Functional Programming
Practicals 3. Program Calculation

Shin-Cheng Mu

July 2018

1. **Longest positive segment**. The function *lpp* computes the length of the longest prefix that is all positive:

$$
\begin{array}{ll}
lpp & :: \mathsf{List\ Int} \rightarrow \mathsf{Nat} \\
lpp\ [\,] & = 0 \\
lpp\ (x : xs) & = \mathbf{if}\ x > 0\ \mathbf{then}\ \mathbf{1}_{+}\ (lpp\ xs)\ \mathbf{else}\ 0 \quad .
\end{array}
$$

The function *lps*, using lpp, computes the length of the longest positive segment:

$$
\begin{array}{ll}
lps & :: \mathsf{List\ Int} \rightarrow \mathsf{Nat} \\
lps\ [\,] & = 0 \\
lps\ (x : xs) & = lpp\ (x : xs) \uparrow lps\ xs \quad .
\end{array}
$$

(a) What are the time complexities of *lpp* and *lps*, with respect to the lengths of their inputs?

(b) Calculate a faster version of *lps*, by tupling *lps* and *lpp*.

Solution: The function *lps*, defined this way, is a $O(n^2)$ program.

To calculate an linear-time version, we define:

$$lpsp\ xs = (lps\ xs, lpp\ xs) \quad .$$

If we can construct a linear-time implementation of *lpsp*, we may define $lps = fst \cdot lpsp$. To calculate *lpsp*:

Case $xs := [\,]$. Apparantly $lpsp\ [\,] = (0,0)$.

Case $xs := x : xs$.

$$
\begin{array}{ll}
 & lpsp\ (x : xs) \\
= & (lps\ (x : xs), lpp\ (x : xs)) \\
= & \quad \{ \text{ definitions of } lps \text{ and } lpp \ \} \\
 & ((\mathbf{if}\ x > 0\ \mathbf{then}\ \mathbf{1}_{+}\ (lpp\ xs)\ \mathbf{else}\ 0) \uparrow lps\ xs,
\end{array}
$$

$$
\begin{array}{ll}
& \quad \textbf{if}\ x > 0\ \textbf{then}\ \mathbf{1}_+\ (lpp\ xs)\ \textbf{else}\ 0) \\
= & \quad \{\ \text{lifting common sub-expressions}\ \} \\
& \textbf{let}\ (m,n) = (lps\ xs, lpp\ xs) \\
& \qquad k \qquad = \textbf{if}\ x > 0\ \textbf{then}\ \mathbf{1}_+\ n\ \textbf{else}\ 0 \\
& \textbf{in}\ (k \uparrow m, k) \\
= & \quad \{\ \text{definition of}\ lpsp\ \} \\
& \textbf{let}\ (m,n) = lpsp\ xs \\
& \qquad k \qquad = \textbf{if}\ x > 0\ \textbf{then}\ \mathbf{1}_+\ n\ \textbf{else}\ 0 \\
& \textbf{in}\ (k \uparrow m, k) \quad .
\end{array}
$$

Thus we have derived:

$$
\begin{array}{ll}
lpsp\ [\,] & = (0,0) \\
lpsp\ (x:xs) & = \textbf{let}\ (m,n) = lpsp\ xs \\
& \qquad\ \ k \qquad = \textbf{if}\ x > 0\ \textbf{then}\ \mathbf{1}_+\ n\ \textbf{else}\ 0 \\
& \quad\ \textbf{in}\ (k \uparrow m, k) \quad .
\end{array}
$$

2. Let *descend* be defined by:

$$
\begin{array}{ll}
descend & :: \mathsf{Nat} \rightarrow \mathsf{List\ Nat} \\
descend\ 0 & = [\,] \\
descend\ (\mathbf{1}_+\ n) & = \mathbf{1}_+\ n : descend\ n \quad .
\end{array}
$$

   (a) Let $sumseries = sum \cdot descend$, synthesise an inductive definition of f.

Solution: It is immediate that $sum\ (descend\ 0) = 0$. For the inductive case we calculate:

$$
\begin{array}{ll}
& sum\ (descend\ (\mathbf{1}_+\ n)) \\
= & \{\ \text{definition of}\ descend\ \} \\
& sum\ ((\mathbf{1}_+\ n) : descend\ n) \\
= & \{\ \text{definition of}\ sum\ \} \\
& \mathbf{1}_+\ n + sum\ (descend\ n)) \\
= & \{\ \text{definition of}\ sum\ \} \\
& \mathbf{1}_+\ n\ + sumseries\ n \quad .
\end{array}
$$

Thus we have

$$
\begin{array}{lcl}
sumseries\ 0 & = & 0 \\
sumseries\ (\mathbf{1}_+\ n) & = & \mathbf{1}_+\ n\ + sumseries\ n \quad .
\end{array}
$$

(b) The function $repeatN :: (\mathsf{Nat}, a) \to \mathsf{List}\ a$ is defined by

$$repeatN\ (n,x) = map\ (const\ x)\ (descend\ n)\ \ .$$

Thus $repeatN\ (n,x)$ produces n copies of x in a list. E.g. $repeatN\ (3, \texttt{'a'}) = \texttt{"aaa"}$. Calculate an inductive definition of *repeatN*.

Solution: It is immediate that $repeatN\ (0,x) = [\,]$. For the inductive case we calculate

$$\begin{aligned}
& repeatN\ (\mathbf{1}_+\ n, x) \\
= & \quad \{ \text{ definition of } repeatN \ \} \\
& map\ (const\ x)\ (descend\ (\mathbf{1}_+\ n)) \\
= & \quad \{ \text{ definition of } descend \ \} \\
& map\ (const\ x)\ (\mathbf{1}_+\ n : descend\ n) \\
= & \quad \{ \text{ definition of } map \text{ and } const \ \} \\
& x : map\ (const\ x)\ (descend\ n) \\
= & \quad \{ \text{ definition of } repeatN \ \} \\
& x : repeatN\ (n,x)\ \ .
\end{aligned}$$

Thus we have

$$\begin{aligned}
repeatN\ (0, \quad\ x) &= [\,] \\
repeatN\ (\mathbf{1}_+\ n, x) &= x : repeatN\ (n,x)\ \ .
\end{aligned}$$

(c) The function $rld :: \mathsf{List}\ (\mathsf{Nat}, a) \to \mathsf{List}\ a$ performs run-length decoding:

$$rld = concat \cdot map\ repeatN\ \ .$$

For example, $rld\ [(2, \texttt{'a'}), (3, \texttt{'b'}), (1, \texttt{'c'})] = \texttt{"aabbbc"}$. Come up with an inductive defintion of *rld*.

Solution: For the base case:

$$\begin{aligned}
& rld\ [\,] \\
= & \quad \{ \text{ definition of } rld \ \} \\
& concat\ (map\ repeatN\ [\,]) \\
= & \quad \{ \text{ definitions of } map \text{ and } concat \ \} \\
& [\,]
\end{aligned}$$

For the inductive case:

$$\begin{aligned}
& rld\ ((n,x) : xs) \\
= & \quad \{ \text{ definition of } rld \ \} \\
& concat\ (map\ repeatN\ ((n,x) : xs))
\end{aligned}$$

$$
\begin{aligned}
&= \quad \{ \text{ definitions of } \mathit{map} \} \\
&\quad \mathit{concat}\ (\mathit{repeatN}\ (n,x) : \mathit{map}\ \mathit{repeatN}\ \mathit{xs}) \\
&= \quad \{ \text{ definitions of } \mathit{concat} \} \\
&\quad \mathit{repeatN}\ (n,x) +\!\!+ \mathit{concat}\ (\mathit{map}\ \mathit{repeatN}\ \mathit{xs}) \\
&= \quad \{ \text{ definition of } \mathit{rld} \} \\
&\quad \mathit{repeatN}\ (n,x) +\!\!+ \mathit{rld}\ \mathit{xs} \quad .
\end{aligned}
$$

We have thus derived:

$$
\begin{aligned}
&\mathit{rld}\ [\,] &&= [\,] \\
&\mathit{rld}\ ((n,x) : \mathit{xs}) &&= \mathit{repeatN}\ (n,x) +\!\!+ \mathit{rld}\ \mathit{xs} \quad .
\end{aligned}
$$

3. There is another way to define *pos* such that *pos x xs* yields the index of the first occurrence of x in *xs*:

$$
\begin{aligned}
&\mathit{pos} &&:: \mathit{Eq}\ a \Rightarrow a \rightarrow \mathit{List}\ a \rightarrow \mathit{Int} \\
&\mathit{pos}\ x &&= \mathit{length} \cdot \mathit{takeWhile}\ (x \neq)
\end{aligned}
$$

(This *pos* behaves differently from the one in the lecture when x does not occur in *xs*.) Construct an inductive definition of *pos*.

Solution: It is immediate that $\mathit{pos}\ x\ [\,] = 0$. For the inductive case we calculate:

$$
\begin{aligned}
&\quad \mathit{pos}\ x\ (y : \mathit{xs}) \\
&= \mathit{length}\ (\mathit{takeWhile}\ (x \neq)\ (y : \mathit{xs})) \\
&= \quad \{ \text{ definition of } \mathit{takeWhile} \ \} \\
&\quad \mathit{length}\ (\textbf{if}\ x \neq y\ \textbf{then}\ y : \mathit{takeWhile}\ (x \neq)\ \mathit{xs}\ \textbf{else}\ [\,]) \\
&= \quad \{ \text{ function application distributes into } \textbf{if} \text{ (for total functions) } \} \\
&\quad \textbf{if}\ x \neq y\ \textbf{then}\ \mathit{length}\ (y : \mathit{takeWhile}\ (x \neq)\ \mathit{xs})\ \textbf{else}\ \mathit{length}\ [\,] \\
&= \quad \{ \text{ definition of } \mathit{length} \ \} \\
&\quad \textbf{if}\ x \neq y\ \textbf{then}\ \mathbf{1}_{+}\ \mathit{length}\ (\mathit{takeWhile}\ (x \neq)\ \mathit{xs})\ \textbf{else}\ 0 \\
&= \quad \{ \text{ definition of } \mathit{pos} \ \} \\
&\quad \textbf{if}\ x \neq y\ \textbf{then}\ \mathbf{1}_{+}\ \mathit{pos}\ x\ \mathit{xs}\ \textbf{else}\ 0 \quad .
\end{aligned}
$$

Thus we have constructed:

$$
\begin{aligned}
&\mathit{pos}\ x\ [\,] &&= 0 \\
&\mathit{pos}\ x\ (y : \mathit{xs}) &&= \textbf{if}\ x \neq y\ \textbf{then}\ \mathbf{1}_{+}\ \mathit{pos}\ x\ \mathit{xs}\ \textbf{else}\ 0 \quad .
\end{aligned}
$$

4. Zipping and mapping.

(a) Let $\mathit{second}\ f\ (x,y) = (x, f\,y)$. Prove that $\mathit{zip}\ \mathit{xs}\ (\mathit{map}\ f\ \mathit{ys}) = \mathit{map}\ (\mathit{second}\ f)\ (\mathit{zip}\ \mathit{xs}\ \mathit{ys})$.

Solution: Recall one of the possible definitions of *zip*:

$$\begin{aligned}
&\mathit{zip}\ []\ \mathit{ys} &&= [] \\
&\mathit{zip}\ (x:\mathit{xs})\ [] &&= [] \\
&\mathit{zip}\ (x:\mathit{xs})\ (y:\mathit{ys}) &&= (x,y):\mathit{zip}\ \mathit{xs}\ \mathit{ys}.
\end{aligned}$$

Following the structure, we prove the proposition by induction on *xs* and *ys*. A tip for equational reasoning: it is usually easier to go from the more complex side to the simpler side, from the side with more structure to the side with less structure. Thus we start from the left-hand side.

Case $\mathit{xs} := []$.

$$\begin{aligned}
&\quad \mathit{map}\ (\mathit{second}\ f)\ (\mathit{zip}\ []\ \mathit{ys}) \\
&= \quad \{\ \text{definition of } \mathit{zip}\ \} \\
&\quad \mathit{map}\ (\mathit{second}\ f)\ [] \\
&= \quad \{\ \text{definition of } \mathit{map}\ \} \\
&\quad [] \\
&= \quad \{\ \text{definition of } \mathit{zip}\ \} \\
&\quad \mathit{zip}\ []\ (\mathit{map}\ f\ \mathit{ys}).
\end{aligned}$$

Case $\mathit{xs} := x:\mathit{xs}$, $\mathit{ys} := []$.

$$\begin{aligned}
&\quad \mathit{map}\ (\mathit{second}\ f)\ (\mathit{zip}\ (x:\mathit{xs})\ []) \\
&= \quad \{\ \text{definition of } \mathit{zip}\ \} \\
&\quad \mathit{map}\ (\mathit{second}\ f)\ [] \\
&= \quad \{\ \text{definition of } \mathit{map}\ \} \\
&\quad [] \\
&= \quad \{\ \text{definition of } \mathit{zip}\ \} \\
&\quad \mathit{zip}\ (x:\mathit{xs})\ [] \\
&= \quad \{\ \text{definition of } \mathit{map}\ \} \\
&\quad \mathit{zip}\ (x:\mathit{xs})\ (\mathit{map}\ f\ []).
\end{aligned}$$

Case $\mathit{xs} := x:\mathit{xs}$, $\mathit{ys} := y:\mathit{ys}$.

$$\begin{aligned}
&\quad \mathit{map}\ (\mathit{second}\ f)\ (\mathit{zip}\ (x:\mathit{xs})\ (y:\mathit{ys})) \\
&= \quad \{\ \text{definition of } \mathit{zip}\ \} \\
&\quad \mathit{map}\ (\mathit{second}\ f)\ ((x,y):\mathit{zip}\ \mathit{xs}\ \mathit{ys}) \\
&= \quad \{\ \text{definition of } \mathit{map}\ \} \\
&\quad \mathit{second}\ f\ (x,y):\mathit{map}\ (\mathit{second}\ f)\ (\mathit{zip}\ \mathit{xs}\ \mathit{ys})
\end{aligned}$$

$$
\begin{aligned}
&= \quad \{\ \text{definition of } \mathit{second}\ \} \\
&\quad (x, f\ y) : \mathit{map}\ (\mathit{second}\ f)\ (\mathit{zip}\ \mathit{xs}\ \mathit{ys}) \\
&= \quad \{\ \text{induction}\ \} \\
&\quad (x, f\ y) : \mathit{zip}\ \mathit{xs}\ (\mathit{map}\ f\ \mathit{ys}) \\
&= \quad \{\ \text{definition of } \mathit{zip}\ \} \\
&\quad \mathit{zip}\ (x : \mathit{xs})\ (f\ y : \mathit{map}\ f\ \mathit{ys}) \\
&= \quad \{\ \text{definition of } \mathit{map}\ \} \\
&\quad \mathit{zip}\ (x : \mathit{xs})\ (\mathit{map}\ f\ (y : \mathit{ys})).
\end{aligned}
$$

(b) Consider the following definition

$$
\begin{aligned}
&\mathit{delete} && :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\
&\mathit{delete}\ [\,] && = [\,] \\
&\mathit{delete}\ (x : \mathit{xs}) && = \mathit{xs} : \mathit{map}\ (x:)\ (\mathit{delete}\ \mathit{xs}) \quad ,
\end{aligned}
$$

such that

$$
\mathit{delete}\ [1,2,3,4] = [[2,3,4],[1,3,4],[1,2,4],[1,2,3]] \quad .
$$

That is, each element in the input list is deleted in turns. Let $\mathit{select} :: \mathsf{List}\ a \to \mathsf{List}\ (a, \mathsf{List}\ a)$ be defined by $\mathit{select}\ \mathit{xs} = \mathit{zip}\ \mathit{xs}\ (\mathit{delete}\ \mathit{xs})$. Come up with an inductive definition of *select*. **Hint**: you may find *second* useful.

Solution: The base case $[\,]$ is immediate. For the inductive case:

$$
\begin{aligned}
&\quad \mathit{select}\ (x : \mathit{xs}) \\
&= \quad \{\ \text{definition of } \mathit{select}\ \} \\
&\quad \mathit{zip}\ (x : \mathit{xs})\ (\mathit{delete}\ (x : \mathit{xs})) \\
&= \quad \{\ \text{definition of } \mathit{delete}\ \} \\
&\quad \mathit{zip}\ (x : \mathit{xs})\ (\mathit{xs} : \mathit{map}\ (x:)\ (\mathit{delete}\ \mathit{xs})) \\
&= \quad \{\ \text{definition of } \mathit{zip}\ \} \\
&\quad (x, \mathit{xs}) : \mathit{zip}\ \mathit{xs}\ (\mathit{map}\ (x:)\ (\mathit{delete}\ \mathit{xs})) \\
&= \quad \{\ \text{property proved above}\ \} \\
&\quad (x, \mathit{xs}) : \mathit{map}\ (\mathit{second}\ (x:))\ (\mathit{zip}\ \mathit{xs}\ (\mathit{delete}\ \mathit{xs})) \\
&= \quad \{\ \text{definition of } \mathit{select}\ \} \\
&\quad (x, \mathit{xs}) : \mathit{map}\ (\mathit{second}\ (x:))\ (\mathit{select}\ \mathit{xs}) \quad .
\end{aligned}
$$

We thus have

$$
\begin{aligned}
&\mathit{select}\ [\,] && = [\,] \\
&\mathit{select}\ (x : \mathit{xs}) && = (x, \mathit{xs}) : \mathit{map}\ (\mathit{second}\ (x:))\ (\mathit{select}\ \mathit{xs}) \quad .
\end{aligned}
$$

(c) An alternative specification of *delete* is

$$\begin{array}{l} \mathit{delete}\ \mathit{xs} = \mathit{map}\ (\mathit{del}\ \mathit{xs})\ [0\,..\,\mathit{length}\ \mathit{xs} - 1] \\ \quad \textbf{where}\ \mathit{del}\ \mathit{xs}\ i = \mathit{take}\ i\ \mathit{xs} \mathbin{+\!\!+} \mathit{drop}\ (1+i)\ \mathit{xs}\ \ , \end{array}$$

(here we take advantage of the fact that $[0\,..\,n]$ returns $[\,]$ when n is negative). From this specification, derive the inductive definition of *delete* given above. **Hint**: you may need the following property:

$$[0\,..\,n] = 0 : \mathit{map}\ (\mathbf{1}_+)\ [0\,..\,n-1], \quad \text{if } n \geqslant 0, \tag{1}$$

and the *map-fusion* law.

Solution:

$$\begin{array}{cl} & \mathit{delete}\ (x:\mathit{xs}) \\ = & \quad \{\ \text{definition of } \mathit{delete}\ \} \\ & \mathit{map}\ (\mathit{del}\ (x:\mathit{xs}))\ [0\,..\,\mathit{length}\ (x:\mathit{xs}) - 1] \\ = & \quad \{\ \text{defintion of } \mathit{length}\text{, arithmetics}\ \} \\ & \mathit{map}\ (\mathit{del}\ (x:\mathit{xs}))\ [0\,..\,\mathit{length}\ \mathit{xs}] \\ = & \quad \{\ \mathit{length}\ \mathit{xs} \geqslant 0\text{, by (1)}\ \} \\ & \mathit{map}\ (\mathit{del}\ (x:\mathit{xs}))\ (0 : \mathit{map}\ (\mathbf{1}_+)\ [0\,..\,\mathit{length}\ \mathit{xs} - 1]) \\ = & \quad \{\ \text{definition of } \mathit{map}\ \} \\ & \mathit{del}\ (x:\mathit{xs})\ 0 : \mathit{map}\ (\mathit{del}\ (x:\mathit{xs}))\ (\mathit{map}\ (\mathbf{1}_+)\ [0\,..\,\mathit{length}\ \mathit{xs} - 1]) \\ = & \quad \{\ \text{map fusion } (\mathbf{??})\ \} \\ & \mathit{del}\ (x:\mathit{xs})\ 0 : \mathit{map}\ (\mathit{del}\ (x:\mathit{xs}) \cdot (\mathbf{1}_+))\ [0\,..\,\mathit{length}\ \mathit{xs} - 1] \end{array}$$

Now we pause for a while to inspect $\mathit{del}\ (x:\mathit{xs})$. Apparently, $\mathit{del}\ (x:\mathit{xs})\ 0 = \mathit{xs}$. For $\mathit{del}\ (x:\mathit{xs}) \cdot (\mathbf{1}_+)$ we calculate:

$$\begin{array}{cl} & (\mathit{del}\ (x:\mathit{xs}) \cdot (\mathbf{1}_+))\ i \\ = & \quad \{\ \text{definition of } (\cdot)\ \} \\ & \mathit{del}\ (x:\mathit{xs})\ (\mathbf{1}_+\ i) \\ = & \quad \{\ \text{definition of } \mathit{del}\ \} \\ & \mathit{take}\ (\mathbf{1}_+\ i)\ (x:\mathit{xs}) \mathbin{+\!\!+} \mathit{drop}\ (\mathbf{1}_+\ (\mathbf{1}_+\ i))\ (x:\mathit{xs}) \\ = & \quad \{\ \text{definitions of } \mathit{take} \text{ and } \mathit{drop}\ \} \\ & x : \mathit{take}\ i\ \mathit{xs} \mathbin{+\!\!+} \mathit{drop}\ (\mathbf{1}_+\ i)\ \mathit{xs} \\ = & \quad \{\ \text{definition of } \mathit{del}\ \} \\ & x : \mathit{del}\ \mathit{xs}\ i \\ = & \quad \{\ \text{definition of } (\cdot)\ \} \\ & ((x:) \cdot \mathit{del}\ \mathit{xs})\ i\ \ . \end{array}$$

We resume the calculation:

$$\begin{array}{cl} & \mathit{del}\ (x:\mathit{xs})\ 0 : \mathit{map}\ (\mathit{del}\ (x:\mathit{xs}) \cdot (\mathbf{1}_+))\ [0\,..\,\mathit{length}\ \mathit{xs} - 1] \\ = & \quad \{\ \text{calculation above}\ \} \end{array}$$

$$
\begin{aligned}
& xs : map\ ((x{:}) \cdot del\ xs)\ [0\,..\,length\ xs - 1] \\
= \quad & \{\ \text{map fusion } (\textbf{??})\ \} \\
& xs : map\ (x{:})\ (map\ (del\ xs)\ [0\,..\,length\ xs - 1]) \\
= \quad & \{\ \text{definition of } delete\ \} \\
& xs : map\ (x{:})\ (delete\ xs)\ \ .
\end{aligned}
$$

We have thus derived the first, inductive definition of *delete*.

5. Assume that multiplication $(\times)$ is a constant-time operation. One possible definition for $exp\ m\ n = m^n$ could be:

$$
\begin{aligned}
& exp && :: Nat \to Nat \to Nat \\
& exp\ m\ 0 && = 1 \\
& exp\ m\ (1+n) && = m \times exp\ m\ n
\end{aligned}
$$

Therefore, to compute $exp\ m\ n$, multiplication is called n times: $m \times m \times \ldots \times m \times 1$. Can we do better?

Yet another way to represent a natural number is to use the binary representation.

(a) The function $binary :: Nat \to [Bool]$ returns the *reversed* binary representation of a natural number. For example:

$$
\begin{aligned}
& binary\ 0 = [\,], \\
& binary\ 1 = [T], \\
& binary\ 2 = [F,T], \\
& binary\ 3 = [T,T], \\
& binary\ 4 = [F,F,T].
\end{aligned}
$$

Given the following functions:

$even :: Nat \to Bool$, returning true iff the input is even,

$odd :: Nat \to Bool$, returning true iff the input is odd, and

$div :: Nat \to Nat \to Nat$, for integral division,

define *binary*. You may just present the code.

Hint One possible implementation discriminates between 3 cases – the input is 0, the input is odd, and the input is even.

Solution:

$$
\begin{aligned}
& binary && :: Nat \to List\ Bool \\
& binary\ 0 && = \ [\,] \\
& binary\ n && \mid even\ n = False : binary\ (n\ \text{`}div\text{`}\ 2) \\
& && \mid odd\ n = True : binary\ ((n-1)\ \text{`}div\text{`}\ 2)
\end{aligned}
$$

(b) Briefly explain in words whether your implementation of *binary* terminates for all input in *Nat*, and why.

> **Solution:** All non-zero natural numbers strictly decreases when being divided by 2, and thus we eventually reaches the base case for 0.

(c) Define a function $decimal :: List\ Bool \to Nat$ that takes the reversed binary representation and returns the corresponding natural number. E.g. $decimal\ [T,T,F,T] = 11$. You may just present the code.

> **Solution:**
>
> $$\begin{array}{ll} decimal & :: List\ Bool \to Nat \\ decimal\ [\,] & = 0 \\ decimal\ (False : xs) & = 2 \times decimal\ xs \\ decimal\ (True : xs) & = 1 + (2 \times decimal\ xs) \end{array}$$

(d) Let $roll\ m = exp\ m \cdot decimal$. Assuming we have proved that $exp\ m\ n$ satisfies all arithmetic laws for m^n. Construct (with algebraic calculation) a definition of *roll* that does not make calls to *exp* or *decimal*.

> **Solution:** Let's calculate $roll\ m\ xs = exp\ m\ (decimal\ xs)$ by distinguishing between the three cases of n: **Case** $xs := [\,]$:
>
> $$\begin{array}{ll} & roll\ m\ [\,] \\ = & exp\ m\ (decimal\ [\,]) \\ = & \{\ \text{definition of } decimal\ \} \\ & exp\ m\ 0 \\ = & \{\ \text{definition of } exp\ \} \\ & 1 \end{array}$$
>
> **Case** $xs = False : xs$:
>
> $$\begin{array}{ll} & roll\ m\ (False : xs) \\ = & \{\ \text{definition of } roll\ \} \\ & exp\ m\ (decimal\ (False : xs)) \\ = & \{\ \text{definition of } decimal\ \} \\ & exp\ m\ (2 \times decimal\ xs) \\ = & \{\ \text{arithmetic: } m^{2n} = (m^2)^n\ \} \\ & exp\ (m \times m)\ (decimal\ xs) \\ = & \{\ \text{definition of } roll\ \} \end{array}$$

$$roll\ (m \times m)\ xs$$

Case $xs = True : xs$:

$$
\begin{aligned}
& roll\ m\ (True : xs) \\
= & \quad \{ \text{ definition of } roll \ \} \\
& exp\ m\ (decimal\ (True : xs)) \\
= & \quad \{ \text{ definition of } decimal \ \} \\
& exp\ m\ (1 + 2 \times decimal\ xs) \\
= & \quad \{ \text{ definition of } exp \ \} \\
& m \times exp\ m\ (2 \times decimal\ xs) \\
= & \quad \{ \text{ arithmetic: } m^{2n} = (m^2)^n \ \} \\
& m \times exp\ (m \times m)\ (decimal\ xs) \\
= & \quad \{ \text{ definition of } roll \ \} \\
& m \times roll\ (m \times m)\ xs
\end{aligned}
$$

We have thus constructed:

$$
\begin{aligned}
& roll\ m\ [\,] && = 1 \\
& roll\ m\ (False : xs) && = roll\ (m \times m)\ xs \\
& roll\ m\ (True : xs) && = m \times roll\ (m \times m)\ xs
\end{aligned}
$$

Remark If the fusion succeeds, we have derived a program computing m^n:

$$fastexp\ m = roll\ m \cdot binary.$$

The algorithm runs in time proportional to the length of the list generated by *binary*, which is $O(\log_2 n)$.

6. Alternatively, define *repeatN* by:

$$repeatN\ (n, x) = map\ (const\ x)\ [0 \mathinner{..} n - 1] \ .$$

   (a) Try to construct an inductive definition of *repeatN* by induction on n, and see how this might not work.

   (b) Define $repeatFrom\ i\ (n, x) = map\ (const\ x)\ [i \mathinner{..} n - 1]$.

7. The function *from* generates an infinite list of numbers:

$$
\begin{aligned}
& from && :: Int \to List\ Int \\
& from\ n && = n : from\ (1 + n)
\end{aligned}
$$

In fact, $\mathit{from}\ n = [n..]$. Consider the following definition:

$$\begin{aligned}
&\mathit{positions} && :: (a \to \mathit{Bool}) \to \mathit{List}\ a \to \mathit{List}\ \mathit{Int} \\
&\mathit{positions}\ p && = \mathit{map}\ \mathit{fst} \cdot \mathit{filter}\ (p \cdot \mathit{snd}) \cdot \mathit{zip}\ (\mathit{from}\ 0)
\end{aligned}$$

One problem with the definition is that it builds many intermediate lists in the middle. Try deriving, with algebraic calculation, a alternative definition of *positions* that do not build those intermediate lists.

Hint: Start with trying to construct a definition of *positions p xs* that is inductively defined on *xs*. You might then find out that this does not work, and you need to define a generalised function, for which *positions p xs* is a special case.

Solution: One may start with trying to inductively define *positions p* on the input list. We omit the base case and look at the inductive case:

$$\begin{aligned}
& \mathit{positions}\ p\ (x : xs) \\
= {} & \mathit{map}\ \mathit{fst}\ (\mathit{filter}\ (p \cdot \mathit{snd})\ (\mathit{zip}\ (\mathit{from}\ 0)\ (x : xs)) \\
= {} & \quad \{\ \text{definition of } \mathit{zip}\ \} \\
& \mathit{map}\ \mathit{fst}\ (\mathit{filter}\ (p \cdot \mathit{snd})\ ((0, x) : \mathit{zip}\ (\mathit{from}\ 1)\ xs)
\end{aligned}$$

We may proceed with it but soon we will encounter difficulty not being able to fold back $\mathit{map}\ \mathit{fst}\ (\mathit{filter}\ (p \cdot \mathit{snd})\ (\mathit{zip}\ (\mathit{from}\ 1)\ xs)$.

Instead, we define

$$\begin{aligned}
&\mathit{posFrom} && :: (a \to \mathit{Bool}) \to \mathit{Int} \to \mathit{List}\ a \to \mathit{List}\ \mathit{Int} \\
&\mathit{posFrom}\ p\ n\ xs && = \mathit{map}\ \mathit{fst}\ (\mathit{filter}\ (p \cdot \mathit{snd})\ (\mathit{zip}\ (\mathit{from}\ n)\ xs))
\end{aligned}$$

If we can construct a quick definition of *posFrom*, we may simply let

$$\mathit{positions}\ p\ xs = \mathit{posFrom}\ p\ 0\ xs$$

Now we try to construct *posFrom*. The base case *posFrom p n xs* is easy. We look at the inductive case with input $x : xs$:

$$\begin{aligned}
& \mathit{posFrom}\ p\ n\ (x : xs) \\
= {} & \mathit{map}\ \mathit{fst}\ (\mathit{filter}\ (p \cdot \mathit{snd})\ (\mathit{zip}\ (\mathit{from}\ n)\ (x : xs))) \\
= {} & \quad \{\ \text{definition of } \mathit{zip}\ \} \\
& \mathit{map}\ \mathit{fst}\ (\mathit{filter}\ (p \cdot \mathit{snd})\ ((n, x) : \mathit{zip}\ (\mathit{from}\ (1 + n))\ xs)) \\
= {} & \quad \{\ \text{definition of } \mathit{filter}\ \} \\
& \mathit{map}\ \mathit{fst}\ (\mathbf{if}\ (p\ (\mathit{snd}\ (n, x)))\ \mathbf{then}\ (n, x) : \mathit{filter}\ (p \cdot \mathit{snd})\ (\mathit{zip}\ (\mathit{from}\ (1 + n))\ xs) \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{filter}\ (p \cdot \mathit{snd})\ (\mathit{zip}\ (\mathit{from}\ (1 + n))\ xs) \\
= {} & \quad \{\ \text{function composition, } \mathit{snd}\ \}
\end{aligned}$$

$$
\begin{array}{ll}
& \mathit{map}\ \mathit{fst}\ (\textbf{if}\ p\ x\ \textbf{then}\ (n,x) : \mathit{filter}\ (p \cdot \mathit{snd})\ (\mathit{zip}\ (\mathit{from}\ (1+n))\ \mathit{xs}) \\
& \qquad\qquad\qquad \textbf{else}\ \mathit{filter}\ (p \cdot \mathit{snd})\ (\mathit{zip}\ (\mathit{from}\ (1+n))\ \mathit{xs}) \\
= & \{\ f\ (\textbf{if}\ q\ \textbf{then}\ e_1\ \textbf{else}\ e_2) = \textbf{if}\ q\ \textbf{then}\ f\ e_1\ \textbf{else}\ f\ e_2\ \} \\
& \textbf{if}\ p\ x\ \textbf{then}\ \mathit{map}\ \mathit{fst}\ ((n,x) : \mathit{filter}\ (p \cdot \mathit{snd})\ (\mathit{zip}\ (\mathit{from}\ (1+n))\ \mathit{xs})) \\
& \qquad \textbf{else}\ \mathit{map}\ \mathit{fst}\ (\mathit{filter}\ (p \cdot \mathit{snd})\ (\mathit{zip}\ (\mathit{from}\ (1+n))\ \mathit{xs})) \\
= & \{\ \text{definition of } \mathit{map}\ \} \\
& \textbf{if}\ p\ x\ \textbf{then}\ n : \mathit{map}\ \mathit{fst}\ \mathit{filter}\ (p \cdot \mathit{snd})\ (\mathit{zip}\ (\mathit{from}\ (1+n))\ \mathit{xs})) \\
& \qquad \textbf{else}\ \mathit{map}\ \mathit{fst}\ (\mathit{filter}\ (p \cdot \mathit{snd})\ (\mathit{zip}\ (\mathit{from}\ (1+n))\ \mathit{xs})) \\
= & \{\ \text{definition of } \mathit{posFrom}\ \} \\
& \textbf{if}\ p\ x\ \textbf{then}\ n : \mathit{posFrom}\ p\ (1+n)\ \mathit{xs} \\
& \qquad \textbf{else}\ \mathit{posFrom}\ p\ (1+n)\ \mathit{xs}
\end{array}
$$

Thus we have

$$
\begin{array}{ll}
\mathit{posFrom}\ p\ n\ [\,] & = [\,] \\
\mathit{posFrom}\ p\ n\ (x : \mathit{xs}) & = \textbf{if}\ p\ x\ \textbf{then}\ n : \mathit{posFrom}\ p\ (1+n)\ \mathit{xs} \\
& \qquad\quad \textbf{else}\ \mathit{posFrom}\ p\ (1+n)\ \mathit{xs}
\end{array}
$$

8. Prove that $\mathit{reverse} \cdot \mathit{reverse} = \mathit{id}$ (for finite lists). It will turn out that you need to prove a stronger lemma, which may need the alternative definition of *reverse* in terms of *revcat*.

Solution:

The goal is to prove that

$$\mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) = \mathit{xs} \tag{2}$$

which, if we take $\mathit{reverse}\ \mathit{xs} = \mathit{revcat}\ \mathit{xs}\ [\,]$ as known, is equivalent to

$$\mathit{reverse}\ (\mathit{revcat}\ \mathit{xs}\ [\,]) = \mathit{xs} \tag{3}$$

The base case for $[\,]$ is trivial, for the inductive case $(x : \mathit{xs})$, our first attempt could be

$$
\begin{array}{ll}
& \mathit{reverse}\ (\mathit{reverse}\ (x : \mathit{xs})) \\
= & \{\ \mathit{reverse}\ \mathit{xs} = \mathit{revcat}\ \mathit{xs}\ [\,]\ \} \\
& \mathit{reverse}\ (\mathit{revcat}\ (x : \mathit{xs})\ [\,]) \\
= & \{\ \text{definition of } \mathit{revcat}\ \} \\
& \mathit{reverse}\ (\mathit{revcat}\ \mathit{xs}\ [x])
\end{array}
$$

Then we are stuck — we cannot use (**??**) as the inductive hypothesis, since we have $[x]$, not $[\,]$, as the argument of *revcat*.

Thus we generalise (**??**) to

$$\mathit{reverse}\ (\mathit{revcat}\ \mathit{xs}\ \mathit{ys}) = ?$$

what should the right-hand side be? A moment's thought leads to

$$\mathit{reverse}\ (\mathit{revcat}\ \mathit{xs}\ \mathit{ys}) = \mathit{revcat}\ \mathit{ys}\ \mathit{xs} \qquad (4)$$

Or something equivalent (e.g. $\mathit{reverse}\ (\mathit{revcat}\ \mathit{xs}\ \mathit{ys}) = \mathit{reverse}\ \mathit{ys} +\!\!+ \mathit{xs}$. If you use this one you may need some more additional steps in the proof later, but it still works anyway).

Note that once we prove (**??**), (**??**) follows as a corollary by letting $\mathit{ys} = [\,]$. Thus we do not need another inductive proof for (**??**).

We prove (**??**) by induction on *xs*. The base case [] is omitted. For the inductive case:

$$\begin{aligned}
& \mathit{reverse}\ (\mathit{revcat}\ (x : \mathit{xs})\ \mathit{ys}) \\
= \ & \{\ \text{definition of } \mathit{revcat}\ \} \\
& \mathit{reverse}\ (\mathit{revcat}\ \mathit{xs}\ (x : \mathit{ys})) \\
= \ & \{\ \text{induction hypothesis}\ \} \\
& \mathit{revcat}\ (x : \mathit{ys})\ \mathit{xs} \\
= \ & \{\ \text{definition of } \mathit{revcat}\ \} \\
& \mathit{revcat}\ \mathit{ys}\ (x : \mathit{xs})
\end{aligned}$$

In fact, you could rephrase (**??**) as

$$\mathit{reverse}\ (\mathit{reverse}\ \mathit{xs} +\!\!+ \mathit{ys}) = \mathit{reverse}\ \mathit{ys} +\!\!+ \mathit{xs}$$

and use only the original definition of *reverse* (that is, $\mathit{reverse}\ (x : \mathit{xs}) = \mathit{reverse}\ \mathit{xs} +\!\!+ [x]$), and the fact that $(+\!\!+)$ is associative. Thinking in terms of *revcat* was how I discovered (**??**), though.

9. Recall the standard definition of factorial:

$$\begin{aligned}
& \mathit{fact} && :: \mathit{Int} \to \mathit{Int} \\
& \mathit{fact}\ 0 && = 1, \\
& \mathit{fact}\ (\mathbf{1}_+\ n) && = (\mathbf{1}_+\ n) \times \mathit{fact}\ n.
\end{aligned}$$

This program implicitly uses space linear to n in the call stack.

1. Introduce $\mathit{factit}\ n\ m = \ldots$ where m is an accumulating parameter.
2. Express *fact* in terms of *factit*.

3. Construct a space efficient implementation of *factit*.

Solution: To exploit associativity of $(\times)$, we define:

$$\mathit{factit}\ n\ m = m \times \mathit{fact}\ n.$$

We recover *fact* by letting

$$\mathit{fact}\ n = \mathit{factit}\ n\ 1.$$

To construct *factit* we derive:
Case $n := 0$:

$$\begin{aligned} & \mathit{factit}\ 0\ m \\ = & \quad \{ \text{ definition of } \mathit{factit} \ \} \\ & m \times \mathit{fact}\ 0 \\ = & \quad \{ \text{ definition of } \mathit{fact} \ \} \\ & m. \end{aligned}$$

Case $n := \mathbf{1}_+\ n$:

$$\begin{aligned} & \mathit{factit}\ (\mathbf{1}_+\ n)\ m \\ = & \quad \{ \text{ definition of } \mathit{factit} \ \} \\ & m \times \mathit{fact}\ (\mathbf{1}_+\ n) \\ = & \quad \{ \text{ definition of } \mathit{fact} \ \} \\ & m \times ((\mathbf{1}_+\ n) \times \mathit{fact}\ n) \\ = & \quad \{ \ (\times) \text{ associative } \ \} \\ & (m \times (\mathbf{1}_+\ n)) \times \mathit{fact}\ n \\ = & \quad \{ \text{ definition of } \mathit{factit} \ \} \\ & \mathit{factit}\ n\ (m \times (\mathbf{1}_+\ n)). \end{aligned}$$

Thus,

$$\begin{aligned} \mathit{factit}\ 0\ m &= m \\ \mathit{factit}\ (\mathbf{1}_+\ n)\ m &= \mathit{factit}\ n\ (m \times (\mathbf{1}_+\ n)). \end{aligned}$$

10. Recall the standard definition of Fibonacci:

$$\begin{aligned} \mathit{fib}\ 0 &= 0 \\ \mathit{fib}\ 1 &= 1 \\ \mathit{fib}\ (\mathbf{1}_+\ (\mathbf{1}_+ n)) &= \mathit{fib}\ (\mathbf{1}_+\ n) + \mathit{fib}\ n. \end{aligned}$$

Let us try to derive a linear-time, tail-recursive algorithm computing *fib*.

1. Given the definition $\mathit{ffib}\ n\ x\ y = \mathit{fib}\ n \times x + \mathit{fib}\ (\mathbf{1}_+\ n)\ \times y$. Express *fib* using *ffib*.
2. Derive a linear-time version of *ffib*.

Solution: $\mathit{fib}\ n = \mathit{ffib}\ n\ 1\ 0$.

To construct *ffib*, we calculate:
Case $n := 0$:

$$
\begin{aligned}
& \mathit{ffib}\ 0\ x\ y \\
= & \quad \{ \text{ definition of } \mathit{ffib} \ \} \\
& \mathit{fib}\ 0 \times x + \mathit{fib}\ 1 \times y \\
= & \quad \{ \text{ definition of } \mathit{fib} \ \} \\
& 0 \times x + 1 \times y \\
= & \quad \{ \text{ arithmetics } \} \\
& y
\end{aligned}
$$

Case $n := \mathbf{1}_+\ n$:

$$
\begin{aligned}
& \mathit{ffib}\ (\mathbf{1}_+\ n)\ x\ y \\
= & \quad \{ \text{ definition of } \mathit{ffib} \ \} \\
& \mathit{fib}\ (\mathbf{1}_+\ n) \times x + \mathit{fib}\ (\mathbf{1}_+(\mathbf{1}_+ n)) \times y \\
= & \quad \{ \text{ definition of } \mathit{fib} \ \} \\
& \mathit{fib}\ (\mathbf{1}_+\ n) \times x + (\mathit{fib}\ (\mathbf{1}_+\ n) + \mathit{fib}\ n) \times y \\
= & \quad \{ \text{ arithmetics } \} \\
& \mathit{fib}\ (\mathbf{1}_+\ n) \times (x + y) + \mathit{fib}\ n \times y \\
= & \quad \{ \text{ definition of } \mathit{ffib} \ \} \\
& \mathit{ffib}\ n\ y\ (x+y)
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
\mathit{ffib}\ 0\ x\ y & = y \\
\mathit{ffib}\ (\mathbf{1}_+\ n)\ x\ y & = \mathit{ffib}\ n\ y\ (x+y)
\end{aligned}
$$

11. The following problem concerns calculating the sum $\sum_{i=0}^{n}(x_i \times y^i)$. Let *geo* be defined by:

$$
\begin{aligned}
\mathit{geo}\ y & = 1 : \mathit{map}\ (y\times)\ (\mathit{geo}\ y), \\
\mathit{horner}\ y\ \mathit{xs} & = \mathit{sum}\ (\mathit{map}\ \mathit{mul}\ (\mathit{zip}\ \mathit{xs}\ (\mathit{geo}\ y))),
\end{aligned}
$$

where $\mathit{mul}\ (a,b) = a \times b$. Let $\mathit{xs} = [x_0, x_1, \ldots, x_n]$, $\mathit{horner}\ y\ \mathit{xs}$ computes the sum $x_0 + x_1 \times y + x_2 \times y^2 + \cdots + x_n \times y^n$.

(a) Show that $mul \cdot second\ (y\times) = (y\times) \cdot mul$.
(**Remark**: for those who familiar with currying, $mul = uncurry\ (\times)$.)

Solution:

$$
\begin{aligned}
& mul\ (second\ (y\times)\ (x,z)) \\
= & \quad \{ \text{ definition of } second \ \} \\
& mul\ (x, y \times z) \\
= & \quad \{ \text{ definition of } mul \ \} \\
& x \times (y \times z) \\
= & \quad \{ \text{ arithmetics } \} \\
& y \times (x \times z) \\
= & \quad \{ \text{ definition of } mul \ \} \\
& y \times mul\ (x,z).
\end{aligned}
$$

(b) Let $n = length\ xs$. Asymptotically (that is, in terms of the big-O notation), how many multiplications $(\times)$ one must perform to compute $horner\ y\ xs$?

(c) Construct an inductive definition of $horner$ that uses only $O(n)$ multiplications to compute $horner\ y\ xs$. **Hint**: you will need properties proved in the previous problems in this exercise, and a property in the midterm exam concerning sum and $map\ (y\times)$, and perhaps some more properties. Unlike in the previous problem, however, you do not need to generalise $horner$.

Solution: We construct an inductive definition of $horner$ by case analysis.

Case $xs := []$. It is immediate that $horner\ y\ [] =\ 0$. Details omitted.

Case $xs := x : xs$.

$$
\begin{aligned}
& horner\ y\ (x : xs) \\
= & \quad \{ \text{ definition of } horner \ \} \\
& sum\ (map\ mul\ (zip\ (x : xs)\ (geo\ y))) \\
= & \quad \{ \text{ definition of } geo \ \} \\
& sum\ (map\ mul\ (zip\ (x : xs)\ (1 : map\ (y\times)\ (geo\ y)))) \\
= & \quad \{ \text{ definition of } zip \ \} \\
& sum\ (map\ mul\ ((x,1) : zip\ xs\ (map\ (y\times)\ (geo\ y)))) \\
= & \quad \{ \text{ definition of } map \text{ and } mul \ \} \\
& sum\ (x : map\ mul\ (zip\ xs\ (map\ (y\times)\ (geo\ y)))) \\
= & \quad \{ \text{ definition of } sum \ \} \\
& x + sum\ (map\ mul\ (zip\ xs\ (map\ (y\times)\ (geo\ y))))
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{ \text{ since } \mathit{zip}\ \mathit{xs}\ (\mathit{map}\ f\ \mathit{ys}) = \mathit{map}\ (\mathit{second}\ f)\ (\mathit{zip}\ \mathit{xs}\ \mathit{ys}) \ \} \\
&\quad x + \mathit{sum}\ (\mathit{map}\ \mathit{mul}\ (\mathit{map}\ (\mathit{second}\ (y\times))\ (\mathit{zip}\ \mathit{xs}\ (\mathit{geo}\ y)))) \\
&= \quad \{ \text{ since } \mathit{map}\ f \cdot \mathit{map}\ g = \mathit{map}\ (f \cdot g) \ \} \\
&\quad x + \mathit{sum}\ (\mathit{map}\ (\mathit{mul} \cdot \mathit{second}\ (y\times))\ (\mathit{zip}\ \mathit{xs}\ (\mathit{geo}\ y))) \\
&= \quad \{ \text{ since } \mathit{mul} \cdot \mathit{second}\ (y\times) = (y\times) \cdot \mathit{mul} \ \} \\
&\quad x + \mathit{sum}\ (\mathit{map}\ ((y\times) \cdot \mathit{mul})\ (\mathit{zip}\ \mathit{xs}\ (\mathit{geo}\ y))) \\
&= \quad \{ \text{ since } \mathit{map}\ f \cdot \mathit{map}\ g = \mathit{map}\ (f \cdot g) \ \} \\
&\quad x + \mathit{sum}\ (\mathit{map}\ (y\times)\ (\mathit{map}\ \mathit{mul}\ (\mathit{zip}\ \mathit{xs}\ (\mathit{geo}\ y)))) \\
&= \quad \{ \text{ since } \mathit{sum} \cdot \mathit{map}\ (y\times) = (y\times) \cdot \mathit{sum} \ \} \\
&\quad x + y \times \mathit{sum}\ (\mathit{map}\ \mathit{mul}\ (\mathit{zip}\ \mathit{xs}\ (\mathit{geo}\ y))) \\
&= \quad \{ \text{ definition of } \mathit{horner} \ \} \\
&\quad x + y \times \mathit{horner}\ y\ \mathit{xs}.
\end{aligned}
$$

Thus we conclude that

$$
\begin{aligned}
\mathit{horner}\ y\ [\,] &= 0 \\
\mathit{horner}\ y\ (x : \mathit{xs}) &= x + y \times \mathit{horner}\ y\ \mathit{xs}.
\end{aligned}
$$