ECM2418

(with Answers)

Referred / Deferred

UNIVERSITY OF EXETER

COLLEGE OF ENGINEERING, MATHEMATICS AND PHYSICAL SCIENCES

COMPUTER SCIENCE

Referred/Deferred Examination, August 2018

Computer Languages and Representations

Module Leader: Dr David Wakeling

Duration: TWO HOURS

Answer question 1, and any two of the three other questions.

Question 1 is worth 40 marks. Other questions are worth 30 marks each.

No electronic calculators of any sort are to be used during the course of this examination.

This is a CLOSED BOOK examination.

SECTION A (Compulsory Question)

Question 1

(a) Functional programs are sometimes said to suffer from an "embarrassment of parallelism". What is meant by this, and what could be done about it?

BOOKWORK

An "embarrassment of parallelism" means that there are usually far more expressions that could be evaluated in parallel that there are processors to carry out that evaluation **[1 mark]**. The amount of parallel evaluation must therefore be limited to the number of processors available **[1 mark]**. This could either be done automatically by the implementation creating new tasks only when there are processors free **[1 mark]**, or by the programmer using annotations to create tasks only when (after careful consideration) they deem it reasonable **[1 mark]**.

(4 marks)

(b) Trace an application of the Haskell function "`lopper`" below to an argument. Assume the standard definitions of the functions "`reverse`" and "`tail`". State what (if any) type it has, and what (if any) result it computes.

```
lopper
  =  reverse . tail . reverse
```

BOOKWORK

```
lopper xs
  =  reverse (tail (reverse xs))

lopper [1,2,3,4]
  ===>  reverse (tail (reverse [1,2,3,4]))
  ===>  reverse (tail [4,3,2,1])
  ===>  reverse [3,2,1]
  ===>  [1,2,3]

lopper ::  [a] -> [a]
```

The function lopper returns all-but-the-last element of a list.

For trace **(2 marks)**. For type **(1 mark)**. For explanation **(1 mark)**.

(4 marks)

(c) With the aid of examples, explain what distinguishes a Haskell polymorphic type from a monomorphic one, and why one might prefer functions with polymorphic types to ones with monomorphic types.

BOOKWORK

`reverse :: [Int] -> [Int]` – monomorphic **[1 mark]**

`reverse :: [a] -> [a]` – polymorphic **[1 mark]**

A Haskell polymorphic type is distinguished from a monomorphic one because it contains one or more type variables **[1 mark]**. Functions with polymorphic types are preferred to monomorphic ones because they can be used on more occasions **[1 mark]**.

(4 marks)

(d) Write the type of the Haskell function

```
f []
  = ([], [])
f (x:xs)
  |  x `mod` 2 == 0 = (ps, x:qs)
  |  otherwise      = (x:ps, qs)
     where
     (ps, qs) = f xs
```

and say what it computes.

BOOKWORK

The type of the function is `f :: [Int] -> ([Int], [Int])` – type class version also acceptable **[1 mark]**. This function computes a pair of lists of integers from a list of integers **[1 mark]**. The list on the left of the pair is of all the odd integers **[1 mark]**. The list on the right of the pair is of all the even integers **[1 mark]**.

(4 marks)

(e) What is meant by the *most general unifier* of two Prolog literals? Consider the literals `q(X,Y,a)` and `q(g(W),Z,Z)`, and the substitution θ = `{[X/g(b),W/b,Y/a,Z/a]}`. Is θ a unifier? Give your reasons. Is θ the most general unifier? If not, write down the most general unifier for the

two literals and state the result of applying this unifier to them.

BOOKWORK

The most general unifier of literals x and y is the smallest substitution μ such that $x\mu = y\mu$, if any such substitution exists, otherwise it is undefined **(2 marks)**.

The substitution θ is a unifier for the two literals because `q(X,Y,a)`$\theta =$ `q(g(b),a,a)` $=$ `q(g(W),Z,Z)`θ **(2 marks)**.

However, θ is not the most general unifier of the two literals because there is a unifier which is more compact than θ **(2 marks)**.

In particular, consider the substitution $\eta =$ `[X/g(W),Y/a,Z/a]`, which is smaller than θ. The substitution η is the most general unifier of the two literals, and we have that `q(X,Y,a)`$\eta =$ `q(g(W),a,a)` $=$ `q(g(W),Z,Z)`η **(2 marks)**.

(8 marks)

(f) What is the difference between *monotonic* and *non-monotonic* reasoning/inference? Explain what this means intuitively.

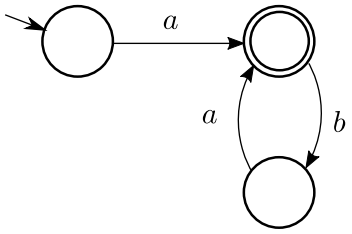
BOOKWORK

We say that a reasoning semantics is monotonic if, for every pair of sets of premises Σ and Σ' such that $\Sigma \subseteq \Sigma'$, the fact that $\Sigma \models \alpha$ implies that $\Sigma' \models \alpha$. Intuitively, this means that, in a monotonic setting, adding new knowledge to the set of premises cannot cause a logical conclusion to be retracted **(3 marks)**.

We say that a reasoning semantics is non-monotonic if there are two sets of premises Σ and Σ' such that $\Sigma \subseteq \Sigma'$, for which $\Sigma \models \alpha$ and $\Sigma' \not\models \alpha$. Intuitively, this means that, in a non-monotonic setting, adding new knowledge to the set of premises may cause a logical conclusion to be retracted **(3 marks)**.

(6 marks)

(g) Consider the following finite-state automaton



(i) State which of the following strings are accepted, and which are rejected, by this automaton

$$\Lambda, a, b, aba, abb.$$

(2 marks)

(ii) Write down a regular expression for the language accepted by this automaton.

(2 marks)

BOOKWORK

(i) Strings accepted: a, aba **[1 mark]**.
Strings rejected: Λ, b, abb **[1 mark]**.

(ii) The regular expression of the language recognised by the automaton is: $a(ba)^*$ **(2 marks)**.

(h) State the three regular operations on a formal language, and give an illustration of each of them.

BOOKWORK

The three regular operations are *union* $(X \cup Y)$, *concatenation* (XY), and *string-set* (X^*) **(3 marks)**.

If $X = \{a\}$ and $Y = \{b\}$ then $X \cup Y = \{a, b\}$, $XY = \{ab\}$, and $X^* = \{\Lambda, a, aa, aaa, aaaa, \ldots\}$ **(3 marks)**.

(6 marks)

(Total 40 marks)

SECTION B (Optional Questions)

Question 2

Study this type of Haskell trees, and then answer the following questions.

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

(a) Write the type and definition of a Haskell function "`insertTree`" that inserts an element into an ordered tree of integers. For example,

```
insertTree 4 (Node (Node Tip 5 Tip) 6 Tip)
  ===>  Node (Node (Node Tip 4 Tip) 5 Tip) 6 Tip
```

```
insertTree ::  Int -> Tree Int -> Tree Int
insertTree e Tip
  =  (Node Tip e Tip)
insertTree e (Node l x r)
  |  e <= x    = Node (insertTree e l) x r
  |  otherwise = Node l x (insertTree e r)
```

For type **[1 mark]**. For first equation **[1 mark]**. For second equation, **[3 marks]**.

(5 marks)

(b) Write the type and definition of a Haskell function "`mapTree`" that maps a function over a tree of values. For example,

```
mapTree (+ 1) ((Node Tip 5 Tip) 6 Tip)
  ===>  Node (Node Tip 6 Tip) 7 Tip
```

```
mapTree ::  (a -> b) -> Tree a -> Tree b
mapTree f Tip
  =  Tip
mapTree f (Node l x r)
  =  Node (mapTree f l) (f x) (mapTree f r)
```

For type **[1 mark]**. For first equation **[1 mark]**. For second equation, **[2 marks]**.

(4 marks)

(c) Write the type and definition of a Haskell function "`foldTree`" that uses a

function and an initial value to fold a tree of values up to a single value. For example,

```
foldTree (+) 0 ((Node Tip 4 Tip) 5 Tip)
  ===>  9
```

```
foldTree ::  (a -> b -> a) -> a -> Tree b -> a
foldTree f z Tip
  =  z
foldTree f z (Node l x r)
  =  foldTree f (foldTree f (f z x) l) r
```

For type **[1 mark]**. For first equation **[1 mark]**. For second equation **[2 marks]**.

(4 marks)

(d) Show how (if at all) the "`mapTree`" function from part (b) could be made suitable for parallel execution by using "`par`" and "`seq`" annotations.

```
parMapTree ::  (a -> b) -> Tree a -> Tree b
parMapTree f Tip
  =  Tip
parMapTree f (Node l x r))
  =  par pl (par pr (seq pl (seq pr (Node pl (f x) pr))))
     where
     pl  = parMapTree f l
     pr =  parMapTree f r
```

For type **[1 mark]**. For first equation **[1 mark]**. For second equation, with annotations **[4 marks]**.

(6 marks)

(e) Show how (if at all) the "`foldTree`" function from part (c) could be made suitable for parallel execution by using "`par`" and "`seq`" annotations.

There is no obvious way to fold a binary tree to a single value using the parallel annotations because of the accumulating nature of the binary operator. The right branch of the tree cannot be folded until the left branch of the tree has been folded to provide the initial value. Even

constraining the folding operator to be associative does not help. The marks here are awarded to the student who can think this through, rather than trying to use the annotations **(6 marks)**.

(6 marks)

(f) Algorithms are more clearly expressed in the functional style that the imperative one. Do you agree? Argue one way or the other.

Either argue yes, without the baggage of sequence and state that comes with the imperative style it is much easier to understand an algorithm expressed in the functional style. A classic example would be the famous quicksort algorithm that is rather mysterious in the imperative style, where arrays are shuffled to produce a partition, but rather obvious in the functional style where the same operation is carried out on lists. A better understanding is bound to lead to a better implementation in any language **[5 marks]**.

Or argue no, it is often difficult to do without the notions of sequence and state that come with the imperative style when trying to understand an algorithm. A classic example would be graph traversal algorithms, where the notion of a "visited" bit at each node is the obvious way to prevent cycles. To do something similar, the functional style requires a rather unnatural auxiliary data structure be dragged around, consulted, and updated. These unnatural data structures will have to be removed as soon as the algorithm is implemented **[5 marks]**.

(5 marks)

(Total 30 marks)

Question 3

(a) What is meant by the Closed World Assumption? Give an example to show that this assumption may sometimes be reasonable in real-world contexts.

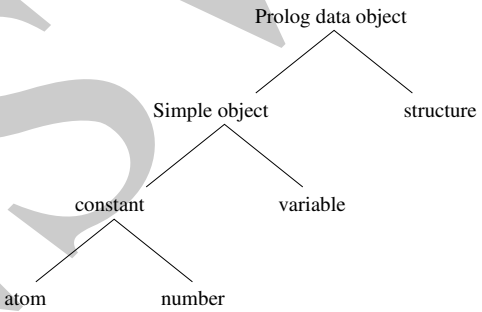
The Closed World Assumption is the assumption that all and only positive information is stored within a Prolog database. Therefore, the information is assumed to be complete in the sense that the database includes or logically implies every true fact about the domain of interest **(3 marks)**.

An example where this is reasonable is a railway timetable: it is reasonable to assume that if the timetable does not show a service running from A to B at time t then there is no such service scheduled in reality **(1 mark)**.

(4 marks)

(b) Outline the classification of data objects in Prolog. Your answer should make clear the structure of the class inclusion hierarchy, and also give a brief description of what is included in each class.

The classification is shown in the tree:



(4 marks)

- Atoms are used as constants, variables, and functors. They are represented by character-strings beginning with a lower-case letter or enclosed in single quotes.
- Numbers can be integers or reals, written in the standard way.
- Variables are character strings beginning with an upper-case letter or the underscore character "_".
- A structure consists of an atom called the functor and a sequence

of one or more Prolog data objects called arguments. Functors are used as both functions and predicates.

((**4 marks**): 1 mark each)

(8 marks)

(c) Consider the following lists in Prolog:

- `[1,2|[3,4]]`
- `[[1,2],[3,4]]`
- `[1,[2,3|[4]]]`
- `[[1,2]|[3,4]]`

Write down the above lists in an expanded way (i.e., without using the "tail" notation with '`|`'). State the length of each list. Are any two of these lists the same?

The lists are `[1,2,3,4]`, `[[1,2],[3,4]]`, `[1,2,3,4]`, `[[1,2],3,4]` **(3 marks)**.

Their lengths are 4, 2, 4, and 3, respectively **(2 marks)**.

The first and the third list are the same **(1 mark)**.

(6 marks)

(d) Consider the following Prolog function:

```
f(A,B,C):- A=[],B=C.
f(A,B,C):- A=[H1|T1],f(T1,B,T),C=[H1|T].
```

- What is the aim of this function?

(5 marks)

- Is there a way to write it more compactly?

(4 marks)

`f(A,B,C)` is the function that concatenates the list in `A` with the list in `B` to obtain the list `C` **(5 marks)**.

A more compact way to write the same function is:

```
append([],X,X).

append([H1|T1],L2,[H1|T]):-append(T1,L2,T).
```

(4 marks).

(e) What is the meaning of the operator '`\+`' in Prolog?

'`\+`' is the Prolog negation-as-failure operator. The query '`?- \+p`' succeeds if and only if the query '`?- p`' fails. **(3 marks)**

(3 marks)

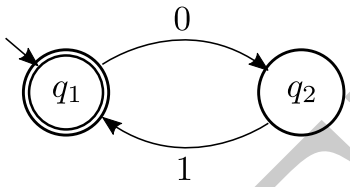
(Total 30 marks)

Question 4

(a) (i) What five elements are required to specify a deterministic finite-state automaton?

(5 marks)

(ii) Consider the following automaton.



What are the five elements for this particular automaton?

(5 marks)

(i) The five elements required are:

- A finite set of states Q
- A finite input alphabet Σ
- A distinguished start state $q_0 \in Q$
- A transition function $\delta : Q \times \Sigma \to Q$
- A distinguished set of accepting states $A \subseteq Q$.

(1 mark each = **(5 marks)**)

(ii) The five elements for the particular automaton are: $Q = \{q_1, q_2\}$, $\Sigma = \{0, 1\}$, $q_0 = q_1$, $A = \{q_1\}$.

The transition function is as follows: $\delta(q_1, 0) = q2$, $\delta(q_1, 1)$ is not defined, $\delta(q_2, 1) = q_1$, and $\delta(q_2, 0)$ is not defined **(5 marks)**.

(b) Consider the language L defined via the regular expression $(aa)^* + (aaa)^*$.

(i) Describe in words the language L.

(2 marks)

(ii) Write two strings belonging to L and two strings not belonging to L, respectively.

(4 marks)

(iii) Define a non-deterministic finite-state automaton recognising L. (**Hint:** devise a machine that non-deterministically decides whether the input string has to be checked against either $(aa)^*$ or $(aaa)^*$.) State where the non-determinism occurs.

(7 marks)

(iv) Transform the automaton for L defined above into a deterministic one.

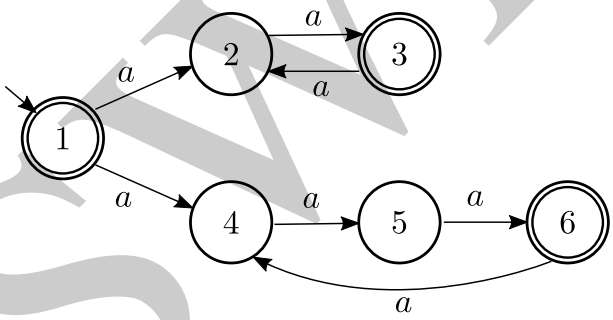
(7 marks)

(i) L is the language of the strings made of a number of 'a' that is either a multiple of 2 or a multiple of 3. The empty string belongs to L. **(2 marks)**

(ii) Strings belonging to L: Λ, aa. (**1 mark** per each string)

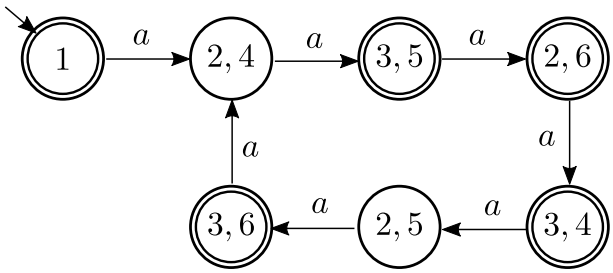
String not belonging to L: $a, aaaaa$. (**1 mark** per each string)

(iii) A non-deterministic automaton recognising L is:



The non-determinism occurs in the state 1 where the machine, when reading a, may move to state 2 or 4. (**(7 marks)**, of which: 1 mark if it is stated correctly where the non-determinism is)

(iv) The deterministic automaton derived from the automaton of the point above is:



(7 marks)

(Total 30 marks)