CSE301 Functional Programming, Fall 2022

In-class exam

This exam consists of 20 questions, each worth 1 point. The questions can be solved in any order, and are divided into five parts (some parts have fewer questions, and are hence worth fewer points). The exam marker thanks you for writing legibly.

1 First-order data types

Question 1.1. Consider the following mutually recursive definitions:

- A wug is a list of mleps
- A mlep is two or three wugs

Give a corresponding pair of mutually recursive definitions of data types *Wug* and *Mlep*, by translating the English descriptions into Haskell.

Question 1.2. The function $take :: Int \to [a] \to [a]$ from the Haskell Prelude take an integer n and a list xs as arguments, and returns the prefix of xs consisting of the first n elements, or all of xs if it has fewer than n elements. Implement *take* using recursion and pattern-matching.

For the following questions, recall the definitions of the data types *Either a b* and *Both a b*:

$$\begin{aligned}&\textbf{data}\ Either\ a\ b = Left\ a \mid Right\ b\\&\textbf{data}\ Both\ a\ b = Pair\ a\ b\end{aligned}$$

Question 1.3. Write pure functions of the following polymorphic types:

$$\begin{aligned}&f :: Either\ a\ (Both\ b\ c) \to Both\ (Either\ a\ b)\ (Either\ a\ c)\\&g :: Both\ (Either\ a\ b)\ (Either\ a\ c) \to Either\ a\ (Both\ b\ c)\end{aligned}$$

Question 1.4. Recall that a type isomorphism $A \cong B$ is given by a pair of functions $f :: A \to B$ and $g :: B \to A$ such that $g\ (f\ x) = x$ and $f\ (g\ y) = y$ for all $x :: A$ and $y :: B$. Prove that it is *impossible* to give a type isomorphism $Either\ a\ (Both\ b\ c) \cong Both\ (Either\ a\ b)\ (Either\ a\ c)$ that is polymorphic in a, b, c.

For the next question, recall the data type of binary trees with labelled leaves:

$$\textbf{data}\ Bin\ a = L\ a \mid B\ (Bin\ a)\ (Bin\ a)$$

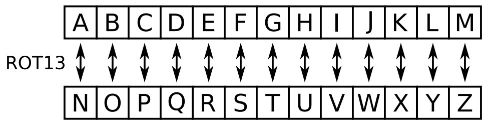
Question 1.5. Consider the following function, which computes the list of leaf labels appearing in a binary tree, in left-to-right order:

$$\begin{aligned}&canopy :: Bin\ a \to [a]\\&canopy\ (L\ x) = [x]\\&canopy\ (B\ t1\ t2) = canopy\ t1 +\!\!+\ canopy\ t2\end{aligned}$$

A deficiency of this implementation is that it has $O(n^2)$ complexity in the worst case, where n is the size of the tree (in particular, it has quadratic behavior on left-branching trees). Rewrite *canopy* so that it computes the same function but with worst case $O(n)$ complexity.

2 Higher-order functions

For the next two questions, you can suppose given a function $\mathit{rot13} :: \mathit{Char} \to \mathit{Char}$ that applies the “ROT13” transformation to a character, as summarized by the following table:



Question 2.1. Using only standard higher-order functions and the function *rot13* mentioned above (and without using any recursion), define a function

$$\mathit{rot13strings} :: [\mathit{String}] \to [\mathit{String}]$$

that applies the ROT13 transformation to a list of strings. For example, you should have *rot13strings* [`"HELLO"`, `"WORLD"`] = [`"URYYB"`, `"JBEYQ"`].

Question 2.2. A “ROT13 pair” is a pair of strings that are ROT13 transformations of each other. For example (`"HELLO"`, `"URYYB"`) is a ROT13 pair. Using only standard higher-order functions and the function *rot13* (and without using any recursion), define a function

$$\mathit{allrot13pairs} :: [(\mathit{String}, \mathit{String})] \to \mathit{Bool}$$

returning *True* just in case the input list contains only ROT13 pairs.

Question 2.3. Recall that the higher-order function $\mathit{filter} :: (a \to \mathit{Bool}) \to [a] \to [a]$ has the following recursive definition

$$\begin{aligned}
&\mathit{filter}\ p\ [\,] = [\,] \\
&\mathit{filter}\ p\ (x : xs) \\
&\quad \mid p\ x = x : \mathit{filter}\ p\ xs \\
&\quad \mid \mathit{otherwise} = \mathit{filter}\ p\ xs
\end{aligned}$$

Prove that

$$\mathit{filter}\ p\ (\mathit{filter}\ q\ xs) = \mathit{filter}\ (\backslash x \to p\ x\ \&\&\ q\ x)\ xs$$

for all lists $xs :: [a]$ and predicates $p, q :: a \to \mathit{Bool}$.

Question 2.4. Write a higher-order function

$$\mathit{unshuffle} :: (a \to \mathit{Either}\ b\ c) \to [a] \to ([b], [c])$$

which takes a function $f :: a \to \mathit{Either}\ b\ c$ and a list $xs :: [a]$ as input, and returns a pair of lists $(ys, zs) = \mathit{unshuffle}\ f\ xs$ such that $\mathit{map}\ f\ xs$ is an interleaving of $\mathit{map}\ \mathit{Left}\ ys$ and $\mathit{map}\ \mathit{Right}\ zs$. For example, letting $f = \backslash x \to \textbf{if}\ \mathit{mod}\ x\ 2 \equiv 0\ \textbf{then}\ \mathit{Left}\ x\ \textbf{else}\ \mathit{Right}\ (\mathit{show}\ x)$, we have $\mathit{unshuffle}\ f\ [1,2,3,4] = ([2,4], [$`"1"`, `"3"`$])$. You can write *unshuffle* using either recursion or other higher-order functions.

Question 2.5. The standard library function $\mathit{inits} :: [a] \to [[a]]$ returns the list of initial prefixes of a list. (For example, *inits* `"hello"` = [`""`, `"h"`, `"he"`, `"hel"`, `"hell"`, `"hello"`].) Define *inits* using the higher-order function *foldr*, without using any recursion. You should express your definition in the form $\mathit{inits} = \mathit{foldr}\ f\ v$, for some f and v.

3 λ-calculus and propositions-as-types

Question 3.1. Compute the β-normal form of the following lambda expression, showing all of the β-reductions that you use to reach the normal form:

$$(\lambda n.\lambda f.\lambda x.f(n\,f\,x))(\lambda f.\lambda x.f(x))$$

Question 3.2. For each of the Haskell functions below, state whether the function is typable, and if so give its most general type.

$f1 = \backslash x\ y \to y$
$f2 = \backslash x\ y \to y\ x\ x$
$f3 = \backslash x\ y\ z \to x\ [y] +\!\!+\ z$

Simply-typed lambda calculus is related to so-called *intuitionistic* logic, which rejects a few classical tautologies such as the law of excluded middle $a \vee \neg a$. However, many classical tautologies can be made intuitionistic by simply double-negating them.

Question 3.3. In Haskell, define a polymorphic function

$notnotlem :: ((Either\ a\ (a \to p)) \to p) \to p$

corresponding to an intuitionistic proof of $\neg\neg(a \vee \neg a)$, where disjunction $a \vee b$ is interpreted by the data type *Either a b* and negation $\neg a$ by the function type $a \to p$. You should write *notnotlem* as a pure Haskell term, without using either recursion or side-effects.

4 Side-effects and monads

Question 4.1. Show that for any fixed type i, the type of functions $i \to a$ is a functor in a, by first completing the class instance definition below:

instance *Functor* $((\to)\ i)$ **where**
  $fmap = undefined :: (a \to b) \to (i \to a) \to (i \to b)$

and then proving that it satisfies the functor laws. (Recall, these state that $fmap\ id = id$ and $fmap\ (f \circ g) = fmap\ f \circ fmap\ g$ for all f and g of appropriate type.)

Question 4.2. Write a function

$summon :: (Monad\ m, Num\ a) \Rightarrow [m\ a] \to m\ a$

that takes a list of monadic computations of numeric type and returns their sum, evaluating the computations from left to right. For example,

summon [*putStrLn* `"Hello"` $\gg$ *return* 1, *putStrLn* `"World!"` $\gg$ *return* 2]

should print

```
Hello
World!
```

and evaluate to 3.

The next two questions refer to the same data type *Bin a* of binary trees with labelled leaves defined in Part 1, as well as the *canopy* function.

Question 4.3. Write a function

$$\mathit{completeBin} :: [\,a\,] \to \mathit{Bin}\ a$$

that takes in a list of values xs and returns a complete[1] binary tree $t = \mathit{completeBin}\ xs$ such that $\mathit{canopy}\ t = xs$, in time $O(n)$ where n is the length of xs. You should define *completeBin* in terms of a helper function

$$\mathit{completeBinST} :: \mathit{Int} \to [\,a\,] \to (\mathit{Bin}\ a, [\,a\,])$$

which builds a complete binary tree in state-passing style. More precisely, $\mathit{completeBinST}\ n\ xs$ should return a pair (t, ys) consisting of a complete binary tree such that $\mathit{canopy}\ t = \mathit{take}\ n\ xs$, together with the remaining elements $ys = \mathit{drop}\ n\ xs$. You can assume that $\mathit{length}\ xs \geqslant n$.

Question 4.4. Write a function

$$\mathit{kthFromRight} :: \mathit{MonadFail}\ m \Rightarrow \mathit{Int} \to \mathit{Bin}\ a \to m\ a$$

such that $\mathit{kthFromRight}\ k\ t$ tries to return the kth leaf from the right of t (where the rightmost leaf counts as the "0th leaf from the right"), or else fails if $k \geqslant \mathit{length}\ xs$. For example,

$$\mathit{kthFromRight}\ 1\ (B\ (B\ (L\ \texttt{'a'})\ (B\ (L\ \texttt{'b'})\ (L\ \texttt{'c'})))\ (L\ \texttt{'d'}))$$

should evaluate to `'c'`. You can use any technique you like, but $\mathit{kthFromRight}\ k\ t$ must evaluate in time $O(k)$. (Reminder: you can use the operation $\mathit{fail} :: \mathit{MonadFail}\ m \Rightarrow \mathit{String} \to m\ a$ to fail with an error message.)

5 Laziness and infinite objects

A *Moore machine* is defined as a tuple $(Q, q_0, I, O, \delta, \rho)$ consisting of the following data:

- a finite set of states Q
- a distinguished state $q_0 \in Q$ called the start state
- a finite set I called the input alphabet
- a finite set O called the output alphabet
- a transition function $\delta : Q \times I \to Q$ mapping a state and a letter of the input alphabet to the next state
- an output function $\rho : Q \to O$ mapping each state to a letter of the output alphabet

An ordinary deterministic finite state automaton may be seen as a Moore machine with the booleans as output alphabet $O = \mathbb{B}$, where the output function ρ labels each state by whether or not it is an accepting state.

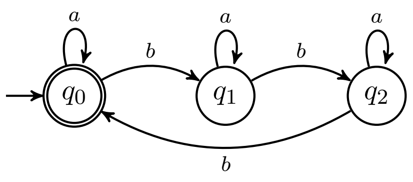
We can represent Moore machines in Haskell as lazy values of the following recursive type:

$$\textbf{data}\ \mathit{Machine}\ i\ o = M\ \{\mathit{out} :: o, \mathit{step} :: i \to \mathit{Machine}\ i\ o\}$$

The idea is that if $m :: \mathit{Machine}\ i\ o$ is a machine, then $\mathit{out}\ m$ gives the output of its start state, while $\mathit{step}\ m\ x$ describes the transition on input x as another Moore machine, which has the successor state as start state.

[1]Recall a binary tree with n leaves is complete if every leaf is at distance $d = \lfloor \log_2(n) \rfloor$ or $d + 1$ from the root.

For instance, the finite state machine:



can be represented as a value *q0* of type *Machine Char Bool*, defined below in mutual recursion with values *q1* and *q2*:

```
q0, q1, q2 :: Machine Char Bool
q0 = M { out = True, step = \x → case x of 'a' → q0; 'b' → q1 }
q1 = M { out = False, step = \x → case x of 'a' → q1; 'b' → q2 }
q2 = M { out = False, step = \x → case x of 'a' → q2; 'b' → q0 }
```

The final output of a Moore machine running on a list of input values can be computed as follows:

```
run :: Machine i o → [i] → o
run m [ ] = out m
run m (x : xs) = run (step m x) xs
```

For example, we have *run q0* `"abbab"` = *True*.

Note that the Haskell type *Machine i o* defined above does not enforce any finiteness restrictions, and in that sense it allows for representing “generalized Moore machines” that do not necessarily have a finite number of states. For clarity, we will refer to a Moore machine with a finite number of states as a “finite Moore machine”.

Question 5.1. Define a finite Moore machine

```
qeven :: Machine a Bool
```

polymorphic in the type *a*, which recognizes input lists of even length. That is, *run qeven xs* should evaluate to *True* if and only if *length xs* is even.

Question 5.2. Define a function

```
reify :: ([i] → o) → Machine i o
```

which turns any function *f* :: [*i*] → *o* into a generalized Moore machine, in such a way that *run* (*reify f*) *xs* = *f xs* for all input lists *xs*.

Question 5.3. Write a function

```
language :: [a] → Machine a Bool → [[a]]
```

which takes as input a finite alphabet Σ (represented as a list of distinct values) and a machine *m*, and lazily constructs the list of all words in Σ^* that are accepted by *m*. You can assume that there are infinitely many such words, and under that assumption you must ensure that for any *n*, *take n* (*language* Σ *m*) evaluates to a list of *n* distinct words in Σ^* that are accepted by *m*.