10 Left and right folds

The fold on lists, an instance of *foldr*, is

```
> fold :: (a -> b -> b) -> b -> [a] -> b
> fold cons nil    []  = nil
> fold cons nil (x:xs) = cons x (fold cons nil xs)
```

and $\mathit{fold}\ (:)\ [\,] = \mathit{id}$. It captures a possible pattern of computation for many functions on lists

$$
\begin{aligned}
\mathit{sum} &= \mathit{fold}\ (+)\ 0 \\
\mathit{product} &= \mathit{fold}\ (\times)\ 1 \\
\mathit{concat} &= \mathit{fold}\ (+\!\!+)\ [\,] \\
\mathit{map}\ f &= \mathit{fold}\ ((:)\cdot f)\ [\,]
\end{aligned}
$$

notice that none of these equations is recursive: only equations defining *fold* are recursive. We might hope to be able to prove things about the others, such as

$$
\begin{aligned}
\mathit{sum}\ (xs +\!\!+ ys) &= \mathit{sum}\ xs + \mathit{sum}\ ys \\
\mathit{product}\ (xs +\!\!+ ys) &= \mathit{product}\ xs \times \mathit{product}\ ys \\
\mathit{concat}\ (xs +\!\!+ ys) &= \mathit{concat}\ xs +\!\!+ \mathit{concat}\ ys \\
\mathit{map}\ f\ (xs +\!\!+ ys) &= \mathit{map}\ f\ xs +\!\!+ \mathit{map}\ f\ ys
\end{aligned}
$$

without resorting to induction for every one of them.

What is needed is a proof that

$$\mathit{fold}\ c\ n\ (xs +\!\!+ ys) \quad = \quad \mathit{fold}\ c\ n\ xs \oplus \mathit{fold}\ c\ n\ ys$$

Setting out to prove this, just once, will reveal what relationship has to exist between c, n and $(\oplus)$.

$$
\begin{array}{ll}
 & \mathit{fold}\ c\ n\ ([\,] +\!\!+ ys) \\
= & \{\text{definition of } (+\!\!+)\} \\
 & \mathit{fold}\ c\ n\ ys
\end{array}
\qquad\qquad
\begin{array}{ll}
 & \mathit{fold}\ c\ n\ [\,] \oplus \mathit{fold}\ c\ n\ ys \\
= & \{\text{definition of } \mathit{fold}\} \\
 & n \oplus \mathit{fold}\ c\ n\ ys
\end{array}
$$

so these will be equal if $x = n \oplus x$ for all x.

$$
\begin{array}{ll}
 & \mathit{fold}\ c\ n\ ((x:xs) +\!\!+ ys) \\
= & \{\text{definition of } (+\!\!+)\} \\
 & \mathit{fold}\ c\ n\ (x:(xs +\!\!+ ys)) \\
= & \{\text{definition of } \mathit{fold}\} \\
 & c\ x\ (\mathit{fold}\ c\ n\ (xs +\!\!+ ys)) \\
= & \{\text{inductive hypothesis}\} \\
 & c\ x\ (\mathit{fold}\ c\ n\ xs \oplus \mathit{fold}\ c\ n\ ys)
\end{array}
\qquad\qquad
\begin{array}{ll}
 & \mathit{fold}\ c\ n\ (x:xs) \oplus \mathit{fold}\ c\ n\ ys \\
= & \{\text{definition of } \mathit{fold}\} \\
 & c\ x\ (\mathit{fold}\ c\ n\ xs) \oplus \mathit{fold}\ c\ n\ ys
\end{array}
$$

and these will be equal if x ‘c‘ $(y \oplus z) = (x$ ‘c‘ $y) \oplus z$. Furthermore,

$$\begin{array}{ll} & \mathit{fold}\ c\ n\ (\bot +\!\!+\ ys) \\ = & \{\text{definition of } (+\!\!+)\} \\ & \mathit{fold}\ c\ n\ \bot \\ = & \{\mathit{fold} \text{ is strict in the list}\} \\ & \bot \end{array} \qquad \begin{array}{ll} & \mathit{fold}\ c\ n\ \bot \oplus \mathit{fold}\ c\ n\ ys \\ = & \{\mathit{fold} \text{ is strict in the list}\} \\ & \bot \oplus \mathit{fold}\ c\ n\ ys \end{array}$$

and these will be equal if $(\oplus)$ is strict, that is if the operator is strict in its left argument.

None of these three properties involves any recursion so they can be checked by induction-free proofs.

10.1 Fusion

The most generally useful property of folds is that, given the right properties of f, g, h, a, and b,

$$f \cdot \mathit{fold}\ g\ a \quad = \quad \mathit{fold}\ h\ b$$

These are functions of a list so the proof of equality is by induction on an argument list

$$\begin{array}{ll} & (f \cdot \mathit{fold}\ g\ a)\ \bot \\ = & \{\text{definition of } (\cdot)\} \\ & f\ (\mathit{fold}\ g\ a\ \bot) \\ = & \{\mathit{fold} \text{ is strict in the list}\} \\ & f\ \bot \end{array} \qquad \begin{array}{ll} & \mathit{fold}\ h\ b\ \bot \\ = & \{\mathit{fold} \text{ is strict in the list}\} \\ & \bot \end{array}$$

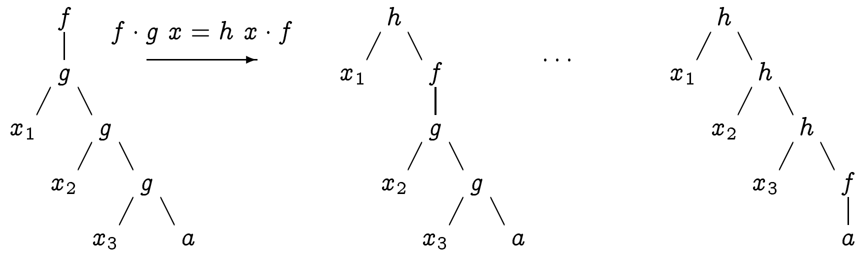
so f must be strict.

$$\begin{array}{ll} & (f \cdot \mathit{fold}\ g\ a)\ [\,] \\ = & \{\text{definition of } (\cdot)\} \\ & f\ (\mathit{fold}\ g\ a\ [\,]) \\ = & \{\text{definition of } \mathit{fold}\} \\ & f\ a \end{array} \qquad \begin{array}{ll} & \mathit{fold}\ h\ b\ [\,] \\ = & \{\text{definition of } \mathit{fold}\} \\ & b \end{array}$$

so $b = f\ a$.

$$
\begin{array}{ll}
 & (f \cdot \mathit{fold}\ g\ a)\ (x : xs) \\
= & \{\text{definition of } (\cdot)\} \\
 & f\ (\mathit{fold}\ g\ a\ (x : xs) \\
= & \{\text{definition of } \mathit{fold}\} \\
 & f\ (g\ x\ (\mathit{fold}\ g\ a\ xs)) \\
= & \{\text{definition of } (\cdot)\} \\
 & (f \cdot g\ x)(\mathit{fold}\ g\ a\ xs)
\end{array}
\qquad
\begin{array}{ll}
 & \mathit{fold}\ h\ b\ (x : xs) \\
= & \{\text{definition of } \mathit{fold}\} \\
 & h\ x\ (\mathit{fold}\ h\ b\ xs) \\
= & \{\text{inductive hypothesis}\} \\
 & h\ x\ ((f \cdot \mathit{fold}\ g\ a)\ xs) \\
= & \{\text{definition of } (\cdot)\text{, twice}\} \\
 & (h\ x \cdot f)\ (\mathit{fold}\ g\ a\ xs)
\end{array}
$$

and these will be equal at least if $h\ x \cdot f = f \cdot g\ x$ or equivalently if $h\ x\ (f\ y) = f\ (g\ x\ y)$.



Most of the laws that we have used that are about functions that are folds have been instances of fusion. We have also been relying on a special case of fusion to show that some function f on lists is a fold, because

$$
\begin{array}{ll}
 & f \\
= & \{\text{unit of composition}\} \\
 & f \cdot \mathit{id} \\
= & \{\text{fold of constructors}\} \\
 & f \cdot \mathit{fold}\ (:)\ [\,] \\
= & \{\text{fusion}\} \\
 & \mathit{fold}\ h\ (f\ [\,])
\end{array}
$$

provided f is strict, and $f\ (x : xs) = h\ x\ (f\ xs)$.

10.2 Left and right folds

One intuition about *fold* is that it produces a right-heavy expression where the arguments replace the constructors of a list:

$$\mathit{fold}\ (\oplus)\ e\ [x_0, x_1, x_2, \ldots x_n] \quad = \quad (x_0 \oplus (x_1 \oplus (x_2 \oplus \cdots (x_n \oplus e) \cdots)))$$

There is a predefined function *foldr* which when restricted to lists agrees with *fold*. We might compute a similar left-heavy expression

$$\mathit{loop}\ (\oplus)\ e\ [x_0, x_1, x_2, \ldots x_n] \quad = \quad (\cdots(((e \oplus x_0) \oplus x_1) \oplus x_2) \oplus \cdots x_n)$$

and might specify this by

$$loop\ s\ n \quad = \quad fold\ (flip\ s)\ n \cdot reverse$$

and calculate from this that it is strict; that

$$\begin{aligned}
& loop\ s\ n\ [\,] \\
= \quad & \{\text{specification}\} \\
& (fold\ (flip\ s)\ n \cdot reverse)\ [\,] \\
= \quad & \{\text{composition}\} \\
& fold\ (flip\ s)\ n\ (reverse\ [\,]) \\
= \quad & \{\text{definition of } reverse\} \\
& fold\ (flip\ s)\ n\ [\,] \\
= \quad & \{\text{definition of } fold\} \\
& n
\end{aligned}$$

and that

$$\begin{aligned}
& loop\ s\ n\ (x : xs) \\
= \quad & \{\text{specification}\} \\
& (fold\ (flip\ s)\ n \cdot reverse)\ (x : xs) \\
= \quad & \{\text{composition}\} \\
& fold\ (flip\ s)\ n\ (reverse\ (x : xs)) \\
= \quad & \{\text{definition of } reverse\} \\
& fold\ (flip\ s)\ n\ (reverse\ xs \mathbin{+\!\!+} [x]) \\
= \quad & \{\text{lemma (exercise 10.1 or 10.2)}\} \\
& fold\ (flip\ s)\ (fold\ (flip\ s)\ n\ [x])\ (reverse\ xs) \\
= \quad & \{\text{definition of } fold\text{, twice}\} \\
& fold\ (flip\ s)\ (flip\ s\ x\ n)\ (reverse\ xs) \\
= \quad & \{\text{definition of } flip\} \\
& fold\ (flip\ s)\ (s\ n\ x)\ (reverse\ xs) \\
= \quad & \{\text{composition}\} \\
& (fold\ (flip\ s)\ (s\ n\ x) \cdot reverse)\ xs \\
= \quad & \{\text{specification}\} \\
& loop\ s\ (s\ n\ x)\ xs
\end{aligned}$$

This justifies defining

```
> loop s n     []  = n
> loop s n (x:xs) = loop s (s n x) xs
```

and this is essentially the same as the predefined *foldl* (restricted to lists).

10.3 Scans

One commonly needs to think about 'partial sums'. The natual thing to think of first for lists is

$$scan\ c\ n \quad = \quad map\ (fold\ c\ n) \cdot tails$$

where the tails of a list are all the suffix segments, in decreasing order of length.

```
> tails :: [a] -> [[a]]
> tails    []  = [[]]
> tails (x:xs) = (x:xs): tails xs
```

There is a very similar standard function *tails* in `Data.List`.

In the way you probably expect, *tails* can be cast as a *fold* because

$$\begin{array}{ll} & tails\ (x : xs) \\ = & \{\text{definition of } tails\} \\ & (x : xs) : tails\ xs \\ = & \{ head\ (tails\ xs) = xs \} \\ & (x : head\ ys) : ys\ \textbf{where}\ ys = tails\ xs \end{array}$$

so $tails = fold\ g\ [[\,]]$ **where** $g\ x\ ys = (x : head\ ys) : ys$. This means that if the conditions of fusion are satisfied, *scan* can be expressed as a *fold*.

Firstly, $map\ (fold\ c\ n)$ is strict; then

$$\begin{array}{ll} & map\ (fold\ c\ n)\ [[\,]] \\ = & \{\text{definition of } map\} \\ & [fold\ c\ n\ [\,]] \\ = & \{\text{definition of } fold\} \\ & [n] \end{array}$$

and then

$$\begin{array}{ll} & map\ (fold\ c\ n)\ (g\ x\ ys) \\ = & \{\text{definition of } g\} \\ & map\ (fold\ c\ n)\ ((x : head\ ys) : ys) \\ = & \{\text{definition of } map\} \\ & (fold\ c\ n\ (x : head\ ys) : map\ (fold\ c\ n)\ ys \\ = & \{\text{definition of } fold\} \\ & c\ x\ (fold\ c\ n\ (head\ ys)) : map\ (fold\ c\ n)\ ys \\ = & \{ f \cdot head = head \cdot map\ f \} \\ & c\ x\ (head\ zs) : zs\ \textbf{where}\ zs = map\ (fold\ c\ n)\ ys \end{array}$$

from which conclude that

$$
\begin{aligned}
& \mathit{scan}\ c\ n \\
= \;& \{\,\text{specification}\,\} \\
& \mathit{map}\ (\mathit{fold}\ c\ n) \cdot \mathit{tails} \\
= \;& \{\,\text{fusion}\,\} \\
& \mathit{fold}\ h\ [n]\ \textbf{where}\ h\ x\ \mathit{zs} = c\ x\ (\mathit{head}\ \mathit{zs}) : \mathit{zs}
\end{aligned}
$$

Notice that executing the specification directly gives a quadratic algorithm: for a list xs of length n there are about $\frac{1}{2}n^2$ applications of c. However there are only n applications of h, each of which calls c exactly once (and does a constant amount of consing and unconsing). The result is a linear algorithm for

```
> scan c n = fold h [n] where h x zs = c x (head zs):zs
```

The predefined function *scanr* is equal to *scan*, and even has the same strictness.

10.4 Aside: the names of *fold* and *loop*

In text books you will find *fold* being called *foldr*. It would have been natural for the arguments to be in the same order as the constructors appear in the type of lists, but the name and the argument order of *foldr* have been the same at least since David Turner's *SASL* in the mid 1970s. The first argument has also been called f and the second either r or e, and you will find *foldr* f e in textboooks. My use of c and n is for menmonic reasons.

Similarly, my *loop* s n appears as *foldl* f e in texts. I used to call it *tailfold* because it is a tail call, but justifying calling it a fold is initially harder. It is (exercise 10.4) the fold on lists built with *snoc* constructors that add a last element to a list, hence my use of s for its argument.

10.5 Aside: strictness

The function *tails* defined above is strict, but *Data.List.tails* is not. However the implementation of *scan* as a fold is strict (as is the predefined *scanr* which is equal to *scan*), because folds are strict.

Had we defined *tails* to be non-strict,

```
> tails xs = xs : if null xs then [] else tails (tail xs)
```

it would not have been possible to implement it by a fold. The rest of the derivation of the implementation of *scan* as a fold is sound.

You might argue that the efficient implementation of *scan* is not a faithful implementation of $\mathit{map}\ (\mathit{fold}\ c\ n) \cdot \mathit{tails}$ if *tails* is not strict.

10.6 Aside: fusion and computability

The statement of the fold fusion law $f \cdot fold\ g\ a = fold\ h\ b$ was proved by showing that $f\ (fold\ g\ a\ xs) = fold\ h\ b\ xs$ by induction on xs, but no mention was made in section 10.1 of chain completeness. In fact the three conditions for fold fusion are only enough to prove this equality for finite or partial xs.

To complete the proof by induction for infinite xs it is necessary to show that the proposition $f\ (fold\ g\ a\ xs) = fold\ h\ b\ xs$ is chain complete with respect to xs.

Fortunately if both sides are computable functions of xs, as they would be if they were Haskell-definable functions, an equation is chain complete.

However an equation proved to be true on all partial lists does not necessarily hold at infinite lists if the two sides are non-computable functions of the list, and fold fusion cannot necessarily be applied to such functions. (See exercise 10.6.)

Exercises

10.1 Prove directly by induction that

$$fold\ c\ n\ (xs +\!\!+\ ys) \quad = \quad fold\ c\ (fold\ c\ n\ ys)\ xs$$

for all lists xs and ys (whether partial, finite or infinite).

10.2 Use fold fusion to show that the section $(+\!\!+\ bs)$ is a fold.

Deduce without resort to induction that

$$fold\ c\ n\ (xs +\!\!+\ ys) \quad = \quad fold\ c\ (fold\ c\ n\ ys)\ xs$$

10.3 Use fold fusion to show that *filter* p is a fold.

Deduce that

$$filter\ p\ (xs +\!\!+\ ys) \quad = \quad filter\ p\ xs +\!\!+\ filter\ p\ ys$$

10.4 A data type very like that of lists might be defined by

```
> data Liste a = Snoc (Liste a) a | Lin
```

There will be elements of $Liste\ \alpha$ and of $[\alpha]$ corresponding to finite lists, for example $Snoc\ (Snoc\ (Snoc\ Lin\ 1)\ 2)\ 3$ corresponds to $1 : (2 : (3 : [\,]))$, that is $[1, 2, 3]$.

Write a recursive definition of a function $cat :: Liste\ \alpha \to Liste\ \alpha \to Liste\ \alpha$ which concatenates two elements of *Liste*.

Define a function *folde* which is the natural fold for $Liste\ \alpha$.

Express *cat* in terms of *folde*.

Define (as folds) functions $list :: Liste\ \alpha \to [\alpha]$ and $liste :: [\alpha] \to Liste\ \alpha$ which express the identification of finite lists represented as elements of $Liste\ \alpha$ and of $[\alpha]$. (That is, they should be mutually inverse on finite lists.)

What does *liste* return when applied to an infinite list? What are the infinite objects of type $Liste\ \alpha$?

Find equivalent definitions of *list* and *liste* as instances of *loop* and the corresponding function for $Liste\ \alpha$.

10.5 Recall that the unfold function for $[\alpha]$

```
> unfold n h t x | n x       = []
>                | otherwise = h x : unfold n h t (t x)
```

yields the identity function *unfold null head tail* when applied to the deconstructors for $[\alpha]$.

Using the same property for the identity of $Liste\ \alpha$ define the unfold function *unfolde* for $Liste\ \alpha$.

Write *list* and *liste* as the appropriate unfolds.

You may want to know that there are predefined functions $init :: [\alpha] \to [\alpha]$ and $last :: [\alpha] \to \alpha$ for which $xs = init\ xs + [last\ xs]$ for all non-null xs. It might help to work out first how these might be defined by recursion.

10.6 A function $f :: [a] \to [a]$ satisfies $f\ xs = xs$ for all infinite lists xs, and $f\ xs = \perp$ otherwise. Show that f is not computable.

The function $(!:)$, read as *tail-strict cons*, is defined so that $x\ !: \perp = \perp$, and $x\ !: xs = x : xs$ otherwise.

Show that whilst f satisfies the three conditions for the corresponding fold fusion,

$$f \cdot fold\ (:)\ [\,] \quad \neq \quad fold\ (!:)\ \perp$$