12 Some efficiency concerns

Recall that we first defined the *reverse* function by recursion:

$$\begin{aligned} \mathit{reverse}\ [\,] &= [\,] \\ \mathit{reverse}\ (x : xs) &= \mathit{reverse}\ xs \mathbin{+\!\!+} [x] \\ &= \mathit{snoc}\ (\mathit{reverse}\ xs)\ x \\ &= \mathit{flip}\ \mathit{snoc}\ x\ (\mathit{reverse}\ xs) \end{aligned}$$

Deduce from this that

$$\mathit{reverse} \quad = \quad \mathit{fold}\ (\mathit{flip}\ \mathit{snoc})\ [\,]\ \textbf{where}\ \mathit{snoc}\ xs\ x = xs \mathbin{+\!\!+} [x]$$

However, this algorithm is quadratic: it takes about $\frac{1}{2}n^2$ steps to reverse a list of length n. Why is this? Each catenation

$$\begin{aligned} [\,] \mathbin{+\!\!+} ys &= ys \\ (x : xs) \mathbin{+\!\!+} ys &= x : (xs \mathbin{+\!\!+} ys) \end{aligned}$$

(or $(\mathbin{+\!\!+} ys) = \mathit{fold}\ (:)\ ys$) takes a number of steps linear in the length of its left argument. It follows that *snoc* takes a number of steps linear in its list argument, and *reverse* applies *snoc* to (the reverse of) each tail of its argument.

The insight is that we could accumulate the answer: invent

$$\mathit{revcat}\ ys\ xs \quad = \quad \mathit{reverse}\ xs \mathbin{+\!\!+} ys$$

Notice that this is intended as a specfication, not the definition for execution: evaluating this would be at least as bad as the existing *reverse*. We could however use *revcat* to calculate

$$\begin{aligned} & \mathit{reverse}\ xs \\ = \ & \{\, \text{unit of } (\mathbin{+\!\!+})\ (\text{proof?}) \,\} \\ & \mathit{reverse}\ xs \mathbin{+\!\!+} [\,] \\ = \ & \{\, \text{specification of } \mathit{revcat} \,\} \\ & \mathit{revcat}\ [\,]\ xs \end{aligned}$$

and then make the $(\mathbin{+\!\!+})$ vanish, by synthesizing a $(\mathbin{+\!\!+})$-less recursive definition of *revcat*

$$\begin{aligned} & \mathit{revcat}\ ys\ [\,] \\ = \ & \{\, \text{specification of } \mathit{revcat} \,\} \\ & \mathit{reverse}\ [\,] \mathbin{+\!\!+} ys \\ = \ & \{\, \text{definition of } \mathit{reverse} \,\} \\ & [\,] \mathbin{+\!\!+} ys \\ = \ & \{\, \text{definition of } (\mathbin{+\!\!+}) \,\} \\ & ys \end{aligned}$$

and for non-empty lists

$$
\begin{aligned}
& \quad \mathit{revcat}\ \mathit{ys}\ (x : \mathit{xs}) \\
= & \quad \{ \text{specification of } \mathit{revcat} \} \\
& \quad \mathit{reverse}\ (x : \mathit{xs}) \mathbin{+\!\!+} \mathit{ys} \\
= & \quad \{ \text{definition of } \mathit{reverse} \} \\
& \quad (\mathit{reverse}\ \mathit{xs} \mathbin{+\!\!+} [x]) \mathbin{+\!\!+} \mathit{ys} \\
= & \quad \{ \text{associativity of } (\mathbin{+\!\!+}) \} \\
& \quad \mathit{reverse}\ \mathit{xs} \mathbin{+\!\!+} ([x] \mathbin{+\!\!+} \mathit{ys}) \\
= & \quad \{ \text{definition of } (\mathbin{+\!\!+}) \} \\
& \quad \mathit{reverse}\ \mathit{xs} \mathbin{+\!\!+} (x : \mathit{ys}) \\
= & \quad \{ \text{specification of } \mathit{revcat} \} \\
& \quad \mathit{revcat}\ (x : \mathit{ys})\ \mathit{xs}
\end{aligned}
$$

This gives us a definition

```
> reverse = revcat []
>           where revcat ys    []  = ys
>                 revcat ys (x:xs) = revcat (x:ys) xs
```

This one is linear in the length of the list being reversed: each call of *revcat* corresponds to one of the conses in the list, and each call does a constant amount of work before the recursive call.

The correspondence between conses and calls of *revcat* suggests that we think of a fold, but it is not a *fold*. Compare it with

$$
\begin{aligned}
\mathit{tailfold}\ s\ n\ [\,] &= n \\
\mathit{tailfold}\ s\ n\ (x : \mathit{xs}) &= \mathit{tailfold}\ s\ (s\ n\ x)\ \mathit{xs}
\end{aligned}
$$

and by inspection $\mathit{revcat}\ \mathit{ys} = \mathit{tailfold}\ (\mathit{flip}\ (:))\ \mathit{ys}$ so

$$\mathit{reverse} = \mathit{tailfold}\ (\mathit{flip}\ (:))\ [\,]$$

12.1 Flattening trees

The flatten function for

```
> data BTree a = Leaf a | Fork (BTree a) (BTree a)
```

is $\mathit{flatten} :: \mathit{BTree}\ \alpha \to [\alpha]$ for which

$$
\begin{aligned}
\mathit{flatten}\ (\mathit{Leaf}\ x) &= [x] \\
\mathit{flatten}\ (\mathit{Fork}\ \mathit{ls}\ \mathit{rs}) &= \mathit{flatten}\ \mathit{ls} \mathbin{+\!\!+} \mathit{flatten}\ \mathit{rs}
\end{aligned}
$$

The length of the result is the number of leaves in the tree, the size of the tree

$$size \quad = \quad foldBTree\ (const\ 1)\ (+)$$

however in general it takes more steps than that to produce it.

For a balanced tree of size n there will be a $(+\!\!+)$ at the root that takes about $\frac{1}{2}n$ steps, below that two that take $\frac{1}{4}n$ steps each, and so on, which amounts to about $\frac{1}{2}n \log n$ steps. If the tree has a long left spine, the algorithm can be as bad as quadratic.

As before the insight is that we should specify

$$flatcat\ t\ ys \quad = \quad flatten\ t +\!\!+ ys$$

and synthesize

$$\begin{array}{cl}
 & flatcat\ (Leaf\ x)\ ys \\
= & \{\text{specification of } flatcat\} \\
 & flatten\ (Leaf\ x) +\!\!+ ys \\
= & \{\text{definition of } flatten\} \\
 & [x] +\!\!+ ys \\
= & \{\text{definition of } (+\!\!+)\} \\
 & x : ys
\end{array}$$

and

$$\begin{array}{cl}
 & flatcat\ (Fork\ ls\ rs)\ ys \\
= & \{\text{specification of } flatcat\} \\
 & flatten\ (Fork\ ls\ rs) +\!\!+ ys \\
= & \{\text{definition of } flatten\} \\
 & (flatten\ ls +\!\!+ flatten\ rs) +\!\!+ ys \\
= & \{\text{associativity of } (+\!\!+)\} \\
 & flatten\ ls +\!\!+ (flatten\ rs +\!\!+ ys) \\
= & \{\text{specification of } flatcat\} \\
 & flatcat\ ls\ (flatcat\ rs\ ys)
\end{array}$$

so $flatcat = foldBTree\ (:)\ (\cdot)$ and $flatten\ t = foldBTree\ (:)\ (\cdot)\ t\ [\,]$ and, relying on the associativity of $(+\!\!+)$, synthesis has produced a linear algorithm from a less efficient one.

12.2 Associativity and folds

When is $fold\ (\oplus)\ e = tailfold\ (\otimes)\ f$?

Suppose we try to prove this by induction. It is chain complete, and both sides are strict. Applying both sides to $[\,]$ shows that it is necessary that $e = f$. The substantial part of the proof is

$$
\begin{aligned}
& \mathit{fold}\ (\oplus)\ e\ (x : xs) \\
= \quad & \{\text{definition of } \mathit{fold}\} \\
& x \oplus \mathit{fold}\ (\oplus)\ e\ xs \\
= \quad & \{\text{lemma to be proved}\} \\
& \mathit{tailfold}\ (\otimes)\ (e \otimes x)\ xs \\
= \quad & \{\text{definition of } \mathit{tailfold}\} \\
& \mathit{tailfold}\ (\otimes)\ e\ (x : xs)
\end{aligned}
$$

The essence of the result is the missing lemma, again to be proved by induction.

The assertion to be proved is chain complete. If $xs = \bot$ conclude that $x \oplus \bot = \bot$ for all x, so $(\oplus)$ must be strict in its second argument. If $xs = [\,]$ conclude that $e \otimes x = x \oplus e$. The substantial part of the proof of the lemma is

$$
\begin{aligned}
& \mathit{tailfold}\ (\otimes)\ (e \otimes x)\ (y : ys) \\
= \quad & \{\text{definition of } \mathit{tailfold}\} \\
& \mathit{tailfold}\ (\otimes)\ ((e \otimes x) \otimes y)\ ys \\
= \quad & \{\text{suppose } (a \otimes b) \otimes c = a \otimes (b \odot c)\} \\
& \mathit{tailfold}\ (\otimes)\ (e \otimes (x \odot y))\ ys \\
= \quad & \{\text{induction hypothesis}\} \\
& (x \odot y) \oplus \mathit{fold}\ (\oplus)\ e\ ys \\
= \quad & \{\text{suppose } (a \odot b) \oplus c = a \oplus (b \oplus c)\} \\
& x \oplus (y \oplus \mathit{fold}\ (\oplus)\ e\ ys) \\
= \quad & \{\text{definition of } \mathit{fold}\} \\
& x \oplus \mathit{fold}\ (\oplus)\ e\ (y : ys)
\end{aligned}
$$

Notice that this is a proof for all values of x, and the induction hypothesis is that it holds for all x and a specific ys.

Collecting the requrements:

$$\mathit{fold}\ (\oplus)\ e \quad = \quad \mathit{tailfold}\ (\otimes)\ e$$

is proved for right-strict $(\oplus)$, provided $e \otimes x = x \oplus e$ and provided there is a $(\odot)$ for which $a \otimes (b \odot c) = (a \otimes b) \otimes c$ and $(a \odot b) \oplus c = a \oplus (b \oplus c)$.

The obvious case is when all three of $(\oplus)$, $(\otimes)$ and $(\odot)$ are equal, are right-strict, are associative, and have e as a left and right unit.

$$
\begin{aligned}
sum &= fold\ (+)\ 0 \\
&= tailfold\ (+)\ 0 \\
product &= fold\ (\times)\ 1 \\
&= tailfold\ (\times)\ 1 \\
concat &= fold\ (+\!\!+)\ [\,] \\
&\neq tailfold\ (+\!\!+)\ [\,]
\end{aligned}
$$

this inequality is because $xs +\!\!+ \perp \neq \perp$. The *fold* form produces output when applied to an infinite list of lists provided enough of them are non-empty, but the *tailfold* form cannot produce any output for an infinite (or partial) input.

12.3 Bounding space

One reason for preferring *tailfold* $(+)\ 0$ to *fold* $(+)\ 0$ is that the *fold* is generally obliged to build up the whole expression before any evaluation:

$$
\begin{aligned}
& fold\ (+)\ 0\ [1,2,3,4] \\
= {} & 1 + fold\ (+)\ 0\ [2,3,4] \\
= {} & 1 + (2 + fold\ (+)\ 0\ [3,4]) \\
= {} & 1 + (2 + (3 + fold\ (+)\ 0\ [4])) \\
= {} & 1 + (2 + (3 + (4 + fold\ (+)\ 0\ [\,]))) \\
= {} & 1 + (2 + (3 + (4 + 0))) \\
= {} & 1 + (2 + (3 + 4)) \\
= {} & 1 + (2 + 7) \\
= {} & 1 + 9 \\
= {} & 10
\end{aligned}
$$

whreas the *tailfold* can be seen as evaluating the expression as it goes. In fact, because of lazy evaluation

$$
\begin{aligned}
& tailfold\ (+)\ 0\ [1,2,3,4] \\
= {} & tailfold\ (+)\ (0+1)\ [2,3,4] \\
= {} & tailfold\ (+)\ ((0+1)+2)\ [3,4] \\
= {} & tailfold\ (+)\ (((0+1)+2)+3)\ [4] \\
= {} & tailfold\ (+)\ ((((0+1)+2)+3)+4)\ [\,] \\
= {} & (((0+1)+2)+3)+4 \\
= {} & ((1+2)+3)+4 \\
= {} & (3+3)+4 \\
= {} & 6+4 \\
= {} & 10
\end{aligned}
$$

the same space build-up can happen.

To prevent it, *tailfold* would have to be made strict in this argument.

```
> tailfold' s n [] = n
> tailfold' s n (x:xs) = let !snx = s n x in tailfold' s snx xs
```

The ! decoration ensures that the variable *snx* is evaluated before the recurive call.

12.4 Fast exponentiation

On the face of it, calculating x^n appears to require about n multiplication. But multiplication is associative, so $x^{2n} = (x^2)^n$ and x^{2n} can be calculated in only one more multiplication than x^n. So we could specify $pow\ x\ n = x^n$ and synthesize

```
pow x 0 = 1
pow x n | even n = pow (x*x) (n`div`2)
        | odd n  = pow x (n-1) * x
```

This function will be called no more than $2 \log n$ times in x^n.

However, just like the *fold* version of *product*, this function will unnecessarily build up a big expression before any evaluation. Specify $power\ y\ x\ n = pow\ x\ n \times y$ and synthesize

$$\begin{aligned}
power\ y\ x\ 0 &= pow\ x\ 0 \times y \\
&= 1 \times y \\
&= y \\
power\ y\ x\ n \mid even\ n &= pow\ x\ n \times y \\
&= pow\ (x \times x)\ (n\ \mathbf{div}\ 2) \times y \\
&= power\ y\ (x \times x)\ (n\ \mathbf{div}\ 2) \\
power\ y\ x\ n \mid odd\ n &= pow\ x\ n \times y \\
&= (pow\ x\ (n-1) \times x) \times y \\
&= pow\ x\ (n-1) \times (x \times y) \\
&= power\ (x \times y)\ x\ (n-1)
\end{aligned}$$

We could also abstract on the multiplication:

```
> power (*) y x n -- x^n*y
>       | n == 0 = y
>       | even n = power (*) y (x*x) (n`div`2)
>       | odd n  = power (*) (x*y) x (n-1)
```

Notice that the development of this code used only the associativity of $(\times)$, so it will calculate other repeated operations such as repeated matrix multiplication.