The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 2 MODULE, SPRING SEMESTER 2017-2018

ADVANCED FUNCTIONAL PROGRAMMING

Time allowed TWO hours

Candidates may complete the front cover of their answer book and sign their desk card but must NOT write anything else until the start of the examination period is announced.

Answer ALL FOUR QUESTIONS

Dictionaries are not allowed with one exception. Those whose first language is not English may use a standard translation dictionary to translate between that language and English provided that neither language is the subject of this examination. Subject specific translation dictionaries are not permitted.

No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.

DO NOT turn examination paper over until instructed to do so

ADDITIONAL MATERIAL: Haskell Standard Prelude

Question 1 (functors and applicatives):

a) Define the class `Functor` of functorial types in Haskell, and explain how this definition generalises the familiar `map` function on lists. (3)

b) Given the type declaration

```
data Result a = Fail | Succeed a
```

show how to make the `Result` type into an instance of the `Functor` class, stating the type of the function that you define. (3)

c) Define the class `Applicative` of applicative functors in Haskell, and explain how this definition can be understood in English. (6)

d) Show how to make the `Result` type into an instance of the `Applicative` class, stating the type of each function that you define. (5)

e) Explain the idea of *applicative style*, and use this style to complete the missing parts ? in the sequence of definitions below. You may assume in your definitions that `f` is an an applicative functor: (5)

```
fmap1 :: (a -> b) -> f a -> f b
fmap1 g x = ?

fmap2 :: (a -> b -> c) -> f a -> f b -> f c
fmap2 g x y = ?

fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
fmap3 g x y z = ?
```

f) Given the function definition

```
safediv :: Int -> Int -> Result Int
safediv _ 0 = Fail
safediv n m = Succeed (n ‘div‘ m)
```

explain why the following expression is invalid: (3)

```
pure safediv <*> Succeed 6 <*> Succeed 3
```

Question 2 (monads and state)

a) Show how the notion of a *state transformer* can be represented in Haskell as a parameterised type `ST`, and explain your definition. (4)

b) Define appropriate functions `return` and `>>=` that make `ST` into a monad, and explain your definitions with the aid of pictures. (6)

c) Given the type definition

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

define a *non-monadic* function

```
relabel :: Tree a -> Int -> (Tree Int, Int)
```

that replaces every leaf value in such a tree with a unique or *fresh* integer, by taking the next fresh integer as an additional argument, and returning the next fresh integer as an additional result. (4)

d) Show how the `relabel` function can be redefined in a monadic manner by exploiting the fact that `ST` forms a monad. (8)

e) Why is the monadic definition for `relabel` preferable? (3)

Question 3 (inductive reasoning):

a) Explain the principle of *induction* for the type of finite lists. (5)

b) Given the definitions

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs

reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

and using the fact that

```
length (xs ++ ys)  =  length xs + length ys
```

prove the following property using induction on the list `xs`, justifying each step in your equational reasoning with a short hint: (7)

```
length (reverse xs)  =  length xs
```

c) Show how a more efficient version of `reverse`, called `fastrev`, can be defined in terms of an auxiliary function `rev :: [a] -> [a] -> [a]` whose second argument is an accumulator. Just write down the definitions for `fastrev` and `rev` — there is no need to calculate them. (3)

d) Prove the following property using induction on the list `xs`, justifying each step in your equational reasoning with a short hint:

```
(reverse xs) ++ ys  =  rev xs ys
```

You may assume standard properties of the ++ operator for lists. (7)

e) Explain why `fastrev` is more efficient than `reverse`. (3)

Question 4 (binary trees):

a) Given the type declaration

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

define functions

```
leaves :: Tree a -> [a]
size   :: Tree a -> Int
```

that respectively return the list of leaf values contained in a tree, and the size of a tree (defined as the number of leaves in the tree). (3)

b) A tree is *balanced* if every node has the property that its two subtrees have the same size, with leaves being trivially balanced. Define a function `balanced :: Tree a -> Bool` that decides if a tree is balanced. (5)

c) Define a function `halve :: [a] -> ([a],[a])` that splits a list with an even number of elements into two halves of equal length. (3)

d) Using `halve`, define a function `balance :: [a] -> Tree a` that converts a list whose length is a power of two into the corresponding balanced tree, i.e. for which `leaves (balance xs)` = `xs` for any such list `xs`. (5)

e) Define a recursive function `mirror :: Tree a -> Tree a` that returns the *mirror image* of a tree, in which the leaves appear in reverse order, i.e. `leaves (mirror t)` = `reverse (leaves t)` for any tree `t`. (3)

f) Using your definitions and the fact that

```
size (mirror t)  =  size t
```

prove the following property by induction on the tree `t`, justifying each step in your equational reasoning with a short hint: (6)

```
balanced (mirror t)  =  balanced t
```