CSE301 / CSC_3F001_EP Functional Programming

In-class exam (sample questions)

(solutions)

This exam consists of five parts that can be solved in any order. For questions that ask you to write some kind of function, you should assume that the input satisfies whatever conditions are given in the specification of the function. For example, if a question asks you to write a function $sqrt :: Double \to Double$ computing the square root of a non-negative floating point number, you do not need to handle the case that the input is negative.

1 First-order data types

Question 1.1. Define a data type of unary-binary trees with labelled leaves. ("Unary-binary" means that every internal node has 1 or 2 children.)

Solution :

```
data UniBin a = Leaf a
  | Node1 (UniBin a)
  | Node2 (UniBin a) (UniBin a)
```

□

Question 1.2. The function $lookup :: Eq\ a \Rightarrow a \to [(a, b)] \to Maybe\ b$ from the Haskell Prelude tries to find the value associated to a key in a list of key-value pairs, returning *Just* that value if it is exists, or else *Nothing*. Implement *lookup* using recursion and pattern-matching.

Solution :

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
lookup x [] = Nothing
lookup x ((k, v) : kvs)
  | x == k = Just v
  | otherwise = lookup x kvs
```

□

[...]

2 Higher-order functions

Question 2.1. Recall the recursive definition of the higher-order function *map* over lists:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = f x : map f xs
```

Prove that *map* satisfies the functor laws $map\ id\ xs = xs$ and $map\ (f \circ g)\ xs = map\ f\ (map\ g\ xs)$, for all input lists xs and functions f and g of appropriate type.

Solution : We prove both statements proof is by induction on xs.

1. $map\ id\ xs = xs$:

   - Base case: $map\ id\ [\,] = [\,]$ by definition of map
   - Inductive step:

$$\begin{aligned} map\ id\ (x : xs) &= x : map\ id\ xs && \text{by definition of } map \\ &= x : xs && \text{by induction hypothesis} \end{aligned}$$

2. $map\ (f \circ g)\ xs = map\ f\ (map\ g\ xs)$:

   - Base case: $map\ (f \circ g)\ [\,] = [\,] = map\ f\ [\,] = map\ f\ (map\ g\ [\,])$ applying the definition of map three times.
   - Inductive step: $map\ (f \circ g)\ (x : xs) = f\ (g\ x) : map\ (f \circ g)\ xs = f\ (g\ x) : map\ f\ (map\ g\ xs) = map\ f\ (g\ x : map\ g\ xs) = map\ f\ (map\ g\ (x : xs))$ applying the definition of $map\ (f \circ g)$, then the inductive hypothesis, and then the definition of $map\ g$ and $map\ f$.

□

Question 2.2. Using *foldr* (and without using recursion), define a function

$$sumsqr :: Num\ a \Rightarrow [\,a\,] \to a$$

which computes the sum of squares of a list of numbers. (For example, $sumsqr\ [1\,..\,5] = 55$.) You should express your definition in the form $sumsqr = foldr\ f\ v$, for some f and v.

Solution :

$$\begin{aligned} &sumsqr :: Num\ a \Rightarrow [\,a\,] \to a \\ &sumsqr = foldr\ (\backslash x\ n \to x\ \hat{}\ 2 + n)\ 0 \end{aligned}$$

□

[...]

3 λ-calculus and propositions-as-types

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

1. $(\lambda x.\lambda y.y)(\lambda x.x)$
2. $(\lambda x.x\,x)(\lambda y.y)$
3. $\lambda a.(\lambda x.\lambda y.y\,x\,x)a(\lambda b.b)$

Solution :

1. $(\lambda x.\lambda y.y)(\lambda x.x) \to \lambda y.y$
2. $(\lambda x.x\,x)(\lambda y.y) \to (\lambda y.y)(\lambda y.y) \to \lambda y.y$
3. $\lambda a.(\lambda x.\lambda y.y\,x\,x)a(\lambda b.b) \to \lambda a.(\lambda y.y\,a\,a)(\lambda b.b) \to \lambda a.(\lambda b.b)\,a\,a \to \lambda a.a\,a$

□

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

$$f1 = \backslash x \to (x, x)$$
$$f2 = \backslash x \to x\ (\backslash y \to y)$$
$$f3 = \backslash x\ y\ z \to y\ [z\ x, z]$$

Solution :

$$f1 :: a \to (a, a)$$
$$f2 :: ((a \to a) \to b) \to b$$
f3 untypable

□

[...]

4 Side-effects and monads

Question 4.1. Write a function

$$evalmlr :: Monad\ m \Rightarrow m\ a \to m\ b \to m\ c \to m\ (a, b, c)$$

that takes three monadic computations as inputs and evaluates them consecutively in the order *middle-left-right*, returning the resulting triple of values. For example,

$$evalmlr\ (putStr\ \texttt{"La"} \gg return\ 1)\ (putStr\ \texttt{"Di"} \gg return\ 2)\ (putStr\ \texttt{"Da"} \gg return\ 3)$$

should print the output

```
DiLaDa
```

and evaluate to $(1, 2, 3)$.

Solution :

$$evalmlr :: Monad\ m \Rightarrow m\ a \to m\ b \to m\ c \to m\ (a, b, c)$$
$$evalmlr\ mx\ my\ mz = \textbf{do}$$
$$\quad y \leftarrow my$$
$$\quad x \leftarrow mx$$
$$\quad z \leftarrow mz$$
$$\quad return\ (x, y, z)$$

□

[...]

5 Laziness and infinite objects

The *Hamming sequence* is an infinite sequence of numbers $h_1, h_2, \ldots$ defined as follows:

- the first number is $h_1 = 1$;
- the sequence is strictly increasing $h_i < h_{i+1}$ for all i;

- if x is in the sequence (i.e., $x = h_i$ for some i), then so are $2x$, $3x$, and $5x$.

The sequence begins $1, 2, 3, 4, 5, 6, 8, 9, 10, 12, \ldots$

Question 5.1. Define the Hamming sequence in Haskell as a lazy value

$$hamming :: [Integer]$$

For full credit, your definition should have the property that it is possible to compute $h_n = hamming \mathbin{!!} n$ in time $O(n)$.

Solution :

```
hamming :: [Integer]
hamming = 1 : merge3 (map (2*) hamming) (map (3*) hamming) (map (5*) hamming)
  where
    merge3 xs ys zs = merge xs (merge ys zs)
      -- merge two duplicate-free sorted lists to produce a duplicate-free sorted list
    merge :: Ord a => [a] -> [a] -> [a]
    merge (x:xs) (y:ys) | x < y = x : merge xs (y:ys)
        | x == y = x : merge xs ys
        | x > y = y : merge (x:xs) ys
```

□

[...]