

## In-class exam (sample questions)

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  $\text{sqrt} :: \text{Double} \rightarrow \text{Double}$  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.)

**Question 1.2.** The function  $\text{lookup} :: \text{Eq } a \Rightarrow a \rightarrow [(a, b)] \rightarrow \text{Maybe } b$  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 exists, or else *Nothing*. Implement *lookup* using recursion and pattern-matching.

[...]

### 2 Higher-order functions

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

$$\begin{aligned} \text{map} &:: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ \text{map } f [] &= [] \\ \text{map } f (x : xs) &= f x : \text{map } f xs \end{aligned}$$

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

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

$$\text{sumsq} :: \text{Num } a \Rightarrow [a] \rightarrow a$$

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

[...]

### 3 $\lambda$ -calculus and propositions-as-types

**Question 3.1.** Compute the  $\beta$ -normal form of the following lambda expressions, showing all of the  $\beta$ -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.  $\lambda a. (\lambda x. \lambda y. y x x) a (\lambda b. b)$

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

$$\begin{aligned} f1 &= \backslash x \rightarrow (x, x) \\ f2 &= \backslash x \rightarrow x (\backslash y \rightarrow y) \\ f3 &= \backslash x \ y \ z \rightarrow y [z \ x, z] \end{aligned}$$

[...]

## 4 Side-effects and monads

**Question 4.1.** Write a function

$$\text{evalmlr} :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ c \rightarrow m\ (a, b, c)$$

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,

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

should print the output

DiLaDa

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

[...]

## 5 Laziness and infinite objects

The *Hamming sequence* is an infinite sequence of numbers  $h_1, h_2, \dots$  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, ...

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

$$\text{hamming} :: [\text{Integer}]$$

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

[...]