Functional Programming
Practicals 0

Shin-Cheng Mu

FLOLAC 2022

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 $quad :: \mathrm{Int} \to \mathrm{Int}$ such that $quad\ x$ computes x^4.
   (b) Type in this definition into your working file. Describe, in words, what this function does.

$$\begin{aligned} &twice :: (a \to a) \to (a \to a) \\ &twice\ f\ x = f\ (f\ x)\ \ . \end{aligned}$$

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

$$\begin{aligned} &twice :: (a \to a) \to (a \to a) \\ &twice\ f = f \cdot f\ \ . \end{aligned}$$

   (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.

$$forktimes\ f\ g\ x = f\ x \times g\ x\ \ .$$

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

$$(t \to Int) \to (t \to Int) \to t \to Int\ \ .$$

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: $lift2\ h\ f\ g\ x = h\ (f\ x)\ (g\ x)$. Find out the type of *lift2*. Can you explain its type?

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

1 Definitions and Proofs by Induction

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

$$length\ (xs +\!\!+ ys) = length\ xs + length\ ys\ \ .$$

2. Prove: $sum \cdot concat = sum \cdot map\ sum$.

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

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

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:

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

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 $\perp$? 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, $minT \cdot mapT\ (n+) = (n+) \cdot minT$.

2 Simple Program Calculation

1. Given the definition below, $pos\ x\ xs$ yields the index of the first occurrence of x in *xs*, provided that x occurs in *xs*:

$$pos :: \mathrm{Eq}\ a \Rightarrow a \rightarrow \mathrm{List}\ a \rightarrow \mathrm{Int}$$
$$pos\ x = length \cdot takeWhile\ (x \not\equiv)$$

(What happens when x does not occur in *xs*?) Construct an inductive definition of *pos*.

2. Zipping and mapping.

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

(b) Consider the following definition

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

such that

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

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 $\mathit{select}\ xs = \mathit{zip}\ xs\ (\mathit{delete}\ xs)$. Come up with an inductive definition of *select*. **Hint**: you may find *second* useful.

(c) An alternative specification of *delete* is

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

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

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

and the *map-fusion* law (2) given below.

3. Prove the following *map-fusion* law:

$$\mathit{map}\ f \cdot \mathit{map}\ g = \mathit{map}\ (f \cdot g)\ \ . \tag{2}$$

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

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

Therefore, to compute $\mathit{exp}\ m\ n$, multiplication is called n times: $m \times m \ldots m \times 1$. Can we do better? Yet another way to represent a natural number is to use the binary representation.

(a) The function $\mathit{binary} :: \mathsf{Nat} \to \mathsf{List}\ \mathsf{Bool}$ returns the *reversed* binary representation of a natural number. For example:

$$\begin{aligned}
&\mathit{binary}\ 0 = [\,]\ \ ,\\
&\mathit{binary}\ 1 = [\mathsf{T}]\ \ ,\\
&\mathit{binary}\ 2 = [\mathsf{F},\mathsf{T}]\ \ ,
\end{aligned}$$

$$binary\ 3 = [\mathrm{T}, \mathrm{T}]\ ,$$
$$binary\ 4 = [\mathrm{F}, \mathrm{F}, \mathrm{T}]\ ,$$

where T and F abbreviates True and False. 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.

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

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

(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*.

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 $O(\log_2 n)$.

3 Program Calculation Techniques

1. Consider the internally labelled binary tree:

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

(a) Define $sumT :: \mathrm{ITree\ Int} \to \mathrm{Int}$ that computes the sum of labels in an ITree.

(b) A *baobab tree* is a kind of tree with very thick trunks. An Itree Int is called a baobab tree if every label in the tree is larger than the sum of the labels in its two subtrees. The following function determines whether a tree is a baobab tree:

$$\begin{array}{l}
baobab :: \mathrm{ITree\ Int} \to \mathrm{Bool} \\
baobab\ \mathrm{Null} \qquad\qquad = \mathrm{True} \\
baobab\ (\mathrm{Node}\ x\ t\ u) = baobab\ t \wedge baobab\ u\ \wedge \\
\qquad\qquad\qquad\qquad\quad x > (sumT\ t + sumT\ u)\ .
\end{array}$$

What is the time complexity of *baobab*? Define a variation of *baobab* that runs in time proportional to the size of the input tree by tupling.

2. Recall the externally labelled binary tree:

$$\textbf{data}\ \mathrm{Etree}\ a = \mathrm{Tip}\ a \mid \mathrm{Bin}\ (\mathrm{ETree}\ a)\ (\mathrm{ETree}\ a)\ \ .$$

The function *size* computes the size (number of labels) of a tree, while *repl t xs* tries to relabel the tips of t using elements in xs. Note the use of *take* and *drop* in *repl*:

$$\begin{aligned}
&\mathit{size}\ (\mathrm{Tip}\ _) &&= 1\\
&\mathit{size}\ (\mathrm{Bin}\ t\ u) &&= \mathit{size}\ t + \mathit{size}\ u\ \ .
\end{aligned}$$

$$\begin{aligned}
&\mathit{repl} :: \mathrm{ETree}\ a \rightarrow \mathrm{List}\ b \rightarrow \mathrm{ETree}\ b\\
&\mathit{repl}\ (\mathrm{Tip}\ _)\quad xs = \mathrm{Tip}\ (\mathit{head}\ xs)\\
&\mathit{repl}\ (\mathrm{Bin}\ t\ u)\ xs = \mathrm{Bin}\ (\mathit{repl}\ t\ (\mathit{take}\ n\ xs))\ (\mathit{repl}\ u\ (\mathit{drop}\ n\ xs))\\
&\quad \textbf{where}\ n = \mathit{size}\ t\ \ .
\end{aligned}$$

The function *repl* runs in time $O(n^2)$ where n is the size of the input tree. Can we do better? Try discovering a linear-time algorithm that computes *repl*. **Hint**: try calculating the following function:

$$\begin{aligned}
&\mathit{repTail} :: \mathrm{ETree}\ a \rightarrow \mathrm{List}\ b \rightarrow (\mathrm{ETree}\ b, \mathrm{List}\ b)\\
&\mathit{repTail}\ s\ xs = (???, ???)\ \ ,\\
&\quad \textbf{where}\ n = \mathit{size}\ s\ \ ,
\end{aligned}$$

where the function *repTail* returns a tree labelled by some prefix of xs, together with the suffix of xs that is not yet used (how to specify that formally?).

You might need properties including:

$$\begin{aligned}
\mathit{take}\ m\ (\mathit{take}\ (m+n)\ xs) &= \mathit{take}\ m\ xs\ \ ,\\
\mathit{drop}\ m\ (\mathit{take}\ (m+n)\ xs) &= \mathit{take}\ n\ (\mathit{drop}\ m\ xs)\ \ ,\\
\mathit{drop}\ (m+n)\ xs &= \mathit{drop}\ n\ (\mathit{drop}\ m\ xs)\ \ .
\end{aligned}$$

3. The function *tags* returns all labels of an internally labelled binary tree:

$$\begin{aligned}
&\mathit{tags} :: \mathrm{ITree}\ a \rightarrow \mathrm{List}\ a\\
&\mathit{tags}\ \mathrm{Null} &&= [\,]\\
&\mathit{tags}\ (\mathrm{Node}\ x\ t\ u) &&= \mathit{tags}\ t +\!\!+ [x] +\!\!+ \mathit{tags}\ u\ \ .
\end{aligned}$$

Try deriving a faster version of *tags* by calculating

$$\begin{aligned}
&\mathit{tagsAcc} :: \mathrm{ITree}\ a \rightarrow \mathrm{List}\ a \rightarrow \mathrm{List}\ a\\
&\mathit{tagsAcc}\ t\ ys = \mathit{tags}\ t +\!\!+ ys\ \ .
\end{aligned}$$

4. Recall the standard definition of factorial:

$$\begin{aligned}
&\mathit{fact} :: \mathrm{Nat} \rightarrow \mathrm{Nat}\\
&\mathit{fact}\ 0 = 1\\
&\mathit{fact}\ (\mathbf{1}_+\ n) = \mathbf{1}_+\ n \times \mathit{fact}\ n\ \ .
\end{aligned}$$

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

1. Introduce *factAcc* $n\ m = \ldots$ where m is an accumulating parameter.
2. Express *fact* in terms of *factAcc*.
3. Construct a space efficient implementation of *factAcc*.

5. Define the following function *expAcc*:

$$\begin{aligned}&\mathit{expAcc} :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\&\mathit{expAcc}\ b\ n\ x = x \times \mathit{exp}\ b\ n\ \ .\end{aligned}$$

(a) Calculate a definition of *expAcc* that uses only $O(\log n)$ multiplications to compute b^n. You may assume all the usual arithmetic properties about exponentials. **Hint**: consider the cases when n is zero, non-zero even, and odd.

(b) The derived implementation of *expAcc* shall be tail-recursive. What imperative loop does it correspond to?

6. Recall the standard definition of Fibonacci:

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

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*.