Functional Programming
Practicals

Shin-Cheng Mu

1 Functions

1. Define a function $even :: Int \to Bool$ that determines whether the input is an even number. You may use the following functions:

$$\begin{aligned} mod &:: Int \to Int \to Int \quad , \\ (==) &:: Int \to Int \to Bool \quad . \end{aligned}$$

   (Types of the functions written above are not in their most general form.)

2. Define a function that computes the area of a circle with given radius r (using 22/7 as an approximation to π). The return type of the function might be *Double*.

3. Type in the definition of *smaller* into your working file. Then try the following:
   (a) In GHCi, type `:t smaller` to see the type of *smaller*.
   (b) Try applying it to some arguments, e.g. *smaller* 3 4, *smaller* 3 1.
   (c) In your working file, define a new function $st3 = smaller\ 3$.
   (d) Find out the type of *st3* in GHCi. Try *st3* 4, *st3* 1. Explain the results you see.

4. Type in the definition of *square* in your working file.
   (a) Define a function $quad :: Int \to Int$ such that $quad\ x$ computes x^4.
   (b) Type in this definition into your working file. Describe, in words, what this function does.

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

   (c) Define *quad* using *twice*.

5. Replace the previous *twice* with this definition:

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

   (a) Does *quad* still behave the same?
   (b) Explain in words what this operator $(\cdot)$ does.

6. Let the following identifiers have type:

$$\begin{aligned} f &:: Int \to Char \\ g &:: Int \to Char \to Int \\ h &:: (Char \to Int) \to Int \to Int \\ x &:: Int \\ y &:: Int \\ c &:: Char \end{aligned}$$

   Which of the following expressions are type correct?

1. $(g \cdot f)\ x\ c$
2. $(g\ x \cdot f)\ y$
3. $(h \cdot g)\ x\ y$
4. $(h \cdot g\ x)\ c$
5. $h \cdot g\ x\ c$

You may type the expressions into Haskell and see whether they type check. To define f, for example, include the following in your working file:

$$\begin{aligned} f &:: \mathit{Int} \to \mathit{Char} \\ f &= \mathit{undefined} \end{aligned}$$

However, it is better if you can explain why the answers are as they are.

2 Products and Sums

1. In GHCi, issue the command

   ```
   let x = ((1,'a'), True)
   ```

   This defines a new symbol x, with value $((1, \texttt{'}a\texttt{'}), \mathsf{True})$.

   (a) Find out the type of x by a GHCi command.

   (b) How do you extract the 1 in x? Type an expression $\ldots\ x$ into GHCi such that the result is 1.

   (c) Try to extract $\texttt{'}a\texttt{'}$ and True from x too.

2. Define a function $\mathit{swap} :: (a, b) \to (b, a)$ that, as the name and type suggests, swaps the components

   (a) Define *swap* using pattern matching: $\mathit{swap}\ (x, y) = \ldots$.

   (b) Define *swap* using *fst* and *snd*: $\mathit{swap}\ x = \ldots$.

   (c) Define *swap* using **case**.

3. Define a function $\mathit{half} :: \mathit{Int} \to \mathit{Either}\ \mathit{Int}\ \mathit{Int}$ such that

   - if n is even, $\mathit{half}\ n$ returns $\mathsf{Left}\ k$ with $2 \times k = n$;
   - if n is odd, $\mathit{half}\ n$ returns $\mathsf{Right}\ k$ with $2 \times k + 1 = n$.

   You may use the function *div*. Find out what it does by youself.

4. What are the types of the following expressions?

   (a) $\lambda x \to (\mathit{snd}\ x, \mathit{fst}\ x)$.

   (b) $\lambda f\ x \to f\ x\ x$.

   (c) Define:

   $$\begin{aligned} \mathit{myEither}\ f\ g\ x = \ &\mathbf{case}\ x\ \mathbf{of} \\ &\quad \mathsf{Left}\ y \to f\ y \\ &\quad \mathsf{Right}\ z \to g\ z \quad . \end{aligned}$$

   What is the type of *myEither*?[1]

   (d) $\lambda f\ x\ y \to f\ (\mathit{fst}\ y)\ x$.

   (e) $\lambda f\ x\ y \to \mathit{fst}\ (f\ y\ x)$.

   (f) $\lambda x\ y \to x$.

   (g) $\lambda f\ g\ x \to f\ x\ (g\ x)$.

[1] There is such a function called *either*, which is sometimes quite convenient.

3 Inductively Defined Functions on Lists

1. Define a function $\mathit{fstEven} :: [\mathit{Int}] \to \mathit{Int}$ that returns the first even number of the input list.

2. Define a function $\mathit{hasZero} :: [\mathit{Int}] \to \mathit{Bool}$ that returns True if and only if there is a 0 in the input list.

3. Define a function myLast that takes a list and returns the last (rightmost) element.
   (a) Let the type be $\mathit{myLast} :: [a] \to a$. Define myLast.
   (b) What happens in the previous definition of the input list is empty?
   (c) Define $\mathit{myLast} :: [a] \to \mathit{Maybe}\ a$, which returns $\mathsf{Nothing}$ if the list is empty.

4. Define a function pos such that $\mathit{pos}\ x\ \mathit{xs}$ looks for x in xs and returns its position. For example, find 'a' "abc" yields 0, and find 'a' "bac" yields 1.
   (a) Let the type be $\mathit{pos} :: \mathit{Eq}\ a \Rightarrow a \to [a] \to \mathit{Int}$. In your definition, what happens if x is not in the list?
   (b) Let the type be $\mathit{pos} :: \mathit{Eq}\ a \Rightarrow a \to [a] \to \mathit{Maybe}\ \mathit{Int}$, such that $\mathit{pos}\ x\ \mathit{xs}$ returns $\mathsf{Nothing}$ if x is not in the list.

5. Define $\mathit{myConcat} :: [[a]] \to [a]$ such that, for example $\mathit{myConcat}\ [[1,2,3],[],[4],[5,6]] = [1,2,3,4,5,6]$. **Hint**: use $(+\!\!+)$.

6. Define $\mathit{double} :: [a] \to [a]$ such that, for example, $\mathit{double}\ [1,2,3] = [1,1,2,2,3,3]$.

7. Define $\mathit{interleave} :: [a] \to [a] \to [a]$ such that, for example, $\mathit{interleave}\ [1,2,3,4]\ [5,6,7] = [1,5,2,6,3,7,4]$.

8. Define $\mathit{splitLR} :: [\mathit{Either}\ a\ b] \to ([a],[b])$ such that, for example:

$$\mathit{splitLR}\ [\mathsf{Left}\ 1, \mathsf{Left}\ 3, \mathsf{Right}\ \texttt{'}a\texttt{'}, \mathsf{Left}\ 2, \mathsf{Right}\ \texttt{'}b\texttt{'}] = ([1,3,2], \texttt{"ab"}) \quad .$$

9. Define a function $\mathit{fan} :: a \to [a] \to [[a]]$ such that $\mathit{fan}\ x\ \mathit{xs}$ inserts x into the 0th, 1st... nth positions of xs, where n is the length of xs. For example:

$$\mathit{fan}\ 5\ [1,2,3,4] = [[5,1,2,3,4],[1,5,2,3,4],[1,2,5,3,4],[1,2,3,5,4],[1,2,3,4,5]] \quad .$$

10. Define $\mathit{perms} :: [a] \to [[a]]$ that returns all permutations of the input list. For example:

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

11. Try to define functions inits and tails yourself, and make sure you understand them. Recall that $\mathit{inits}\ [1,2,3] = [[\,],[1],[1,2],[1,2,3]]$, and $\mathit{tails}\ [1,2,3] = [[1,2,3],[2,3],[3],[\,]]$.

4 Inductively Defined Functions on Natural Numbers

1. Define $\mathit{mul} :: \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that $\mathit{mul}\ m\ n = m \times n$, by induction on natural number, using addition $(+)$.

2. Define $\mathit{myMin} :: \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ that returns the smaller of its two arguments. There is a built-in operator (min) for this, but try defining it inductively on natural numbers.

3. Define a function $\mathit{elemAt} :: \mathbb{N} \to [a] \to a$ such that $\mathit{elemAt}\ n\ \mathit{xs}$ yields the nth element of xs.[2]

4. Define a function $\mathit{insertAt} :: \mathbb{N} \to a \to [a] \to [a]$ such that $\mathit{insertAt}\ n\ x\ \mathit{xs}$ inserts x into xs such that the nth element of the new list is x.

[2]This function is denoted (!!) in the standard library.

5 User-Defined Inductive Datatypes

1. Consider the type

$$\textbf{data } \mathit{ETree}\ a \ = \ \mathsf{Tip}\ a \ \mid \ \mathsf{Bin}\ (\mathit{ETree}\ a)\ (\mathit{ETree}\ a) \ .$$

   (a) How is it different from the type *Tree* in the lecture note?

   (b) Define Define $minT :: ETree\ Int \rightarrow Int$, which computes the minimal element in a tree. The operator for binary minimum in Haskell is $min :: Ord\ a \Rightarrow a \rightarrow a \rightarrow a$.

2. Define $minT :: Tree\ Int \rightarrow Int$, which computes the minimal element in a tree. The operator for binary minimum in Haskell is $min :: Ord\ a \rightarrow a \rightarrow a \rightarrow a$. And the largest *Int* in Haskell is denoted by *maxBound*.

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

4. Define $flatten :: Tree\ a \rightarrow [a]$ that traverses a tree and collects all the labels, in-order, in a list. For example,

```
flatten (Node 4 (Node 2 (Node 1 Null Null)
                        (Node 3 Null Null))
                (Node 6 (Node 5 Null Null)
                        (Node 7 Null Null)))
```

   yields $[1, 2, 3, 4, 5, 6, 7]$. **Hint**: use $(+\!\!+)$.

5. A *binary search tree* is a tree of type *Tree a*, with *Ord a*, defined by:

   1. Null is a binary search tree, and
   2. Node x t u is a binary search tree if:
      - every label in t is less than x,
      - every label in u is greater than x, and
      - t and u are also binary search trees.

   Define (assuming that t is a binary search tree):

   (a) $memberT :: Ord\ a \Rightarrow a \rightarrow Tree\ a \rightarrow Bool$, such that $memberT\ x\ t$ determines whether x occurs in t, and

   (b) $insertT :: Ord\ a \Rightarrow a \rightarrow Tree\ a \rightarrow Tree\ a$, such that $insertT\ x\ t$ inserts x into t and still returns a binary tree, if x does not appear in t, and returns t if x is in t.