The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 2 MODULE, SPRING SEMESTER 2011-2012

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 FOUR out of five 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:

Write a short introduction to the *use of monads to structure Haskell programs*, using the example of writing a function

$$eval \quad :: \quad Expr \rightarrow Maybe\ Int$$

that evaluates expressions in the following simple language, in which attempting to divide by zero results in the value *Nothing*:

$$\textbf{data}\ Expr \quad = \quad Val\ Int \mid Div\ Expr\ Expr$$

You may assume that your audience is familiar with the basics of Haskell, but has no previous experience with monads. (25)

Question 2:

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 $\gg=$ that make ST into a monad, and explain your definitions with the aid of pictures. (6)

c) Given the type definition

$$\textbf{data}\ Tree\ a \quad = \quad Leaf\ a \mid Node\ (Tree\ a)\ (Tree\ a)$$

define a *non-monadic* function

$$label \quad :: \quad Tree\ a \rightarrow Int \rightarrow (Tree\ Int, Int)$$

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

d) Show how the *label* 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 *label* preferable? (3)

Question 3:

a) Using equational reasoning, and being explicit about any arithmetic properties being exploited in each step, prove that: (4)

$$(x+a)(x+b) \quad = \quad xx + (a+b)x + ab$$

b) Given the definitions

$$\begin{array}{lcl} (+\!\!+) & :: & [a] \rightarrow [a] \rightarrow [a] \\ [\,] +\!\!+ ys & = & ys \\ (x : xs) +\!\!+ ys & = & x : (xs +\!\!+ ys) \\ \\ reverse & :: & [a] \rightarrow [a] \\ reverse\ [\,] & = & [\,] \\ reverse\ (x : xs) & = & reverse\ xs +\!\!+ [x] \end{array}$$

and using the fact that $+\!\!+$ is associative, prove that: (6)

$$reverse\ (xs +\!\!+ ys) \quad = \quad reverse\ ys +\!\!+ reverse\ xs$$

c) Explain why the above definition for *reverse* is inefficient, and show how a more efficient version can be defined using the technique of *accumulation*, illustrating your new definition using a simple example. (6)

d) Given the definitions

$$\begin{array}{lcl} sum & :: & [Int] \rightarrow Int \\ sum\ [\,] & = & 0 \\ sum\ (x : xs) & = & x + sum\ xs \\ \\ length & = & [a] \rightarrow Int \\ length\ [\,] & = & 0 \\ length\ (x : xs) & = & 1 + length\ xs \end{array}$$

define a function

$$sumlen \quad :: \quad [Int] \rightarrow (Int, Int)$$

that satisfies the specification $sumlen\ xs = (sum\ xs, length\ xs)$, but only makes a single traversal over the list, rather than two. (4)

e) Prove that your definition for *sumlen* satisfies its specification. (5)

Question 4:

a) Given the function definition

$$\begin{array}{lcl} sum & :: & [Int] \rightarrow Int \\ sum\ [\,] & = & 0 \\ sum\ (x : xs) & = & x + sum\ xs \end{array}$$

explain with the aid of a simple example why this definition is potentially inefficient in terms of memory usage. (3)

b) Given the specification $sum'\ xs\ n = n + sum\ xs$, calculate a recursive definition for sum' using *constructive induction* on xs. You may assume standard arithmetic properties of addition. (5)

c) Given the revised definition $sum\ xs = sum'\ xs\ 0$, explain using your previous example why this definition is potentially more efficient. (3)

d) Given the type declarations

$$\begin{array}{lcl} \textbf{data}\ Expr & = & Val\ Int \mid Add\ Expr\ Expr \\ \textbf{type}\ Stack & = & [Int] \\ \textbf{type}\ Code & = & [Op] \\ \textbf{data}\ Op & = & PUSH\ Int \mid ADD \end{array}$$

define three functions

$$\begin{array}{lcl} eval & :: & Expr \rightarrow Int \\ comp & :: & Expr \rightarrow Code \\ exec & :: & Code \rightarrow Stack \rightarrow Stack \end{array}$$

that evaluate an expression to an integer value, compile an expression to code, and execute code using an initial stack to give a final stack. (6)

e) Assuming the distributivity lemma

$$exec\ (xs +\!\!+\ ys)\ s \quad = \quad exec\ ys\ (exec\ xs\ s)$$

verify the compiler correctness property below by induction on e, justifying each step in your equational reasoning with a short hint. (8)

$$exec\ (comp\ e)\ s \quad = \quad (eval\ e) : s$$

Question 5:

Consider the following representation of Sudoku grids:

$$\begin{array}{lcl} \textbf{type}\ \mathit{Grid} & = & \mathit{Matrix}\ \mathit{Int} \\ \textbf{type}\ \mathit{Matrix}\ a & = & [\mathit{Row}\ a] \\ \textbf{type}\ \mathit{Row}\ a & = & [a] \end{array}$$

a) Suppose that you are given functions

$$\begin{array}{lcl} \mathit{rows} & :: & \mathit{Matrix}\ a \rightarrow [\mathit{Row}\ a] \\ \mathit{cols} & :: & \mathit{Matrix}\ a \rightarrow [\mathit{Row}\ a] \\ \mathit{boxs} & :: & \mathit{Matrix}\ a \rightarrow [\mathit{Row}\ a] \\ \mathit{complete} & :: & \mathit{Row}\ a \rightarrow \mathit{Bool} \end{array}$$

that extract the rows, columns and boxes from a matrix, and decide if a row is complete (contains each of the numbers 1 to 9 exactly once). Using these functions, define a function $\mathit{valid} :: \mathit{Grid} \rightarrow \mathit{Bool}$ that decides if all the rows, columns and boxes in a grid are complete. (4)

b) Define a function $\mathit{choices} :: \mathit{Grid} \rightarrow \mathit{Matrix}\ [\mathit{Int}]$ that replaces each zero value (representing a blank entry) in the grid by the list of choices $[1..9]$ for that value, and each non-zero value n by the singleton list $[n]$. (4)

c) Define a function $\mathit{cp} :: [[a]] \rightarrow [[a]]$ that returns the cartesian product of a list of lists, e.g. $\mathit{cp}\ [[1,2],[3,4]] = [[1,3],[1,4],[2,3],[2,4]]$. (4)

d) Using cp, define a function $\mathit{collapse} :: \mathit{Matrix}\ [a] \rightarrow [\mathit{Matrix}\ a]$ that collapses a matrix of choices into a choice of matrices. (3)

e) Using your answers to the previous parts of this question, define a function $\mathit{solve} :: \mathit{Grid} \rightarrow [\mathit{Grid}]$ that solves Sudoku puzzles. Explain why this function is too inefficient to be practically useful. (5)

f) Suppose that you are given a function

$$\mathit{prune}\ ::\ \mathit{Matrix}\ [\mathit{Int}] \rightarrow \mathit{Matrix}\ [\mathit{Int}]$$

that removes all choices that already occur as single entries in the associated row, column or box of a matrix. Using this function, define a new version of *solve* that can instantly solve any "easy" Sudoku puzzle that only requires the repeated application of the basic constraints of the game. (5)