Functional Programming
Practicals 0

Shin-Cheng Mu

FLOLAC 2024

Reviews...

1. A practice on curried functions.
   (a) Define a function *poly* such that $poly\ a\ b\ c\ x = a \times x^2 + b \times x + c$. All the inputs and the result are of type *Float*.
   (b) Reuse *poly* to define a function *poly1* such that $poly1\ x = x^2 + 2 \times x + 1$.
   (c) Reuse *poly* to define a function *poly2* such that $poly2\ a\ b\ c = a \times 2^2 + b \times 2 + c$.
2. Type in the definition of *square* in your working file.
   (a) Define a function $quad :: \mathrm{Int} \rightarrow \mathrm{Int}$ such that $quad\ x$ computes x^4.
   (b) Type in this definition into your working file. Describe, in words, what this function does.

$$\begin{array}{ll} twice & :: (a \rightarrow a) \rightarrow (a \rightarrow a) \\ twice\ f\ \ x & = f\ (f\ x)\quad . \end{array}$$

   (c) Define *quad* using *twice*.
3. Replace the previous *twice* with this definition:

$$\begin{array}{ll} twice & :: (a \rightarrow a) \rightarrow (a \rightarrow a) \\ twice\ f & = f \cdot f\quad . \end{array}$$

   (a) Does *quad* still behave the same?
   (b) Explain in words what this operator $(\cdot)$ does.
4. Functions as arguments, and a quick practice on sectioning.

(a) Type in the following definition to your working file, without giving the type.

$$\mathit{forktimes}\ f\ g\ x = f\ x \times g\ x\ \ .$$

Use $:t$ in GHCi to find out the type of *forktimes*. You will end up getting a complex type which, for now, can be seen as equivalent to

$$(t \to \mathit{Int}) \to (t \to \mathit{Int}) \to t \to \mathit{Int}\ \ .$$

Can you explain this type?

(b) Define a function that, given input x, use *forktimes* to compute $x^2 + 3 \times x + 2$. **Hint**: $x^2 + 3 \times x + 2 = (x + 1) \times (x + 2)$.

(c) Type in the following definition into your working file: $\mathit{lift2}\ h\ f\ g\ x = h\ (f\ x)\ (g\ x)$. Find out the type of *lift2*. Can you explain its type?

(d) Use *lift2* to compute $x^2 + 3 \times x + 2$.

Definitions and Proofs by Induction

1. Prove that *length* distributes into $(+\!\!+)$:

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

2. Prove: $\mathit{sum} \cdot \mathit{concat} = \mathit{sum} \cdot \mathit{map}\ \mathit{sum}$.

3. Prove: $\mathit{filter}\ p \cdot \mathit{map}\ f = \mathit{map}\ f \cdot \mathit{filter}\ (p \cdot f)$.
Hint: for calculation, it might be easier to use this definition of *filter*:

$$\begin{aligned}
&\mathit{filter}\ p\ [\,] &&= [\,] \\
&\mathit{filter}\ p\ (x : \mathit{xs}) &&= \mathbf{if}\ p\ x\ \mathbf{then}\ x : \mathit{filter}\ p\ \mathit{xs} \\
& &&\quad\ \mathbf{else}\ \mathit{filter}\ p\ \mathit{xs}
\end{aligned}$$

and use the law that in the world of total functions we have:

$$f\ (\mathbf{if}\ q\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) = \mathbf{if}\ q\ \mathbf{then}\ f\ e_1\ \mathbf{else}\ f\ e_2$$

You may also carry out the proof using the definition of *filter* using guards:

$$\begin{aligned}
&\ldots \\
&\mathit{filter}\ p\ (x : \mathit{xs})\ &&|\ p\ x = \ldots \\
& &&|\ \mathbf{otherwise} = \ldots
\end{aligned}$$

You will then have to distinguish between the two cases: $p\ x$ and $\neg\ (p\ x)$, which makes the proof more fragmented. Both proofs are okay, however.

4. Reflecting on the law we used in the previous exercise:

$$f\ (\textbf{if}\ q\ \textbf{then}\ e_1\ \textbf{else}\ e_2) = \textbf{if}\ q\ \textbf{then}\ f\ e_1\ \textbf{else}\ f\ e_2$$

   Can you think of a counterexample to the law above, when we allow the presence of $\bot$? What additional constraint shall we impose on f to make the law true?

5. Prove: $take\ n\ xs \mathbin{+\!\!+} drop\ n\ xs = xs$, for all n and xs.

6. Define a function $fan :: a \to List\ a \to List\ (List\ a)$ such that $fan\ x\ xs$ inserts x into the 0th, 1st… nth positions of xs, where n is the length of xs. For example:

$$fan\ 5\ [1,2,3,4] = [[5,1,2,3,4],[1,5,2,3,4],[1,2,5,3,4],[1,2,3,5,4],[1,2,3,4,5]]\ .$$

7. Prove: $map\ (map\ f) \cdot fan\ x\ =\ fan\ (f\ x) \cdot map\ f$, for all f and x. **Hint**: you will need the *map*-fusion law, and to spot that $map\ f \cdot (y\ {:}) = (f\ y\ {:}) \cdot map\ f$ (why?).

8. Define $perms :: List\ a \to List\ (List\ a)$ that returns all permutations of the input list. For example:

$$perms\ [1,2,3] = [[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]\ .$$

   You will need several auxiliary functions defined in the lectures and in the exercises.

9. Prove: $map\ (map\ f) \cdot perm = perm \cdot map\ f$. You may need previously proved results, as well as a property about *concat* and *map*: for all g, we have $map\ g \cdot concat = concat \cdot map\ (map\ g)$.

10. Define $inits :: List\ a \to List\ (List\ a)$ that returns all prefixes of the input list.

$$inits\ \texttt{"abcde"} = [\texttt{""}, \texttt{"a"}, \texttt{"ab"}, \texttt{"abc"}, \texttt{"abcd"}, \texttt{"abcde"}].$$

   Hint: the empty list has *one* prefix: the empty list. The solution has been given in the lecture. Please try it again yourself.

11. Define $tails :: List\ a \to List\ (List\ a)$ that returns all suffixes of the input list.

$$tails\ \texttt{"abcde"} = [\texttt{"abcde"}, \texttt{"bcde"}, \texttt{"cde"}, \texttt{"de"}, \texttt{"e"}, \texttt{""}].$$

   Hint: the empty list has *one* suffix: the empty list. The solution has been given in the lecture. Please try it again yourself.

12. The function $splits :: List\ a \to List\ (List\ a, List\ a)$ returns all the ways a list can be split into two. For example,

$$\begin{aligned} splits\ [1,2,3,4]\ =\ & [([\,],[1,2,3,4]),([1],[2,3,4]),([1,2],[3,4]),\\ & ([1,2,3],[4]),([1,2,3,4],[\,])]\ . \end{aligned}$$

   Define *splits* inductively on the input list. **Hint**: you may find it useful to define, in a **where**-clause, an auxiliary function $f\ (ys, zs)\ =\ \ldots$ that matches pairs. Or you may simply use $(\lambda\ (ys, zs) \to \ldots)$.

13. An *interleaving* of two lists xs and ys is a permutation of the elements of both lists such that the members of xs appear in their original order, and so does the members of ys. Define $interleave :: List\ a \rightarrow List\ a \rightarrow List\ (List\ a)$ such that $interleave\ xs\ ys$ is the list of interleaving of xs and ys. For example, $interleave\ [1,2,3]\ [4,5]$ yields:

$$[[1,2,3,4,5],[1,2,4,3,5],[1,2,4,5,3],[1,4,2,3,5],[1,4,2,5,3],\\ [1,4,5,2,3],[4,1,2,3,5],[4,1,2,5,3],[4,1,5,2,3],[4,5,1,2,3]].$$

14. A list ys is a *sublist* of xs if we can obtain ys by removing zero or more elements from xs. For example, $[2,4]$ is a sublist of $[1,2,3,4]$, while $[3,2]$ is *not*. The list of all sublists of $[1,2,3]$ is:

$$[[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]].$$

Define a function $sublist :: List\ a \rightarrow List\ (List\ a)$ that computes the list of all sublists of the given list. **Hint**: to form a sublist of xs, each element of xs could either be kept or dropped.

15. Consider the following datatype for internally labelled binary trees:

$$\mathbf{data}\ Tree\ a\ =\ \mathrm{Null} \mid \mathrm{Node}\ a\ (Tree\ a)\ (Tree\ a)\ \ .$$

(a) Given $(\downarrow) :: Nat \rightarrow Nat \rightarrow Nat$, which yields the smaller one of its arguments, define $minT :: Tree\ Nat \rightarrow Nat$, which computes the minimal element in a tree. (Note: $(\downarrow)$ is actually called *min* in the standard library. In the lecture we use the symbol $(\downarrow)$ to be brief.)

(b) Define $mapT :: (a \rightarrow b) \rightarrow Tree\ a \rightarrow Tree\ b$, which applies the functional argument to each element in a tree.

(c) Can you define $(\downarrow)$ inductively on Nat?

(d) Prove that for all n and t, $minT\ (mapT\ (n+)\ t) = n + minT\ t$. That is, $minT \cdot mapT\ (n+) = (n+) \cdot minT$.