The University Of Sheffield.

Data Provided:
Type-Inference Rules (page 7);
Complexity Calculations (pages 7–8).

DEPARTMENT OF COMPUTER SCIENCE **Spring Semester 2016-2017**

ADVANCED PROGRAMMING TOPICS **2 Hours**

Answer THREE questions only.

All questions carry equal weight. Figures in square brackets indicate the percentage of available marks allocated to each part of a question.

THIS PAGE IS BLANK

1. This question concerns the use of abstract data types (ADTs) to specify and reason about data structures. Some of the questions are based on the following scenario.

   A software developer is designing a new abstract data type $\mathbf{BA}$ (short for "Boolean array"), and chooses to give it the following syntax:

   - $newBA : \mathbf{Int} \to (\mathbf{BA} \cup \mathbf{Err})$
     $newBA\ n$ creates a new array capable of holding up to n values, each of which is initialised to $false$. If $n \leq 0$, it returns an error value, $error$.
   - $set : \mathbf{BA} \to \mathbf{Int} \to \mathbf{Bool} \to \mathbf{BA}$
     $set\ a\ n\ b$ sets the n^{th} element of a equal to b, and returns the resulting array. You should assume that array indexing begins at 1. If n is out of range (ie. if $n \leq 0$ or if there are fewer than n entries in the array), the function returns the original array unchanged.
   - $get : \mathbf{BA} \to \mathbf{Int} \to (\mathbf{Bool} \cup \mathbf{Err})$
     $get\ a\ n$ returns the n^{th} element of a. If n is out of range, the function returns an error value, $error$.
   - $all : \mathbf{BA} \to \mathbf{Bool}$
     $all\ a$ returns $true$ if every entry in a is $true$, and returns $false$ otherwise.
   - $some : \mathbf{BA} \to \mathbf{Bool}$
     $some\ a$ returns $true$ if at least one entry in a is $true$, and returns $false$ otherwise.

   a) Explain what are meant by the

      (i) *sorts*; [5%]

      (ii) *syntax*; [5%]

      (iii) *semantics* [5%]

      of an abstract data type.

   b) In the scenario at the start of Question 1, which of the functions $newBA$, set, get, all and $some$ are

      (i) constructors; [5%]

      (ii) observers; [5%]

      (iii) mutators? [5%]

   c) Write down the semantics for the abstract data type $\mathbf{BA}$. [25%]

   d) Write down a Haskell implementation of the abstract data type $\mathbf{BA}$. [30%]

   e) Say that two arrays are equal if they have the same capacity and their entries at each index are equal. Prove formally that, if n is not out of range, then $set\ a\ n\ (get\ a\ n) = a$ for all finite defined arrays a of type $\mathbf{BA}$. [15%]

2. This question asks you to perform complexity calculations for functions defined in Haskell, and to evaluate their types.

   a) Given the rules provided in "Appendix A: Type-Inference Rules" on page 7, together with the type declarations

   ```
   fmap :: Functor f => (a -> b) -> f a -> f b
   even :: Num a => a -> Bool
   ```

   and the information that `Maybe` is an instance of the class `Functor`, show in detail how to calculate the type of the expression

   ```
   \n -> fmap even (Just n)
   ```

   [50%]

   b) For the purposes of this question you should assume that: Haskell is an eager (ie., not a lazy) language; that `(:)` is a primitive function; and that the function `(++)` is defined by

   ```
   [] ++ ys = ys                    -- [++.0]
   (x:xs) ++ ys = x : (xs ++ ys)    -- [++.1]
   ```

   Consider the following code:

   ```
   data Stream a = Void | AddAtEnd (Stream a) a

   merge :: Stream a -> Stream a -> Stream a
   merge Void s = s                                        -- [m.V]
   merge (AddAtEnd s' x) s = AddAtEnd (merge s' s) x       -- [m.A]

   store :: Stream a -> [a]
   store Void = []                                  -- [s.V]
   store (AddAtEnd s x) = store s ++ [x]            -- [s.A]
   ```

   Using the rules provided in "Appendix B: Complexity Calculations" on pages 7–8, show in detail how to

   (i) find the step-counting function $T_{\texttt{merge}}$ associated with the function `merge`; [20%]

   (ii) calculate the worst-case time complexity of the function `store`. [30%]

3. This question concerns the use of Haskell's class and type systems. Some of the questions are based on the following scenario.

   The type class `Num` includes the following declarations

```
class Num a where
  (+)           :: a -> a -> a  -- addition
  negate        :: a -> a       -- return the negative of a value
```

   and a programmer writes the following implementation of the integers

```
data MyInt = Null | After Value | Before Value
```

   where `Null` represents the value `0`, `After n` represents `(n+1)`, and `Before n` represents `(n-1)`. We'll say that a value of type `MyInt` is *normalised* provided it is defined using `Null` and at most one of the constructors `Before` and `After`. For example, the value `After (Before Null)` contains both `Before` and `After`, so it isn't normalised. The normalised version of `After (Before Null)` — ie., the integer value $((0-1)+1)$ — is just `Null`.

   a) Write down three significant differences between classes in Haskell and classes in Java. [10%]

   b) What Haskell syntax is used to declare an algebraic data type to be an instance of a type class? Give an example. [10%]

   c) For the scenario at the start of Question 3, write down a Haskell implementation of a function `normalise :: MyInt -> MyInt` which returns the normalised version of its input. [20%]

   d) Suppose the programmer has declared `MyInt` to be an instance of the type class `Num`. Write down suitable Haskell implementations for the type `MyInt` of the functions

      (i) `negate` [20%]

      (ii) `(+)` [20%]

   e) The programmer adds the code

```
instance Eq MyInt where
  m == n  =  (normalise m) `equals` (normalise n)
    where equals Null Null             =  True
          equals (After m) (After n)   =  equals m n
          equals (Before m) (Before n) =  equals m n
          equals _ _                   =  False
```

   to the scenario at the start of the question.

   Do the implementations of `negate` and `(+)` you gave for Question 3(d) satisfy the equality `(x + negate x == Null)`, for all finite defined values `x` of type `MyInt`? Justify your answer in detail. [20%]

4. This question concerns the use of structural induction to reason over algebraic data types.

   Consider the following Haskell implementation of the natural numbers:

```
data Nat = Zero | One | Sum Nat Nat

next :: Nat -> Nat
next n = Sum n One                                  -- [next]

two :: Nat
two = next One                                      -- [two]

asInt :: Nat -> Int
asInt n = case n of
  Zero -> 0                                         -- [asInt.0]
  One  -> 1                                         -- [asInt.1]
  Sum m n -> asInt m + asInt n                      -- [asInt.+]

instance Eq Nat where
  m == n = (asInt m == asInt n)                     -- [==]

mult :: Nat -> Nat -> Nat
mult Zero _ = Zero                                  -- [mult.0]
mult One  n = n                                     -- [mult.1]
mult (Sum x y) n = Sum (mult x n) (mult y n)        -- [mult.+]
```

   a) Show, in detail, that `(Sum One two == Sum two One)` evaluates to `True`. [10%]

   b) Write down the principle of structural induction for the algebraic data type `Nat`. [10%]

   c) Prove in detail that `(Sum m n == Sum n m)` evaluates to `True` for all finite defined `m` and `n` of type `Nat`. [10%]

   d) Prove in detail that `(mult m n == mult n m)` evaluates to `True` for all finite defined `m` and `n` of type `Nat`. [40%]

   e) (i) Write down the Haskell definition of a value `inf` of type `Nat` which is *not* a "finite defined value". [10%]

      (ii) Given your definition of `inf`, are the values `(next inf)` and `(Sum inf inf)` equal to one another or different? Justify your answer. [20%]

END OF QUESTION PAPER

Appendix A: Type-Inference Rules

Rule 1. Function Application

$$\begin{array}{rl} \text{If} & f::Cons_f \Rightarrow \sigma \to \tau \\ \text{and} & e::Cons_e \Rightarrow \sigma \\ \text{Deduce} & f\ e::(Cons_f, Cons_e) \Rightarrow \tau \end{array}$$

Rule 2. Type Instantiation

$$\begin{array}{rl} \text{If} & f::Cons_f \Rightarrow \tau, \text{ where } \tau \text{ involves the polymorphic type-variable } a \\ \text{and} & \sigma \text{ is any type} \\ \text{Deduce} & f::Cons_f\{\sigma/a\} \Rightarrow \tau\{\sigma/a\} \end{array}$$

Rule 3. Abstraction

$$\begin{array}{rl} \text{If} & (x::\sigma) \text{ implies } e::Cons_e \Rightarrow \tau \\ \text{Deduce} & (\backslash x \to e)::Cons_e \Rightarrow \sigma \to \tau \end{array}$$

Appendix B: Complexity Calculations

The step-counting function T_f and the cost-function T For any f defined in a Haskell program, its step-counting function is called T_f. In order to compute T_f we need to know how much it costs to evaluate the various types of expression we encounter during execution of the program. We write $T(e)$ for the cost of evaluating the expression e. The rules for evaluating T and T_f are as follows:

cost:case if $f\ a_1\ a_2 \ldots a_n$ is defined directly by an expression of the form $f\ a_1\ a_2 \ldots a_n = e$ then evaluating f requires us first to perform pattern matching to identify the relevant expression as e (assume this takes one step), and then to evaluate it (which takes $T(e)$ steps). So define

$$T_f\ a_1\ a_2 \ldots a_n = 1 + T(e)$$

cost:const if c is a constant, it costs nothing to evaluate c, i.e.

$$T(c) = 0$$

cost:var if v is a variable, it costs nothing to look up the value of v, i.e.

$$T(v) = 0$$

cost:if if e is an expression of the form `if` a `then` b `else` c, then the cost depends on the value of a. In either case we have to evaluate a first (this requires $T(a)$ steps); if a is True, we need to evaluate b (using $T(b)$ steps) and otherwise c (using $T(c)$ steps). So we define

$$T(\texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ c) = T(a) + (\texttt{if}\ a\ \texttt{then}\ T(b)\ \texttt{else}\ T(c))$$

cost:prim If p is a primitive operation (like addition) which is implemented as part of the language, we can assume that the implementation is efficient enough that the cost of calling p can be ignored. Consequently, the cost of evaluating $p\ a_1 \ldots a_n$ is just the cost of evaluating the n different arguments. So

$$pa_1 \ldots a_n = T(a_1) + \cdots + T(a_n)$$

cost:func If f isn't a primitive function, and the particular evaluation $f\ a_1\ a_2 \ldots a_n$ isn't one of the cases used to define f, the cost of evaluating $f\ a_1 \ldots a_n$ is the cost of evaluating the various arguments, *together with* the cost of applying f to those results, i.e.

$$T(f\ a_1 \ldots a_n) = T(a_1) + \cdots + T(a_n) + (T_f\ a_1 \ldots a_n)$$

Recursion starting at 0: If f is defined by

```
f 0 = d
f n = f(n-1) + b.n + c
```

then

$$f(n) = \frac{b}{2}n^2 + \left(c + \frac{b}{2}\right)n + d$$

Recursion starting at 1: If f is defined by

```
f 1 = d
f n = f(n-1) + b.n + c
```

then

$$f(n) = \frac{b}{2}n^2 + \left(c + \frac{b}{2}\right)n + (d - c - b)$$