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

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

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

$$\begin{aligned} & \mathit{fold}\ c\ n\ ([\,] +\!\!+ ys) \\ = & \quad \{\text{definition of } (+\!\!+)\} \\ & \mathit{fold}\ c\ n\ ys \end{aligned}$$

$$\begin{aligned} & \mathit{fold}\ c\ n\ [\,] \oplus \mathit{fold}\ c\ n\ ys \\ = & \quad \{\text{definition of } \mathit{fold}\} \\ & n \oplus \mathit{fold}\ c\ n\ ys \end{aligned}$$

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

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

$$\begin{aligned} & \mathit{fold}\ c\ n\ (x{:}xs) \oplus \mathit{fold}\ c\ n\ ys \\ = & \quad \{\text{definition of } \mathit{fold}\} \\ & c\ x\ (\mathit{fold}\ c\ n\ xs) \oplus \mathit{fold}\ c\ n\ ys \end{aligned}$$

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

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

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 is by induction on an argument list

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

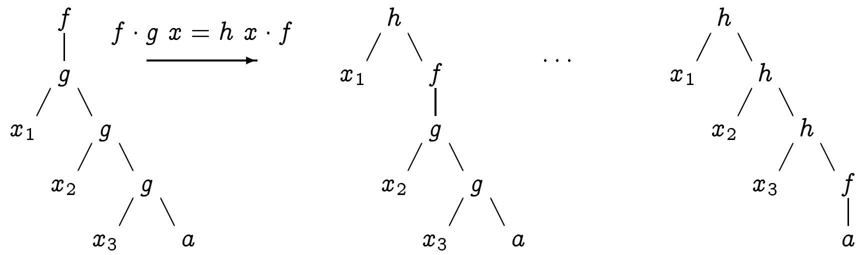
so f must be strict.

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

so $b = f\ a$.

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

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



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

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

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

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

and might specify this by

$$\mathit{tailfold}\ c\ n \quad = \quad \mathit{fold}\ (\mathit{flip}\ c)\ n \cdot \mathit{reverse}$$

and calculate from this that it is strict; that

$$\begin{array}{ll}
 & \mathit{tailfold}\ c\ n\ [\,] \\
= & \{\text{specification}\} \\
 & (\mathit{fold}\ (\mathit{flip}\ c)\ n \cdot \mathit{reverse})\ [\,] \\
= & \{\text{composition}\} \\
 & \mathit{fold}\ (\mathit{flip}\ c)\ n\ (\mathit{reverse}\ [\,]) \\
= & \{\text{definition of } \mathit{reverse}\} \\
 & \mathit{fold}\ (\mathit{flip}\ c)\ n\ [\,] \\
= & \{\text{definition of } \mathit{fold}\} \\
 & n
\end{array}$$

and that

$$\begin{array}{ll}
 & \mathit{tailfold}\ c\ n\ (x : \mathit{xs}) \\
= & \{\text{specification}\} \\
 & (\mathit{fold}\ (\mathit{flip}\ c)\ n \cdot \mathit{reverse})\ (x : \mathit{xs}) \\
= & \{\text{composition}\} \\
 & \mathit{fold}\ (\mathit{flip}\ c)\ n\ (\mathit{reverse}\ (x : \mathit{xs})) \\
= & \{\text{definition of } \mathit{reverse}\} \\
 & \mathit{fold}\ (\mathit{flip}\ c)\ n\ (\mathit{reverse}\ \mathit{xs} +\!\!+ [x]) \\
= & \{\text{lemma (exercise 10.1 or 10.2)}\} \\
 & \mathit{fold}\ (\mathit{flip}\ c)\ (\mathit{fold}\ (\mathit{flip}\ c)\ n\ [x])\ (\mathit{reverse}\ \mathit{xs}) \\
= & \{\text{definition of } \mathit{fold}\text{, twice}\} \\
 & \mathit{fold}\ (\mathit{flip}\ c)\ (\mathit{flip}\ c\ x\ n)\ (\mathit{reverse}\ \mathit{xs}) \\
= & \{\text{definition of } \mathit{flip}\} \\
 & \mathit{fold}\ (\mathit{flip}\ c)\ (c\ n\ x)\ (\mathit{reverse}\ \mathit{xs}) \\
= & \{\text{composition}\} \\
 & (\mathit{fold}\ (\mathit{flip}\ c)\ (c\ n\ x) \cdot \mathit{reverse})\ \mathit{xs} \\
= & \{\text{specification}\} \\
 & \mathit{tailfold}\ c\ (c\ n\ x)\ \mathit{xs}
\end{array}$$

This justifies defining

```
> tailfold c n    []   = n
> tailfold c n (x:xs) = tailfold c (c n x) xs
```

and this is essentially the same as the predefined *foldl* (restricted to lists). The name is justified by the recusion being a *tail call*.

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

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

so $tails = fold\ g\ [[\,]]\ \mathbf{where}\ g\ x\ ys = (x : head\ ys) : ys$. 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

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

from which conclude that

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

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). The result is a linear algorithm for

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

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

10.4 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, ptails :: [a] -> [[a]]
> tails     xs  = xs : ptails xs
> ptails    []  = []
> ptails (_:xs) = tails 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.