12 Some efficiency concerns

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

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

Deduce from this that

$$reverse \;=\; fold\ (flip\ snoc)\ [\,]\ \textbf{where}\ snoc\ xs\ x = xs + [x]$$

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

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

(or $(+ys) = fold\ (:)\ ys$) 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

$$revcat\ ys\ xs \;=\; reverse\ xs + ys$$

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

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

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

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

and for non-empty lists

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

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* on cons-lists. Compare it with

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

(which is *foldl*, the fold on snoc-lists) and by inspection $\mathit{revcat} = \mathit{loop}\ (\mathit{flip}\ (:))$ so

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

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.

To flatten 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 to eliminate the $(++)$ we should specify

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

and synthesize

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

and

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

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

12.2 Associativity and folds

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

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

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

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

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

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

Notice that this is a proof for all values of x, and the induction hypothesis is that it holds for a particular ys and all values in the x position, in particular $(x \odot y)$.

Collecting the requirements:

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

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.

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

This last inequality is possible because $xs +\!\!+ \perp \neq \perp$. The *fold* form produces output when applied to an infinite list of lists provided at least one of them is non-empty, but the *loop* form cannot produce any output for an infinite (or partial) input.

12.3 Bounding space

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

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

whereas the *loop* can safely evaluate the expression as it goes. In practice, because of lazy evaluation

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

the same space build-up can happen. To prevent it, *loop* would have to be made strict in this argument.

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

The ! decoration ensures that the argument is evaluated before the recursive call. (This decoration is now a language extension in Haskell and requires a flag, or the pragma `{-# LANGUAGE BangPatterns #-}` at the top of a script.)

12.4 Fast exponentiation

On the face of it, calculating x^n appears to require about n multiplications. 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 must 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.

Exercises

12.1 A *queue* is a data type with (at least) four operations

```
> empty   :: Queue a
> isEmpty :: Queue a -> Bool
> add     :: a -> Queue a -> Queue a
> get     :: Queue a -> (a, Queue a)
```

The value of *empty* is a queue with nothing in it; a queue satisfies *isEmpty* if all of the values that have been added to it have already been removed; *add* puts a value into a queue; and *get* returns the oldest value still waiting in the queue, along with a queue from which just that value has been removed.

Implement a queue type using a list of the elements in the queue in the order in which they joined. That is, give a declaration of the *Queue* type, and implement each of these four functions.

Estimate roughly how expensive your operations are. Would your answer be any different if the queue were represented by a list of its remaining elements in the reverse of the order in which they join the queue?

Reimplement the *Queue* using two lists of elements, *front* and *back* so that the elements in the queue are those in the list $\mathit{front} +\!\!+ \mathit{reverse}\ \mathit{back}$. What effect does this have on the cost of the operations?

12.2 The Fibonacci sequence

```
> fib 0 = 0
> fib 1 = 1
> fib n = fib (n-1) + fib (n-2)
```

grows very quickly (each value is about 1.6 times bigger than its predecessor).

Use this definition in a GHCi script and try evaluating *fib* 10, *fib* 20 and *fib* 30. Give a brief explanation of why the later calls are so slow.

Let $\mathit{two}\ n = (\mathit{fib}\ n,\ \mathit{fib}\ (n+1))$, and synthesize a definition of *two* by direct recursion. Use this to give a more efficient definition of *fib*. How does the time it takes to calculate $\mathit{fib}\ n$ in this way depend on n?

Roughly how big is the 10 000th Fibonacci number? You might want to use

```
> roughly :: Integer -> String
> roughly n = x : 'e' : show (length xs) where x:xs = show n
```

to produce a readable estimate.

Let F be the matrix $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, and F^n be its nth power, the product of n copies of it.

Explain why $F^n = \begin{pmatrix} \mathit{fib}\ (n-1) & \mathit{fib}\ n \\ \mathit{fib}\ n & \mathit{fib}\ (n+1) \end{pmatrix}$ for $n \geqslant 1$. Use the function *power* from the lecture notes to calculate F^n in no more than about $2 \log n$

matrix multiplications, and use this to give another more efficient definition of *fib*.

Roughly how big is the 1 000 000th Fibonacci number?

12.3 Recall that the Haskell function

```
error :: String -> a
```

never terminates successfully, but prints out a message including its argument. Using the definitions of *loop* and *loop′* from the lectures, and a function

```
> test f = f (const error) () ["strict","lazy"]
```

try to predict what happens when you evaluate each of *test loop* and *test loop′*.

Use GHCi to check your prediction, and explain the difference between the two.

What about *test foldl*?