://www.cs.ox.ac.uk/jeremy.gibbons/

Abstract. Doaitse has always been enthusiastic about attribute grammars, seeing them even where most people don't. While I was writing up my DPhil thesis, he explained to me where they were in that too. This paper is by way of belated thanks for that perspective—and as a more general rendering of the observation, with the benefit of hindsight and twenty years of progress.

1 Introduction

I first met Doaitse Swierstra and became exposed to his predilections at the STOP project summer school on Ameland in 1989. For as long as I have known him, Doaitse has been devoted to attribute grammars—in compiler technology, in program design, and simply in structuring his thought processes.

I next visited the Netherlands in the summer of 1991. I was deep in the middle of writing up my DPhil thesis [1] on upwards and downwards accumulations on trees, and during my trip I gave a couple of seminars about my work, including one at Utrecht hosted by Doaitse. As many who have met him will attest, Doaitse is like a Frenchman: whatever you say to him, he translates into his own language, and forthwith it is something entirely different. In this case, he immediately explained to me that my thesis was really about attribute grammars, and not about accumulations at all.

In a sense, he was right. Indeed, his observation grew into the final chapter of my thesis, which attempted to explain the connection. At the time of my visit, I was looking for one more substantive piece of work to balance out what I already had, so this contribution was very welcome. But I'm not sure that I ever thanked him properly for his gift, and this is my opportunity to do so. Besides, I can now tell the story much better than I could twenty-odd years ago: in particular, I can do so datatype-generically.

So, thank you, Doaitse, for your ever fresh way of looking at the world!

2 Origami programming

One of the advances in functional programming since I did my thesis has been a much better understanding of what I have been calling 'datatype-generic' [2]

and 'origami' [3] programming—that is, the use of structured recursion operators such as maps, folds, and unfolds, parametrized by the shape of data. These ideas were developing at the time, in the work of Hagino [4] and Malcolm [5], but I for one didn't really appreciate them until some years later. With the wisdom of hindsight, the results in my thesis can be presented in this style too.

We will discuss attribute grammars in terms of labelling every node of a data structure. So for simplicity, in this paper we stick to labelled data structures—every node has precisely one label, of type a, and an f-structure of children:

$$\mathbf{data}\ \mathit{Mu}\ f\ a = \mathit{In}\ \{\mathit{root} :: a, \mathit{kids} :: f\ (\mathit{Mu}\ f\ a)\}$$

The datatype $\mathit{Mu}\ f$ supports numerous datatype-generic origami operations, parametrized by the shape function f. Of these, we will only make use in this paper of folds, not unfolds or other more sophisticated origami operators:

$$\begin{aligned}&\mathit{fold} :: \mathit{Functor}\ f \Rightarrow (a \to f\ b \to b) \to \mathit{Mu}\ f\ a \to b\\ &\mathit{fold}\ \phi\ t = \phi\ (\mathit{root}\ t)\ (\mathit{fmap}\ (\mathit{fold}\ \phi)\ (\mathit{kids}\ t))\end{aligned}$$

We'll also need ad-hoc datatype genericity; so we define a universe of polynomial functors.

$$\begin{aligned}&\mathbf{data}\ \mathit{Unit}\ a &&= \mathit{Unit}\\ &\mathbf{data}\ \mathit{Id}\ a &&= \mathit{Id}\ a\\ &\mathbf{data}\ \mathit{Sum}\ f\ g\ a &&= \mathit{Inl}\ (f\ a) \mid \mathit{Inr}\ (g\ a)\\ &\mathbf{data}\ \mathit{Prod}\ f\ g\ a &&= f\ a :\times: g\ a\end{aligned}$$

Here also is the empty datatype—it will be useful for some constructions, although we won't consider it part of our universe of functors:

$$\mathbf{data}\ \mathit{Zero}\ a$$

There are no constructors for Zero, so no (proper) values of that type; therefore, if you were somehow to be able to come up a value of type Zero, you deserve to be able to turn it into anything you want:

$$\begin{aligned}&\mathit{magic} :: \mathit{Zero}\ a \to b\\ &\mathit{magic}\ z = \mathit{seq}\ z\ (\mathit{error}\ \texttt{"It must be magic"})\end{aligned}$$

All these codes represent functors, of course:

$$\begin{aligned}&\mathbf{instance}\ \mathit{Functor}\ \mathit{Zero}\ \mathbf{where}\\ &\quad \mathit{fmap}\ f\ z &&= \mathit{magic}\ z\\ &\mathbf{instance}\ \mathit{Functor}\ \mathit{Unit}\ \mathbf{where}\\ &\quad \mathit{fmap}\ f\ \mathit{Unit} &&= \mathit{Unit}\\ &\mathbf{instance}\ \mathit{Functor}\ \mathit{Id}\ \mathbf{where}\\ &\quad \mathit{fmap}\ f\ (\mathit{Id}\ a) &&= \mathit{Id}\ (f\ a)\\ &\mathbf{instance}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Functor}\ (\mathit{Sum}\ f\ g)\ \mathbf{where}\\ &\quad \mathit{fmap}\ f\ (\mathit{Inl}\ x) &&= \mathit{Inl}\ (\mathit{fmap}\ f\ x)\\ &\quad \mathit{fmap}\ f\ (\mathit{Inr}\ y) &&= \mathit{Inr}\ (\mathit{fmap}\ f\ y)\end{aligned}$$

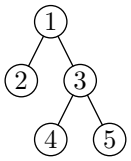

Fig. 1. The tree $t = node\ 1\ (leaf\ 2)\ (node\ 3\ (leaf\ 4)\ (leaf\ 5))$

```
instance (Functor f, Functor g) ⇒ Functor (Prod f g) where
  fmap f (x :×: y) = fmap f x :×: fmap f y
```

For example, *TreeF* is the code for the shape functor of homogeneous binary trees, and *sum* is an example fold for that datatype:

```
type TreeF = Sum Unit (Prod Id Id)
type Tree  = Mu TreeF

add :: Int → TreeF Int → Int
add m (Inl Unit)              = m
add m (Inr (Id n :×: Id p)) = m + n + p

sum :: Tree Int → Int
sum = fold add
```

For convenience, here are two 'constructors' for *Tree*:

```
leaf :: a → Tree a
leaf a = In a (Inl Unit)

node :: a → Tree a → Tree a → Tree a
node a t u = In a (Inr (Id t :×: Id u))
```

Figure 1 shows a small tree t, which we will use for examples.

3 Accumulations on lists

My thesis was about upwards and downwards accumulations on trees, which were intended to be analogous to the accumulations (or 'scans') on lists that had proven so fruitful in Bird's work on the Theory of Lists [6–8]. Recall the standard definitions of *tails*, *foldr*, and *scanr* from the Haskell libraries:

```
tails :: [a] → [[a]]
tails []  = [[]]
tails x   = x : tails (tail x)

foldr :: (a → b → b) → b → [a] → b
foldr f e []      = e
foldr f e (a : x) = f a (foldr f e x)

scanr :: (a → b → b) → b → [a] → [b]
```

$$
\begin{aligned}
&\mathit{scanr}\ f\ e\ [\,] &&= [\,e\,]\\
&\mathit{scanr}\ f\ e\ (a : x) &&= f\ a\ (\mathit{head}\ y) : y\ \mathbf{where}\ y = \mathit{scanr}\ f\ e\ x
\end{aligned}
$$

Here, *scanr* computes all the partial results of a fold, from the right:

$$\mathit{scanr}\ (+)\ 0\ [1,2,3] = [6,5,3,0]$$

It satisfies a very important 'scan lemma', stating that these partial results are precisely the results of folding each of the tails:

$$\mathit{scanr}\ f\ e = \mathit{map}\ (\mathit{foldr}\ f\ e) \circ \mathit{tails}$$

The scan lemma is crucial in deriving numerous efficient algorithms over lists, not least for the famous 'maximum segment sum' problem [8].

Dually, there are functions that work from the opposite end of the list:

$$
\begin{aligned}
&\mathit{inits} :: [\,a\,] \to [\,[\,a\,]\,]\\
&\mathit{inits}\ [\,] &&= [\,[\,]\,]\\
&\mathit{inits}\ (a : x) &&= [\,] : \mathit{map}\ (a{:})\ (\mathit{inits}\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{foldl} :: (b \to a \to b) \to b \to [\,a\,] \to b\\
&\mathit{foldl}\ f\ e\ [\,] &&= e\\
&\mathit{foldl}\ f\ e\ (a : x) &&= \mathit{foldl}\ f\ (f\ e\ a)\ x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{scanl} :: (b \to a \to b) \to b \to [\,a\,] \to [\,b\,]\\
&\mathit{scanl}\ f\ e\ [\,] &&= [\,e\,]\\
&\mathit{scanl}\ f\ e\ (a : x) &&= e : \mathit{scanl}\ f\ (f\ e\ a)\ x
\end{aligned}
$$

for which

$$\mathit{scanl}\ (+)\ 0\ [1,2,3] = [0,1,3,6]$$

Again, there is a scan lemma:

$$\mathit{scanl}\ f\ e = \mathit{map}\ (\mathit{foldl}\ f\ e) \circ \mathit{inits}$$

With these six functions as my inspiration, the essence of my thesis work was to generalize them to trees of various kinds.

4 Upwards accumulations

Generalizing *tails* and *scanr* is the easier task, because tail segments follow the structure of the datatype, whereas *inits* and *scanl* in some sense go against the grain. The idea is that *tails* labels every node of a list with the tail starting at that position, and *scanr* labels every node with the fold of that tail. My thesis made some ad-hoc generalizations of this idea to various kinds of tree. It wasn't until some years later [9] that a datatype-generic construction was discovered, involving a systematic way of deriving a 'labelled variant' for any datatype, which has (precisely) one label at each node. Happily, in this paper we have already restricted ourselves to such datatypes, so we don't need this construction—the 'labelled variant' of $\mathit{Mu}\ f$ is $\mathit{Mu}\ f$ itself.

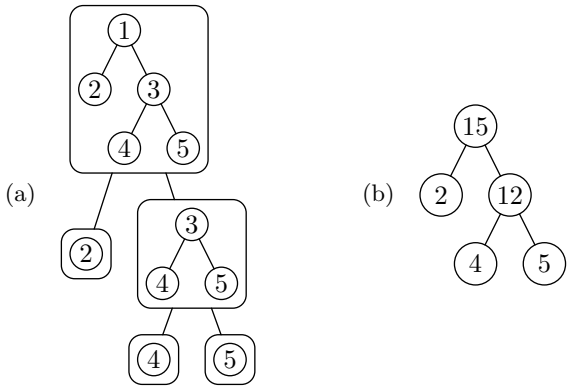

Fig. 2. (a) *subtrees t* and (b) *scanu add t*

Then *subtrees*, the datatype-generic version of *tails*, labels every node of a tree with the subtree rooted at that node. The root label of the output is the whole of the input; and each child in the output is generated from the corresponding child in the input.

$$subtrees :: Functor\ f \Rightarrow Mu\ f\ a \to Mu\ f\ (Mu\ f\ a)$$
$$subtrees = fold\ \psi\ \textbf{where}\ \psi\ a\ ts = In\ (In\ a\ (fmap\ root\ ts))\ ts$$

An upwards accumulation is like *subtrees*, except that it folds every tree it generates. It does this as it goes, so it takes no longer to compute than a mere fold of the input tree does; which is to say, it records all the partial results already involved in folding the original tree.

$$scanu :: Functor\ f \Rightarrow (a \to f\ b \to b) \to Mu\ f\ a \to Mu\ f\ b$$
$$scanu\ \phi = fold\ \psi\ \textbf{where}\ \psi\ a\ ts = In\ (\phi\ a\ (fmap\ root\ ts))\ ts$$

Note that we again have the all-important scan lemma:

$$scanu\ \phi = fmap\ (fold\ \phi) \circ subtrees$$

Figure 2 shows (a) *subtrees t* and (b) *scanu add t*, where t is the tree in Figure 1.

5 Derivatives of datatypes

Generalizing *inits* and *scanl* is more difficult. The function *inits* labels every node of a list with its list of predecessors, and the datatype-generic version should label every node of a data structure with the ancestors of that node; but the ancestors form a completely different datatype, of linear shape whatever the branching structure of the original. Similarly, a downwards accumulation *scand* will label every node of a data structure with some function of its ancestors—and not just any function, but some kind of fold that will allow us to compute

the whole scan in linear time. To capture ancestors, we need to make a detour through derivatives of datatypes [10, 11].

The derivative of a functor f is another functor Δf such that $\Delta f\ a$ is like $f\ a$ but with precisely one a missing: $\Delta f\ a$ is a 'one-hole context' for $f\ a$ with respect to a. We model this idea through a type class *Diff* (of differentiable functors), which has Δ as an associated type synonym:

```
class Functor f ⇒ Diff f where
  type Δf :: ∗ → ∗
```

The class also has two methods, to produce and to consume holes:

```
posns :: f a → f (a, Δf a)
plug  :: (a, Δf a) → f a
```

The idea is that *posns* takes a complete piece of data, and labels every element a in it with the one-hole-context for which a completes the original data structure; whereas *plug* takes a one-hole-context and a value to fill that hole, and puts the latter in the former to make a complete piece of data. The two are related by the following two laws (which I believe completely determine the implementation):

```
fmap fst (posns x)   = x
fmap plug (posns x) = fmap (const x) x
```

Informally, the first law states that *posns* really annotates, so that discarding the annotations is a left inverse; and the second that plugging together each pair in the output of *posns* produces many copies of the original data structure, one for each element.

Here are the *Diff* instances for our universe of functors. The constant functor has no elements, so the derivative is the empty type, *posns* has no effect, and you can never have a hole to plug.

```
instance Diff Unit where
  type ΔUnit = Zero
  posns Unit = Unit
  plug (a, z)  = magic z
```

The identity functor contains precisely one element, so what's left when this is deleted is the unit type:

```
instance Diff Id where
  type ΔId = Unit
  posns (Id a)   = Id (a, Unit)
  plug (a, Unit) = Id a
```

A value of a sum type is either of the left summand or of the right, and in each case a one-hole context is a corresponding one-hole context for that summand; so the two methods simply follow the structure.

```
instance (Diff f, Diff g) ⇒ Diff (Sum f g) where
  type Δ(Sum f g) = Sum (Δf) (Δg)
```

$$
\begin{aligned}
&\mathit{posns}\ (\mathit{Inl}\ x) &&= \mathit{Inl}\ (\mathit{fmap}\ (\lambda(a, dx) \to (a, \mathit{Inl}\ dx))\ (\mathit{posns}\ x)) \\
&\mathit{posns}\ (\mathit{Inr}\ y) &&= \mathit{Inr}\ (\mathit{fmap}\ (\lambda(a, dy) \to (a, \mathit{Inr}\ dy))\ (\mathit{posns}\ y)) \\
&\mathit{plug}\ (a, \mathit{Inl}\ x) &&= \mathit{Inl}\ (\mathit{plug}\ (a, x)) \\
&\mathit{plug}\ (a, \mathit{Inr}\ y) &&= \mathit{Inr}\ (\mathit{plug}\ (a, y))
\end{aligned}
$$

A value of a product type is a pair, and a one-hole context for the pair is either a one-hole context for the left half, together with an intact right half, or an intact left half and a context for the right half.

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Diff}\ f, \mathit{Diff}\ g) \Rightarrow \mathit{Diff}\ (\mathit{Prod}\ f\ g)\ \textbf{where} \\
&\quad \textbf{type}\ \Delta(\mathit{Prod}\ f\ g) = \mathit{Sum}\ (\mathit{Prod}\ (\Delta f)\ g)\ (\mathit{Prod}\ f\ (\Delta g)) \\
&\quad \mathit{posns}\ (x :\times: y) \qquad\qquad = (\mathit{fmap}\ (\lambda(a, dx) \to (a, \mathit{Inl}\ (dx :\times: y)))\ (\mathit{posns}\ x)) :\times: \\
&\qquad\qquad\qquad\qquad\quad\ \ (\mathit{fmap}\ (\lambda(a, dy) \to (a, \mathit{Inr}\ (x :\times: dy)))\ (\mathit{posns}\ y)) \\
&\quad \mathit{plug}\ (a, \mathit{Inl}\ (dx :\times: y)) = \mathit{plug}\ (a, dx) :\times: y \\
&\quad \mathit{plug}\ (a, \mathit{Inr}\ (x :\times: dy)) = x :\times: \mathit{plug}\ (a, dy)
\end{aligned}
$$

For example, consider the type $\mathit{TreeF}\ \mathit{Int}$, whose values are either a unit or a pair of integers. The derivative $\Delta\,\mathit{TreeF}\ \mathit{Int}$ of this type represents data structures with one missing integer. Expanding the definitions from the type class instances, we see that

$$\Delta\,\mathit{TreeF} = \mathit{Sum}\ \mathit{Zero}\ (\mathit{Sum}\ (\mathit{Prod}\ \mathit{Unit}\ \mathit{Id})\ (\mathit{Prod}\ \mathit{Id}\ \mathit{Unit}))$$

The left-hand variant of this sum is void: corresponding TreeF values are just a unit, and there is no way for such a value to be missing an integer. The right-hand variant is itself a sum: corresponding TreeF values have two integers, so there are two ways for such a value to be missing an integer, and in each case what remains is the other integer.

Here is a piece of data in the right-hand variant of $\mathit{TreeF}\ \mathit{Int}$:

$$
\begin{aligned}
&u :: \mathit{TreeF}\ \mathit{Int} \\
&u = \mathit{Inr}\ (\mathit{Id}\ 3 :\times: \mathit{Id}\ 4)
\end{aligned}
$$

Here are two one-hole contexts for u, in each case having the unit value in place of one of the integers:

$$
\begin{aligned}
&v_1, v_2 :: \Delta\,\mathit{TreeF}\ \mathit{Int} \\
&v_1 = \mathit{Inr}\ (\mathit{Inl}\ (\mathit{Unit} :\times: \mathit{Id}\ 4)) \\
&v_2 = \mathit{Inr}\ (\mathit{Inr}\ (\mathit{Id}\ 3 :\times: \mathit{Unit}))
\end{aligned}
$$

If you plug the correct integer back into each context:

$$
\begin{aligned}
&u_1, u_2 :: \mathit{TreeF}\ \mathit{Int} \\
&u_1 = \mathit{plug}\ (3, v_1) \\
&u_2 = \mathit{plug}\ (4, v_2)
\end{aligned}
$$

then you get the original data back again: $u = u_1 = u_2$.

5.1 Zippers

Incidentally, derivatives are intimately connected with *zippers* [12], which represent a data structure with a single subterm highlighted as a 'focus'. Concretely,

a zipper is a pair. The first component is the subterm in focus. The second component is the remainder of the data structure, expressed as a sequence of layers, like an onion, innermost layer first; each layer is the one-hole context into which the structure inside fits.

$$\textbf{type}\ \mathit{Zipper}\ f\ a = (\mathit{Mu}\ f\ a, [(a, \Delta f\ (\mathit{Mu}\ f\ a))])\quad \text{-- innermost layer first}$$

To reconstruct the complete data structure from the zipper, we plug subterms into contexts, from the inside out:

$$\begin{aligned}&\mathit{close} :: \mathit{Diff}\ f \Rightarrow \mathit{Zipper}\ f\ a \to \mathit{Mu}\ f\ a\\ &\mathit{close}\ (x, ds) = \mathit{foldl}\ \mathit{glue}\ x\ ds\ \textbf{where}\ \mathit{glue}\ x\ (a, d) = \mathit{In}\ a\ (\mathit{plug}\ (x, d))\end{aligned}$$

For example, consider a little tree t_1 and two surrounding contexts tc_2, tc_3:

$$\begin{aligned}&t_1 :: \mathit{Tree}\ \mathit{Int}\\ &t_1 = \mathit{leaf}\ 4\end{aligned}$$

$$\begin{aligned}&tc_2, tc_3 :: (\mathit{Int}, \Delta\,\mathit{TreeF}\ (\mathit{Tree}\ \mathit{Int}))\\ &tc_2 = (3, \mathit{Inr}\ (\mathit{Inl}\ (\mathit{Unit} :\times: \mathit{Id}\ (\mathit{leaf}\ 5))))\\ &tc_3 = (1, \mathit{Inr}\ (\mathit{Inr}\ (\mathit{Id}\ (\mathit{leaf}\ 2) :\times: \mathit{Unit})))\end{aligned}$$

The tree in Figure 1 can be reconstructed from these: $t = \mathit{close}\ (t_1, [tc_2, tc_3])$.

6 Downwards accumulations

Now, the ancestors of an element in a data structure form a path to that element, and paths are a projection of zippers. They omit the subterm in focus; they also omit all non-ancestors (siblings, great-aunts, etc) of the focus too, so the type parameter to Δf is the unit type. We call such values 'directions', because they state which direction to take in a parent to get to one of its children.

$$\textbf{type}\ \mathit{Dir}\ f = \Delta f\ ()$$

For example, the derivative of *TreeF* is a sum type, and so there are different possible directions for each variant. However, as we have already seen, the left-hand summand of $\Delta\,\mathit{TreeF}$ is *Zero*, indicating that there is no direction you can turn to get to a child of a leaf. The right-hand summand is another sum type, and this yields two possible directions to turn for a child of an internal node.

$$\begin{aligned}&\mathit{left}, \mathit{right} :: \mathit{Dir}\ \mathit{TreeF}\\ &\mathit{left}\ \ = \mathit{Inr}\ (\mathit{Inl}\ (\mathit{Unit} :\times: \mathit{Id}\ ()))\\ &\mathit{right} = \mathit{Inr}\ (\mathit{Inr}\ (\mathit{Id}\ () :\times: \mathit{Unit}))\end{aligned}$$

Each node of a $\mathit{Mu}\ f\ a$ data structure has a label of type a and an f-structure of children; so a path to a node (including the root label of the target node) consists of an alternating sequence of elements a and directions $\mathit{Dir}\ f$, with one more element than direction. We represent this sequence innermost-first, so that the path to a child has as a subterm the path to its parent.

$$\textbf{data}\ \mathit{Path}\ f\ a = \mathit{Start}\ a \mid \mathit{Step}\ (\mathit{Path}\ f\ a)\ (\mathit{Dir}\ f)\ a$$

For example, the path to the node labelled 4 in t starts with a 1, turns right to meet a 3, then turns left to meet a 4, so it is represented by the expression $\mathit{Step}\ (\mathit{Step}\ (\mathit{Start}\ 1)\ \mathit{right}\ 3)\ \mathit{left}\ 4$.

There is, of course, a natural pattern of folds for paths:

$$\begin{aligned}&\mathit{foldPath} :: (b \to \mathit{Dir}\ f \to a \to b) \to (a \to b) \to \mathit{Path}\ f\ a \to b\\&\mathit{foldPath}\ f\ g\ (\mathit{Step}\ p\ d\ a) = f\ (\mathit{foldPath}\ f\ g\ p)\ d\ a\\&\mathit{foldPath}\ f\ g\ (\mathit{Start}\ a) \quad\ = g\ a\end{aligned}$$

The function *paths* takes a data structure and labels every node with the path from the root to that node:

$$\mathit{paths} :: (\mathit{Diff}\ f, \mathit{Functor}\ (\Delta f)) \Rightarrow \mathit{Mu}\ f\ a \to \mathit{Mu}\ f\ (\mathit{Path}\ f\ a)$$

The definition will have to involve an accumulating parameter: the paths to children depend on surrounding context (the 'path so far') as well as the children themselves. It seems sweetly reasonable to start by unpacking the data structure $\mathit{In}\ a\ \mathit{ts}$ using *posns*, labelling every child t in ts with its one-hole context. We therefore require an auxilliary function paths' that should depend on a path so far, initially simply $\mathit{Start}\ a$, and apply to both a child and its one-hole context:

$$\begin{aligned}&\mathit{paths}\ (\mathit{In}\ a\ \mathit{ts}) = \mathit{In}\ p\ (\mathit{fmap}\ (\mathit{paths}'\ p)\ (\mathit{posns}\ \mathit{ts}))\ \mathbf{where}\ p = \mathit{Start}\ a\\&\mathit{paths}' :: (\mathit{Diff}\ f, \mathit{Functor}\ (\Delta f)) \Rightarrow\\&\qquad \mathit{Path}\ f\ a \to (\mathit{Mu}\ f\ a, \Delta f\ (\mathit{Mu}\ f\ a)) \to \mathit{Mu}\ f\ (\mathit{Path}\ f\ a)\end{aligned}$$

From here, the remainder of the definition is basically driven by the types. We turn the one-hole context z into a direction by discarding siblings, and use that and the node label to extend the path so far for recursive calls.

$$\begin{aligned}&\mathit{paths}'\ p\ (\mathit{In}\ a\ \mathit{ts}, z) = \mathit{In}\ q\ (\mathit{fmap}\ (\mathit{paths}'\ q)\ (\mathit{posns}\ \mathit{ts}))\\&\quad \mathbf{where}\ q = \mathit{Step}\ p\ (\mathit{fmap}\ \mathit{bang}\ z)\ a\end{aligned}$$

Here, *bang* is a basic combinator for the unit type:

$$\begin{aligned}&\mathit{bang} :: a \to ()\\&\mathit{bang}\ a = ()\end{aligned}$$

Figure 3(a) shows $\mathit{paths}\ t$, where t is the tree in Figure 1.

A downwards accumulation is then a fold mapped over the paths:

$$\begin{aligned}&\mathit{scand} :: (\mathit{Diff}\ f, \mathit{Functor}\ (\Delta f)) \Rightarrow\\&\qquad (b \to \mathit{Dir}\ f \to a \to b) \to (a \to b) \to \mathit{Mu}\ f\ a \to \mathit{Mu}\ f\ b\\&\mathit{scand}\ f\ g = \mathit{fmap}\ (\mathit{foldPath}\ f\ g) \circ \mathit{paths}\end{aligned}$$

But because we carefully arranged that the paths to children share a subterm with the path to their parent, we can compute this incrementally in linear time, assuming that the basic operations take constant time. Again, we use an accumulating parameter; but this time, it will be the image under $\mathit{foldPath}\ f\ g$ of the path so far rather than the path itself.

$$\mathit{scand}\ f\ g\ (\mathit{In}\ a\ \mathit{ts}) = \mathit{In}\ b\ (\mathit{fmap}\ (\mathit{scand}'\ f\ g\ b)\ (\mathit{posns}\ \mathit{ts}))\ \mathbf{where}\ b = g\ a$$

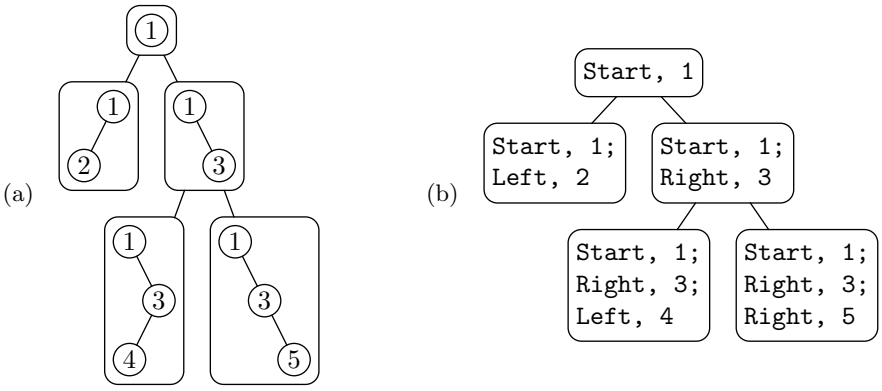

Fig. 3. (a) *paths t* and (b) *routes t* (the latter edited for presentation)

As with *paths*, the remainder of the definition is type-driven: we turn the one-hole context into a direction by discarding siblings, and use that and the node label to update the accumulating parameter for recursive calls.

$$\begin{aligned}
&\mathit{scand}' :: (\mathit{Diff}\ f, \mathit{Functor}\ (\Delta f)) \Rightarrow \\
&\qquad (b \to \mathit{Dir}\ f \to a \to b) \to (a \to b) \to b \to (\mathit{Mu}\ f\ a, \Delta f\ (\mathit{Mu}\ f\ a)) \to \mathit{Mu}\ f\ b \\
&\mathit{scand}'\ f\ g\ b\ (\mathit{In}\ a\ \mathit{ts}, z) = \mathit{In}\ c\ (\mathit{fmap}\ (\mathit{scand}'\ f\ g\ c)\ (\mathit{posns}\ \mathit{ts})) \\
&\quad \textbf{where}\ c = f\ b\ (\mathit{fmap}\ \mathit{bang}\ z)\ a
\end{aligned}$$

(In fact, the second argument g of *scand′* is not used.)

For example, we can produce a little guidebook for a tree, recording in user-friendly format the route to each node in the tree:

$$\begin{aligned}
&\mathit{routes} :: \mathit{Show}\ a \Rightarrow \mathit{Tree}\ a \to \mathit{Tree}\ \mathit{String} \\
&\mathit{routes} = \mathit{scand}\ \mathit{step}\ \mathit{start} \\
&\\
&\mathit{start}\ a = \texttt{"Start, "} +\!\!+ \mathit{show}\ a \\
&\mathit{step}\ s\ (\mathit{Inl}\ z)\ a \qquad\qquad\qquad\qquad\quad = \mathit{magic}\ z \\
&\mathit{step}\ s\ (\mathit{Inr}\ (\mathit{Inl}\ (\mathit{Unit} :\times: \mathit{Id}\ ())))\ a = s +\!\!+ \texttt{"; Left, "} +\!\!+ \mathit{show}\ a \\
&\mathit{step}\ s\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{Id}\ () :\times: \mathit{Unit})))\ a = s +\!\!+ \texttt{"; Right, "} +\!\!+ \mathit{show}\ a
\end{aligned}$$

The first clause for *step* is not really needed, because it can never be called; but its presence makes the definition manifestly total. The results are shown in Figure 3(b) (with the strings edited for presentation).

7 Attribute grammars

Having seen datatype-generic definitions of upwards and downwards accumulations, let us now return to the topic of attribute grammars. These were proposed by Knuth [13] as a tool for presenting the semantics of programming languages. They arose as an extension of the 'syntax-directed' compilation techniques of Irons and others in the early sixties [14]. Using these techniques, the parse tree

of a program is *decorated* with *attributes*, the decoration attached to an element of the parse tree representing some aspect of the semantics of the subtree rooted there. In Irons' formulation, the attribute attached to an element depends only on the descendants of that element; Knuth showed that although no extra power is gained by doing so, the description of the semantics of a language can be considerably simplified by allowing attributes to depend on other parts of the parse tree as well.

Traditionally, an attribute grammar for a context free language is an extension of the grammar which describes the syntax of that language. Each symbol in the grammar is associated with a number of attributes, and each production in the grammar comes with some rules that give values to some of the attributes attached to symbols appearing in that production, in terms of the values of the other attributes that appear. The attributes are classified into two categories, *inherited* and *synthesized*; inherited attributes are those appearing on the right hand side of the production in which their value is defined, and hence concern the 'children' of the production, whereas synthesized attributes appear on the left, and concern the parents. Irons' syntax-directed translation corresponds to attribute grammars with only synthesized attributes. Intuitively, inherited attributes carry information into a subtree and synthesized attributes carry it back out again; in Knuth's [15] words, 'inherited attributes are, roughly speaking, those aspects of meaning which come from the context of a phrase, while synthesized attributes are those aspects which are built up from within the phrase.'

Our view of attribute grammars differs somewhat from this traditional view, and follows instead the approach pioneered by Doaitse and his colleagues [16, 17]. We suppose that a tree has been built already, and that the task is simply to evaluate the attributes. That is, attribute grammars are viewed as a means for describing computations over pre-existing trees, rather than in the 'grammar' sense for recognizing or generating those trees in the first place.

We make the simplification, after [18], that every element has exactly one inherited and one synthesized attribute, and that all inherited attributes have the same type i, and all synthesized attributes the same type s. This entails no loss of generality, since attribute types may be product types.

In our datatype-generic formulation, there is just one datatype constructor *In*, so a single production rule suffices. The production rule takes as input the label (of type a) of a node, the value (of type i) of the sole inherited attribute for that node, and values (collectively of type $f\ s$) for the synthesized attributes of each of the children. It should yield as output the value (of type s) of the sole synthesized attribute of the node, and inherited attributes (collectively of type $f\ i$) for each of the children. We capture this via the following type synonym:

$$\textbf{type}\ \mathit{Rule}\ f\ a\ i\ s = a \to (f\ s, i) \to (s, f\ i)$$

We call a rule $r :: \mathit{Rule}\ f\ a\ i\ s$ 'shape-preserving' if the inherited attributes it generates always match up with the children; that is, for appropriate inputs a, ss, i, if $(s, is) = r\ a\ (ss, i)$ then $\mathit{fmap}\ \mathit{bang}\ is = \mathit{fmap}\ \mathit{bang}\ ss$. We restrict attention in this paper to shape-preserving rules. Of course, 'matching up' will

require the ability to zip together f-structures; we will declare a suitable type class *Zippable* with member function *zip* shortly.

7.1 Lazy evaluation and cyclic programs

The usual formulation of attribute grammars is then to compute the synthesized attribute of the root node of a term, given the inherited attribute. Conventionally a lot of effort goes into deducing the dataflow constraints that arise; for example, maybe an inherited attribute of a right child depends on a synthesized attribute of a left sibling, and so a left-to-right traversal is called for. However, as Johnsson [19] observed, in a lazily evaluated language this can all be ignored: the evaluation mechanism automatically works out the appropriate dataflow. So attribute evaluation can be captured as a simple cyclic lazy functional program:

$$
\begin{array}{l}
\mathit{eval} :: (\mathit{Functor}\ f, \mathit{Zippable}\ f) \Rightarrow \mathit{Rule}\ f\ a\ i\ s \to (\mathit{Mu}\ f\ a, i) \to s \\
\mathit{eval}\ r\ (\mathit{In}\ a\ \mathit{ts}, i) = s \\
\quad \textbf{where} \\
\qquad (s, \mathit{is}) = r\ a\ (\mathit{ss}, i) \\
\qquad \mathit{ss} \quad\ = \mathit{fmap}\ (\mathit{eval}\ r)\ (\mathit{zip}\ \mathit{ts}\ \mathit{is})
\end{array}
$$

Observe that the inherited attributes *is* of the children and their synthesized attributes *ss* are defined using mutual recursion. Statically checking that this recursion is well founded is inherently exponential [20]; indeed, it was one of the first naturally occurring problems to be shown so. In this paper, we assume well-foundedness.

7.2 Matching up

However, one clear necessary condition for well-foundedness is for the shape of *zip ts is* to be determined purely by the shape of *ts*, independent of *is*:

$$\mathit{fmap}\ \mathit{bang}\ (\mathit{zip}\ \mathit{ts}\ \mathit{is}) = \mathit{fmap}\ \mathit{bang}\ (\mathit{zip}\ (\mathit{fmap}\ \mathit{bang}\ \mathit{ts})\ \bot)$$

This is no hardship, because a given *ts* can be zipped successfully with only one shape of *is*. Operationally, we require a datatype-generic *zip* that is non-strict in its second argument. We introduce a type class of zippable functors:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Zippable}\ f\ \textbf{where} \\
\quad \mathit{zip} \quad :: f\ a \to f\ b \to f\ (a, b) \\
\quad \mathit{select} :: f\ a \to \Delta f\ () \to a
\end{array}
$$

We have added an operation *select* too, which takes an f-structure of elements and a position in that structure, and selects the appropriate element; we will use this later. All functors in our universe are zippable:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Zippable}\ \mathit{Unit}\ \textbf{where} \\
\quad \mathit{zip}\ \mathit{Unit}\ \mathit{unit} = \textbf{let}\ \mathit{Unit} = \mathit{unit}\ \textbf{in}\ \mathit{Unit} \\
\quad \mathit{select}\ \mathit{Unit}\ z\ = \mathit{magic}\ z
\end{array}
$$

$$
\textbf{instance}\ \mathit{Zippable}\ \mathit{Id}\ \textbf{where}
$$

$$
\begin{aligned}
&\mathit{zip}\ (\mathit{Id}\ a)\ \mathit{id\_b} \quad = \mathbf{let}\ \mathit{Id}\ b = \mathit{id\_b}\ \mathbf{in}\ \mathit{Id}\ (a, b)\\
&\mathit{select}\ (\mathit{Id}\ a)\ \mathit{Unit} = a
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{instance}\ (\mathit{Zippable}\ f, \mathit{Zippable}\ g) \Rightarrow \mathit{Zippable}\ (\mathit{Sum}\ f\ g)\ \mathbf{where}\\
&\quad \mathit{zip}\ (\mathit{Inl}\ x)\ \mathit{inl\_y}\ = \mathbf{let}\ \mathit{Inl}\ y = \mathit{inl\_y}\ \mathbf{in}\ \ \mathit{Inl}\ (\mathit{zip}\ x\ y)\\
&\quad \mathit{zip}\ (\mathit{Inr}\ x)\ \mathit{inr\_y} = \mathbf{let}\ \mathit{Inr}\ y = \mathit{inr\_y}\ \mathbf{in}\ \mathit{Inr}\ (\mathit{zip}\ x\ y)\\
&\quad \mathit{select}\ (\mathit{Inl}\ x)\ (\mathit{Inl}\ d)\ = \mathit{select}\ x\ d\\
&\quad \mathit{select}\ (\mathit{Inr}\ y)\ (\mathit{Inr}\ d) = \mathit{select}\ y\ d
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{instance}\ (\mathit{Zippable}\ f, \mathit{Zippable}\ g) \Rightarrow \mathit{Zippable}\ (\mathit{Prod}\ f\ g)\ \mathbf{where}\\
&\quad \mathit{zip}\ (x :\times: y)\ \mathit{prod\_x'\_y'} = \mathbf{let}\ x' :\times: y' = \mathit{prod\_x'\_y'}\ \mathbf{in}\\
&\qquad\qquad\qquad\qquad \mathit{zip}\ x\ x' :\times: \mathit{zip}\ y\ y'\\
&\quad \mathit{select}\ (x :\times: y)\ (\mathit{Inl}\ (d :\times: y'))\ = \mathit{select}\ x\ d\\
&\quad \mathit{select}\ (x :\times: y)\ (\mathit{Inr}\ (x' :\times: d)) = \mathit{select}\ y\ d
\end{aligned}
$$

Note the asymmetry in the definitions of *zip*, to ensure non-strictness in the second argument. Of course, *zip* and *select* are partial functions, because the shapes might not match—specifically in the *Sum* case. To be more precise about typing, we could make both methods return a *Maybe* result. However, if we stick to shape-preserving attribute rules, then we will only ever use these two functions on matching shapes.

7.3 Attribute evaluation as a fold

The definition of *eval* above is not obviously in a structured form, since it uses explicit recursion. However, the only use of *ts* is in recursive calls, so it isn't difficult to rearrange the definition into a fold. Of course, children should be processed using different values for the inherited attributes, so it isn't *eval* itself that is a fold. In fact, as many have observed [21–23, 19, 18], it is *curry eval*, computing from a tree a function of type $i \to s$, that is the fold.

$$
\begin{aligned}
&\mathit{curryeval} :: (\mathit{Functor}\ f, \mathit{Zippable}\ f) \Rightarrow \mathit{Rule}\ f\ a\ i\ s \to \mathit{Mu}\ f\ a \to i \to s\\
&\mathit{curryeval}\ r = \mathit{fold}\ \phi\\
&\quad \mathbf{where}\ \phi\ a\ \mathit{fs}\ i = \mathbf{let}\ (s, \mathit{is}) = r\ a\ (\mathit{ss}, i)\\
&\qquad\qquad\qquad\qquad\quad \mathit{ss} \quad\ = \mathit{fmap}\ (\mathit{uncurry}\ (\$))\ (\mathit{zip}\ \mathit{fs}\ \mathit{is})\\
&\qquad\qquad\qquad\ \mathbf{in}\ s
\end{aligned}
$$

Conversely, any fold can be formulated as an attribute grammar using only synthesized attributes. From the algebra ϕ for a fold, we can construct an attribute grammar production rule $\mathit{upRule}\ \phi$ such that $\mathit{curryeval}\ (\mathit{upRule}\ \phi)\ () = \mathit{fold}\ \phi$, using the unit type for inherited attributes:

$$
\begin{aligned}
&\mathit{upRule} :: \mathit{Functor}\ f \Rightarrow (a \to f\ b \to b) \to \mathit{Rule}\ f\ a\ ()\ b\\
&\mathit{upRule}\ \phi\ a\ (\mathit{bs}, ()) = (\phi\ a\ \mathit{bs}, \mathit{fmap}\ \mathit{bang}\ \mathit{bs})
\end{aligned}
$$

7.4 Complete attribute evaluation

Attribute evaluation is conventionally understood to mean evaluation of a single attribute, the synthesized attribute of the root of the tree; all the other attributes

are 'intermediate results' and are of no further interest. For most applications, and in particular for one-off compilation, this is exactly what is required; once the translation of part of a program has been constructed, the translations of subprograms are no longer needed. However, for some applications we want the intermediate results as well; for example, incremental compilers and structure editors such as the Cornell Synthesizer Generator [24] make use of these intermediate results in order to avoid having to recompile parts of a program that remain unchanged. For such applications, we would like attribute evaluation to return the whole tree of attributes, not just the synthesized attribute of the root.

We have seen that the curried evaluation function *curryeval* is a fold; so *fmap curryeval* $\circ$ *subtrees*, yielding a tree of inherited-to-synthesized-attribute functions, is an upwards accumulation. This is not quite enough to allow us to compute all the attributes in the tree: given the inherited attribute of the root, we can certainly find the synthesized attribute of the root, but what will the inherited attributes of the children be? We have thrown that information away.

In fact, to support incremental attribute re-evaluation, we should annotate the input tree to record the values of both the inherited and the synthesized attributes at each node. That can be achieved by a slight modification to *curryeval*, reconstructing the input tree as we go, but retaining the inherited attributes throughout. (For completeness, we also preserve the original labels.)

$$
\begin{array}{l}
\mathit{annotate} :: (\mathit{Functor}\ f, \mathit{Zippable}\ f) \Rightarrow \mathit{Rule}\ f\ a\ i\ s \to \mathit{Mu}\ f\ a \to i \to \mathit{Mu}\ f\ (a, i, s) \\
\mathit{annotate}\ r = \mathit{fold}\ (\psi\ r)\ \textbf{where} \\
\quad \psi\ r\ a\ \mathit{fs}\ i = \mathit{In}\ (a, i, s)\ \mathit{ts}\ \textbf{where} \\
\qquad (s, \mathit{is}) = r\ a\ (\mathit{ss}, i) \\
\qquad \mathit{ts} \quad = \mathit{fmap}\ (\mathit{uncurry}\ (\$))\ (\mathit{zip}\ \mathit{fs}\ \mathit{is}) \\
\qquad \mathit{ss} \quad = \mathit{fmap}\ (\mathit{thd3} \circ \mathit{root})\ \mathit{ts}
\end{array}
$$

(This is roughly the concluding construction in [25].)

An illuminating example of an attribute grammar making essential use of both inherited and synthesized attributes is the ranking problem, in which each node is labelled with its position in left-to-right order. This can be expressed as an attribute grammar with one inherited attribute, representing the first index to use, and two synthesized attributes, recording for each node the rank of that node and the size of (that is, the number of elements in) the subtree rooted at that node. For a binary tree, the inherited attribute passed in to a node is propagated to the left child; but the inherited attribute passed to the right child depends also on the size of the left child. Information flows from left to right, in the same way that it does for a depth-first search. Most of the applications of attribute grammars to programming languages involve dependencies like this, because of the close correspondence between the hierarchical structure of the parse tree and the linear structure of the program text it represents. For our binary tree datatype, we have the following rule:

$$
\begin{array}{l}
\mathit{rankRule} :: \mathit{Rule}\ \mathit{TreeF}\ a\ \mathit{Int}\ (\mathit{Int}, \mathit{Int}) \\
\mathit{rankRule}\ a\ (\mathit{Inl}\ \mathit{Unit}, i) = ((i, 1), \mathit{Inl}\ \mathit{Unit})
\end{array}
$$

$$\begin{aligned}&\mathit{rankRule}\ a\ (\mathit{Inr}\ (\mathit{Id}\ (r_1, s_1) :\times: \mathit{Id}\ (r_2, s_2)), i)\\ &\qquad = ((i{+}s_1, 1{+}s_1{+}s_2), \mathit{Inr}\ (\mathit{Id}\ i :\times: \mathit{Id}\ (i{+}s_1{+}1)))\end{aligned}$$

Then rank ordering is computed by complete attribute evaluation according to this rule, followed by discarding the auxilliary data (the starting index and the size):

$$\begin{aligned}&\mathit{rank} :: \mathit{Tree}\ a \to \mathit{Tree}\ (a, \mathit{Int})\\ &\mathit{rank}\ t = \mathit{fmap}\ (\lambda(a, i, (r, s)) \to (a, r))\ (\mathit{annotate}\ \mathit{rankRule}\ t\ 0)\end{aligned}$$

As another example, consider Bird's 'repmin' problem [26]: replace every element of a tree with the minimum element in that tree. Bird shows a clever circular one-pass solution; as was pointed out soon afterwards, not least (and perhaps first?) by Doaitse himself [27, 19], this circular program precisely corresponds to lazy evaluation of an attribute grammar. Here, the sole inherited attribute is the value with which to replace all labels, and the sole synthesized attribute for a node is the minimum value in the subtree rooted there:

$$\begin{aligned}&\mathit{repminRule} :: \mathit{Ord}\ a \Rightarrow \mathit{Rule}\ \mathit{TreeF}\ a\ a\ a\\ &\mathit{repminRule}\ a\ (\mathit{Inl}\ \mathit{Unit}, m) = (a, \mathit{Inl}\ \mathit{Unit})\\ &\mathit{repminRule}\ a\ (\mathit{Inr}\ (\mathit{Id}\ x :\times: \mathit{Id}\ y), m)\\ &\qquad = (\mathit{min}\ a\ (\mathit{min}\ x\ y), \mathit{Inr}\ (\mathit{Id}\ m :\times: \mathit{Id}\ m))\end{aligned}$$

Solving the repmin problem in a single pass then amounts to complete attribute evaluation according to this rule, followed by discarding the original labels and the synthesized attributes—but ensuring that the initial value of the inherited attribute is the synthesized attribute associated with the root:

$$\begin{aligned}&\mathit{repmin} :: \mathit{Ord}\ a \Rightarrow \mathit{Tree}\ a \to \mathit{Tree}\ a\\ &\mathit{repmin}\ t = \mathbf{let}\ u = \mathit{annotate}\ \mathit{repminRule}\ t\ (\mathit{thd3}\ (\mathit{root}\ u))\ \mathbf{in}\ \mathit{fmap}\ \mathit{snd3}\ u\end{aligned}$$

(where $\mathit{snd3}\ (a, b, c) = b$ and $\mathit{thd3}\ (a, b, c) = c$). The grammar rule is acylic; it is this main program that ties the cyclic knot—u is defined in terms of itself.

7.5 Accumulations as attribute evaluation

Now, complete attribute evaluation in the sense of the previous section generalizes both upwards and downwards accumulations. As we have seen, the algebra for a fold can be expressed as an attribute grammar rule using only synthesized attributes, and evaluating the fold amounts to computing the synthesized attribute of the root node using that rule. As one might therefore expect, any upwards accumulation can be computed by a complete attribute evaluation using the same attribute rule (and then discarding the original labels and the trivial inherited attribute values):

$$\mathit{scanu}\ \phi\ t = \mathit{fmap}\ \mathit{thd3}\ (\mathit{annotate}\ (\mathit{upRule}\ \phi)\ t\ ())$$

Dually, any downwards accumulation can be computed by a complete attribute evaluation using only inherited attributes, using the following rule:

$$
\begin{aligned}
&\mathit{downRule} :: (\mathit{Diff}\ f, \mathit{Functor}\ (\Delta f)) \Rightarrow \\
&\qquad (b \to \mathit{Dir}\ f \to a \to b) \to (a \to b) \to \mathit{Rule}\ f\ a\ (a \to b)\ () \\
&\mathit{downRule}\ f\ g\ a\ (ss, k) = ((), \mathit{fmap}\ (f\ (k\ a) \circ \mathit{fmap}\ \mathit{bang} \circ \mathit{snd})\ (\mathit{posns}\ ss))
\end{aligned}
$$

Note that the inherited attribute for a node can depend only on context outside the tree rooted there, so in particular cannot depend on the node label. Therefore we make the inherited attribute a function from node label to result, and finish off the computation by applying the inherited attributes to the original labels:

$$
\mathit{scand}\ f\ g\ t = \mathit{fmap}\ (\lambda(a, i, ()) \to i\ a)\ (\mathit{annotate}\ (\mathit{downRule}\ f\ g)\ t\ g)
$$

7.6 Attribute evaluations as accumulation

Clearly, there is a strong analogy between complete attribute evaluation and upwards and downwards accumulations: an upwards accumulation is the complete evaluation of an attribute grammar with only synthesized attributes, and a downwards accumulation is the complete evaluation of a grammar with only inherited attributes.

Conversely, it turns out that any complete attribute evaluation can be expressed as an upwards accumulation followed by a downwards accumulation. The idea is to make a first pass over the tree, labelling every node with (the original node label and) a function from the input inherited attribute to the synthesized attribute of that node together with the inherited attributes for the children; then to finish up by composing all the inherited-to-inherited-attribute functions along each of the paths from the root. The first pass is an upwards accumulation:

$$
\begin{aligned}
&\mathit{collect} :: (\mathit{Functor}\ f, \mathit{Zippable}\ f) \Rightarrow \\
&\qquad \mathit{Rule}\ f\ a\ i\ s \to \mathit{Mu}\ f\ a \to \mathit{Mu}\ f\ (a, i \to (s, f\ i)) \\
&\mathit{collect}\ r = \mathit{scanu}\ (\lambda a\ \mathit{afs} \to (a, \phi\ r\ a\ \mathit{afs}))\ \mathbf{where} \\
&\quad \phi\ r\ a\ \mathit{afs}\ i = (s, \mathit{is}')\ \mathbf{where} \\
&\qquad (s, \mathit{is}) = r\ a\ (ss, i) \\
&\qquad \mathit{fis} \quad = \mathit{zip}\ (\mathit{fmap}\ \mathit{snd}\ \mathit{afs})\ \mathit{is} \\
&\qquad ss \quad = \mathit{fmap}\ (\mathit{fst} \circ \mathit{uncurry}\ (\$))\ \mathit{fis} \\
&\qquad \mathit{is}' \quad = \mathit{fmap}\ \mathit{snd}\ \mathit{fis}
\end{aligned}
$$

The second is a downwards accumulation, followed by a map to discard the children's inherited attributes:

$$
\begin{aligned}
&\mathit{distribute} :: (\mathit{Diff}\ f, \mathit{Functor}\ (\Delta f), \mathit{Zippable}\ f) \Rightarrow \\
&\qquad i \to \mathit{Mu}\ f\ (a, i \to (s, f\ i)) \to \mathit{Mu}\ f\ (a, i, s) \\
&\mathit{distribute}\ i\ t = \mathit{fmap}\ \mathit{tidy}\ (\mathit{scand}\ \mathit{step}\ (\mathit{start}\ i)\ t)\ \mathbf{where} \\
&\quad \mathit{step}\ (\_, \_, (\_, \mathit{is}))\ d\ (a, h) = \mathbf{let}\ i = \mathit{select}\ \mathit{is}\ d\ \mathbf{in}\ (a, i, h\ i) \\
&\quad \mathit{start}\ i\ (a, h) \qquad\qquad = (a, i, h\ i) \\
&\quad \mathit{tidy}\ (a, i, (s, \mathit{is})) \qquad\qquad = (a, i, s)
\end{aligned}
$$

Putting them together, we have

$$
\mathit{annotate}\ r\ t\ i = \mathit{distribute}\ i\ (\mathit{collect}\ r\ t)
$$

Incidentally, this pattern of 'collect information upwards through the tree, then redistribute it downwards' is very common; in my thesis [1] and subsequent papers [28, 29] I showed that it also crops up in efficient parallel algorithms for computing prefix sums and in drawing trees.

8 Acknowledgements

The ideas in this paper have been simmering gently at the back of my mind for a long time, and have clearly benefitted from developments from other colleagues (especially Richard Bird and Conor McBride). I am grateful for comments from the Algebra of Programming group at Oxford, for pointing out some errors in drafts. The paper itself was written while I was visiting the National Institute of Informatics in Tokyo, and I would like to thank Zhenjiang Hu for his hospitality.

Last, but not least, I would especially like to thank you, Doaitse, for all the interactions I have had with you over the last twenty years and more: you have always had strong views and strong principles, and have stuck to them tenaciously and with integrity. You have set an uncompromising example for us to follow.

References

1. Gibbons, J.: Algebras for Tree Algorithms. D. Phil. thesis, Programming Research Group, Oxford University (1991) Available as Technical Monograph PRG-94. ISBN 0-902928-72-4.
2. Gibbons, J.: Datatype-generic programming. In Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J., eds.: Spring School on Datatype-Generic Programming. Volume 4719 of Lecture Notes in Computer Science., Springer-Verlag (2007)
3. Gibbons, J.: Origami programming. In Gibbons, J., de Moor, O., eds.: The Fun of Programming. Cornerstones in Computing. Palgrave (2003) 41–60
4. Hagino, T.: A Categorical Programming Language. PhD thesis, Department of Computer Science, University of Edinburgh (1987)
5. Malcolm, G.: Data structures and program transformation. Science of Computer Programming **14** (1990) 255–279
6. Bird, R.S.: An introduction to the theory of lists. In Broy, M., ed.: Logic of Programming and Calculi of Discrete Design. Springer-Verlag (1987) 3–42 Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.
7. Bird, R.S.: Lectures on constructive functional programming. In Broy, M., ed.: Constructive Methods in Computer Science, Springer-Verlag (1988) 151–218 Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University.
8. Bird, R.S.: Algebraic identities for program calculation. Computer Journal **32**(2) (1989) 122–126
9. Bird, R., de Moor, O., Hoogendijk, P.: Generic functional programming with types and relations. Journal of Functional Programming **6**(1) (1996) 1–28
10. McBride, C.: The derivative of a regular type is its type of one-hole contexts. `http://strictlypositive.org/calculus/` (2001)

11. McBride, C.: Clowns to the left of me, jokers to the right: Dissecting data structures. In: Principles of Programming Languages. (2008) 287–295
12. Huet, G.: The zipper. Journal of Functional Programming **7**(5) (1997) 549–554
13. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory **2**(2) (1968) 127–145
14. Irons, E.T.: A syntax directed compiler for Algol 60. Communications of the ACM **4** (1961) 51–55
15. Knuth, D.E.: Examples of formal semantics. In Engeler, E., ed.: Lecture Notes in Mathematics 188: Symposium on Semantics of Algorithmic Languages, Springer-Verlag (1971) 212–235
16. Swierstra, S.D., Azero Alcocer, P.R., Saraiva, J.: Designing and implementing combinator languages. In Swiestra, S., Henriques, P., Oliveira, J., eds.: Advanced Functional Programming, Third International School. Volume 1608 of Lecture Notes in Computer Science., Springer Verlag (1999) 150–206
17. Centre for Software Technology: Utrecht University Attribute Grammar Compiler (UUAGC). `http://www.cs.uu.nl/wiki/bin/view/HUT/AttributeGrammarSystem` (2009)
18. Fokkinga, M., Jeuring, J., Meertens, L., Meijer, E.: A translation from attribute grammars to catamorphisms. The Squiggolist **2**(1) (1991) 20–26
19. Johnsson, T.: Attribute grammars as a functional programming paradigm. In Kahn, G., ed.: LNCS 274: Functional Programming Languages and Computer Architecture, Springer-Verlag (1987) 154–173
20. Jazayeri, M., Ogden, W.F., Rounds, W.C.: The intrinsically exponential complexity of the circularity problem for attribute grammars. Communications of the ACM **18**(12) (1975) 697–706
21. Chirica, L.M., Martin, D.F.: An order-algebraic definition of Knuthian semantics. Mathematical Systems Theory **13**(1) (1979) 1–27
22. Jourdan, M.: Strongly non-circular attribute grammars and their recursive evaluation. In: Symposium on Compiler Construction. (1984) 81–93
23. Katayama, T.: Translation of attribute grammars into procedures. ACM Transactions on Programming Languages and Systems **6**(3) (1984) 345–369
24. Reps, T., Teitelbaum, T.: The Synthesizer Generator—A System for Constructing Language-Based Editors. Springer-Verlag (1989)
25. Jacobs, B., Uustalu, T.: Semantics of grammars and attributes via initiality. In Barendsen, E., Capretta, V., Geuvers, H., Niqui, M., eds.: Reflections on Type Theory, Lambda-Calculus, and the Mind: Essays Dedicated to Henk Barendregt on the Occasion of his 60th Birthday, Radboud Universiteit Nijmegen (2007) 181–196 `http://www.cs.ru.nl/barendregt60/essays/jacobs_uustalu/art15_jacobs_uustalu.pdf`.
26. Bird, R.S.: Using circular programs to eliminate multiple traversals of data. Acta Informatica **21** (1984) 239–250
27. Kuiper, M.F., Swierstra, S.D.: Using attribute grammars to derive efficient functional programs. In: Computing Science in the Netherlands, Amsterdam, SION (1987) 39–52 Also Utrecht University Technical Report RUU-CS86-16, August 1986.
28. Gibbons, J.: Upwards and downwards accumulations on trees. In Bird, R.S., Morgan, C.C., Woodcock, J.C.P., eds.: Mathematics of Program Construction. Volume 669 of Lecture Notes in Computer Science., Springer-Verlag (1993) 122–138
29. Gibbons, J.: Deriving tidy drawings of trees. Journal of Functional Programming **6**(3) (1996) 535–562