The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 2 MODULE, SPRING SEMESTER 2014-2015

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:

a) Define the class `Monad` of monadic types in Haskell, and explain how this definition can be understood in English. (2)

b) Define an instance of the `Monad` class for the `Maybe` type, stating the type for each function that you define. (3)

c) Given the definitions

```
data Expr = Val Int | Add Expr Expr

eval          :: Expr -> Maybe Int
eval (Val n)   = if n == 0 then Nothing else Just n
eval (Add x y) = case eval x of
                     Nothing -> Nothing
                     Just n  -> case eval y of
                        Nothing -> Nothing
                        Just m  -> Just (n + m)
```

show how to rewrite the definition for `eval` using `return` and `>>=` for the `Maybe` monad, and explain why this definition might be preferred. (5)

d) State the three equational laws that every monad must satisfy, and prove that the two *identity* laws hold for the `Maybe` monad. (8)

e) Give two reasons why the monad laws are important. (2)

f) Given the definition

```
data Term a = Val Int | Var a | Add (Term a) (Term a)
```

define an instance of the `Monad` class for the `Term` type, and explain what operation on terms is implemented by the `>>=` function. (5)

Question 2:

a) Given the type declarations

```
type ST a  = State -> (a,State)

type State = Int
```

define the following functions that make `ST` into a monad

```
return :: a -> ST a

(>>=)  :: ST a -> (a -> ST b) -> ST b
```

and explain your definitions in English. (7)

b) Define a *non-monadic* function

```
replace :: [Int] -> Int -> ([Int],Int)
```

that replaces each occurrence of `0` in a list of integers by 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. For example, `replace [0,5,0,7,0] 1` should give `([1,5,2,7,3],4)`. (6)

c) Why is the definition for `replace` problematic? (2)

d) Define a state transformer `fresh :: ST Int` that returns the current state as its result, and the successor of this value as the next state. (2)

e) Using `fresh`, show how the `replace` function can be redefined using the `do` notation by exploiting the fact that `ST` forms a monad. (6)

f) Using `replace`, define a function `relabel :: [Int] -> [Int]` that replaces each occurrence of `0` in a list by a fresh integer, starting from `1`. For example, `relabel [0,5,0,7,0]` should give `[1,5,2,7,3]`. (2)

Question 3:

a) Define an inference rule that formalises the principle of *induction* for the following type of natural numbers, and explain your definition: (5)

```
data Nat = Zero | Succ Nat
```

b) Given the function definition

```
add            :: Nat -> Nat -> Nat
add Zero     y  = y
add (Succ x) y  = Succ (add x y)
```

verify the two properties below using induction, justifying each step in your equational reasoning with a short hint. (8)

```
add x Zero = x

add (add x y) z = add x (add y z)
```

c) Given the function definition

```
product       :: [Int] -> Int
product []     = 1
product (x:xs) = x * product xs
```

explain using the example `product [2,3,4]` why this definition is potentially inefficient in terms of memory usage. (3)

d) Given the specification `product' xs n = n * product xs`, calculate a recursive definition for `product'` using *constructive induction* on `xs`. You may assume standard properties of multiplication. (6)

e) Given the revised definition `product xs = product' xs 1`, explain using `product [2,3,4]` why this definition is potentially more efficient. (3)

Question 4

Suppose that arithmetic expressions built up from integers, addition and multiplication are represented using the following types:

```
data Expr = Val Int | App Op Expr Expr

data Op   = Add | Mul
```

a) Define functions

```
eval   :: Expr -> Int

values :: Expr -> [Int]
```

that respectively evaluate an expression to its integer value, and return the list of integer values contained in an expression. (3)

b) Define a function

```
delete :: Int -> [Int] -> [Int]
```

that deletes the first occurrence (if any) of a value from a list. For example, `delete 2 [1,2,3,2]` should give the result `[1,3,2]`. (3)

c) Using `delete`, define a function

```
perms :: [Int] -> [[Int]]
```

that returns all permutations of a list, given by all possible reorderings of its elements. For example, `perms [1,2,3]` should give: (5)

```
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

d) Using `take` and `drop`, define a function

```
split :: [Int] -> [([Int],[Int])]
```

that returns all splittings of a list into two non-empty parts that append to give the original list. For example, `split [1,2,3,4]` should give: (4)

```
[([1],[2,3,4]),([1,2],[3,4]),([1,2,3],[4])]
```

e) Using `split`, define a function

```
exprs :: [Int] -> [Expr]
```

that returns all expressions whose list of values is a given list. For example, `exprs [1,2,3]` should return all `e` for which `values e = [1,2,3]`. (6)

f) Using your answers to the previous parts, define a function

```
solve :: [Int] -> Int -> [Expr]
```

that returns all expressions whose list of values is a permutation of the given list and whose value is the given value.

For example, `solve [1,2,3,4] 10` should return all expressions `e` for which `values e` is a permutation of `[1,2,3,4]` and `eval e = 10`. (4)