INFOAFP – Exam

Andres Löh

Wednesday, 16 April 2008, 09:00–12:00

Preliminaries

- The exam consists of 7 pages (including this page). Please verify that you got all the pages.
- A maximum of 100 points can be gained.
- For every task, the maximal number of points is stated. Note that the points are distributed unevenly over the tasks.
- One task is marked as (*bonus*) and allows up to 5 extra points.
- Try to give simple and concise answers! Please try to keep your code readable!
- When writing Haskell code, you can use library functions, but make sure that you state which libraries you use.

Good luck!

Evaluation strategies (19 points total)

1 (5 points)**.** Give an example of a Haskell expression of type *Bool* that evaluates to *True* and that would not terminate (i.e., loop forever) in a language with strict evaluation.

Using equational reasoning, give the reduction sequence of your expression to *True* and indicate clearly where a strict reduction strategy would select another redex. •

2 (4 points)**.** Haskell has non-strict semantics (i.e., the implementations use lazy evaluation) and is called a pure language. (S)ML on the other hand has strict semantics and is an impure language.

What does purity mean in this context? Give an (small!) example of why Haskell is considered pure and (S)ML is not. [Syntactic correctness, particularly of (S)ML code, is not important.] •

3 (4 points)**.** Would a lazy impure language or a strict pure language be possible? Would such programming languages be useful? Discuss briefly. •

4 (6 points)**.** Consider the following Haskell functions. Divide the functions into equivalence classes, i.e., group the functions that are semantically equivalent (efficiency is irrelevant). Give as many examples as needed to demonstrate that each of the classes has indeed different behaviour.

$$s1 :: [a] \to b \to b$$
$$s1\ xs\ y = \textbf{case}\ xs\ \textbf{of}\ \{[\,] \to y; \_ \to y\}$$

$$s2 :: [a] \to b \to b$$
$$s2\ xs\ y = seq\ xs\ y$$

$$s3 :: [a] \to b \to b$$
$$s3\ xs\ y = y$$

$$s4 :: [a] \to b \to b$$
$$s4\ xs\ y = \textbf{if}\ null\ xs\ \textbf{then}\ y\ \textbf{else}\ y$$

$$s5 :: [a] \to b \to b$$
$$s5\ xs\ y = \textbf{if}\ map\ (const\ 0)\ xs == [\,]\ \textbf{then}\ y\ \textbf{else}\ y$$

$$s6 :: [a] \to b \to b$$
$$s6\ xs\ y = \textbf{case}\ xs\ \textbf{of}\ \{[\,] \to y; [x] \to y; \_ \to y\}$$

$$s7 :: [a] \to b \to b$$
$$s7\ xs\ y = seq\ [xs]\ y$$

•

Interactive programs (12 points total)

Consider the following datatype:

$$\begin{aligned}\textbf{data}\ GP\ a = {} & End\ a \\ \mid {} & Get\ (Int \to GP\ a) \\ \mid {} & Put\ Int\ (GP\ a)\end{aligned}$$

A value of type *GP* can be used to describe programs that read and write integer values and return a final result of type *a*. Such a program can end immediately (*End*). If it reads an integer, the rest of the program is described as a function depending on this integer (*Get*). If the program writes an integer (*Put*), the value of that integer and the rest of the program are recorded.

The following expression describes a program that continuously reads integers and prints them:

$$echo = Get\ (\lambda n \to Put\ n\ echo)$$

5 (1 point)**.** What is the (inferred) type of *echo*? •

6 (4 points)**.** Write a function

$$run :: GP\ a \to IO\ a$$

that can run a *GP*-program in the *IO* monad. A *Get* should read an integer from the console, and *Put* should write an integer to the console.

Here is an example run from GHCi:

Main⟩ *run echo*
? 42
42
? 28
28
? 1
1
? −5
−5
? `Interrupted.`
Main⟩

[To better distinguish inputs from outputs, this version of *run* prints a question mark when expecting an input. It is not required that your version does the same.] •

7 (3 points)**.** Write a *GP*-program *add* that reads two integers, writes the sum of the two integers, and ultimately returns (). •

8 (4 points)**.** Write a *GP*-program *accum* that reads an integer. If the integer is 0, it returns the current total. If the integer is not 0, it adds the integer to the current total, prints the current total, and starts from the beginning. •

Simulation (21 points total)

9 (4 points)**.** Instead of running a *GP*-program in the *IO* monad, we can also simulate the behaviour of such a program by providing a (possibly infinite) list of input values. Write a function

$$simulate :: GP\ a \rightarrow [Int] \rightarrow (a, [Int])$$

that takes such a list of input values and returns the final result plus the (possibly infinite) list of all the output values generated. •

10 (3 points)**.** What is the result of evaluating the following two expressions?

$$simulate\ accum\ [5, 4 \mathinner{..} 0]$$

$$simulate\ accum\ [5, 4 \mathinner{..} 1]$$

•

11 (4 points)**.** Define a QuickCheck property that states the following property using *simulate*:

"If *echo* is given n numbers as input, then the first n numbers of its output will be identical to the input." •

12 (4 points)**.** Which parts of the definition of *simulate* are covered by your property, and which are not? (I.e., which parts of the definition of *simulate* would be highlighted by HPC after running QuickCheck on your property – assuming that QuickCheck generates suitably random lists.) •

13 (6 points)**.** This is an attempt to define a QuickCheck property for *accum*:

$$
\begin{aligned}
& accumP :: [Int] \rightarrow Property \\
& accumP\ xs = all\ (\lambda x \rightarrow x > 0)\ xs \Longrightarrow \\
& \qquad\qquad simulate\ accum\ (xs +\!\!+ [0]) == (last\ sl, sl) \\
& \quad \textbf{where}\ sl = scanl1\ (+)\ xs
\end{aligned}
$$

Here, *scanl1* is defined as follows

$$
\begin{aligned}
& scanl1 :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow [a] \\
& scanl1\ f\ []\qquad\ \ = [] \\
& scanl1\ f\ (x : xs) = scanl\ f\ x\ xs
\end{aligned}
$$

$$
\begin{aligned}
& scanl :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow [a] \\
& scanl\ f\ x\ xs = x : \textbf{case}\ xs\ \textbf{of} \\
& \qquad\qquad\qquad\quad [] \quad\ \ \rightarrow [] \\
& \qquad\qquad\qquad\quad y : ys \rightarrow scanl\ f\ (f\ x\ y)\ ys
\end{aligned}
$$

There are at least two problems with this property. Describe how they can be fixed [a description is sufficient]. •

Functors and monads (24 points total)

A map function for GP can be defined as follows:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ \mathit{GP}\ \textbf{where} \\
\quad \mathit{fmap}\ f\ (\mathit{End}\ x) \quad = \mathit{End}\ (f\ x) \\
\quad \mathit{fmap}\ f\ (\mathit{Get}\ g) \quad = \mathit{Get}\ (\mathit{fmap}\ f \circ g) \\
\quad \mathit{fmap}\ f\ (\mathit{Put}\ n\ x) = \mathit{Put}\ n\ (\mathit{fmap}\ f\ x)
\end{array}
$$

14 (2 points)**.** Describe the difference between the behaviour of *run accum* and the behaviour of $\mathit{run}\ (\mathit{fmap}\ ({*}2)\ \mathit{accum})$. •

Instances of class *Functor* should generally fulfill the following two laws:

$$
\begin{array}{lll}
\forall x. & \mathit{fmap}\ \mathit{id}\ x & \equiv x \\
\forall f\ g\ x. & \mathit{fmap}\ (f \circ g)\ x & \equiv \mathit{fmap}\ f\ (\mathit{fmap}\ g\ x)
\end{array}
$$

15 (8 points)**.** Prove the **first** of the two laws using equational reasoning (and ignoring that values can be $\perp$).

Note that if you want to prove a property $P\ p$ for any $p :: GP\ a$ via structural induction, you have to prove the following three cases:

$$
\begin{array}{ll}
\forall x. & P\ (\mathit{End}\ x) \\
\forall g. & (\forall x. P\ (g\ x)) \Rightarrow P\ (\mathit{Get}\ g) \\
\forall n\ p. & P\ p \Rightarrow P\ (\mathit{Put}\ n\ p)
\end{array}
$$

(Here, $\Rightarrow$ denotes logical implication.) Note that the second case is slightly unusual due to the function argument of *Get*: you may assume that $P\ (g\ x)$ holds for any value of x! •

16 (5 points)**.** Define a sensible monad instance for *GP*. •

17 (5 points)**.** Define a sensible *MonadState* instance for *GP*. Recall the *MonadState* class:

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MonadState}\ s\ m \mid m \rightarrow s\ \textbf{where} \\
\quad \mathit{get} :: m\ s \\
\quad \mathit{put} :: s \rightarrow m\ ()
\end{array}
$$

•

18 (4 points)**.** What is the difference between the normal state monad as defined in module *Control.Monad.State* and *GP*? Discuss whether you think it is a good idea to make *GP* an instance of *MonadState*. •

Type classes (10 points total, 5 bonus points)

19 (2 points)**.** Consider this program:

$$
\begin{aligned}
&\mathit{equal} :: (\mathit{Eq}\ s, \mathit{MonadState}\ s\ m) \Rightarrow m\ \mathit{Bool} \\
&\mathit{equal} = \mathbf{do} \\
&\qquad\qquad x \leftarrow \mathit{get} \\
&\qquad\qquad y \leftarrow \mathit{get} \\
&\qquad\qquad \mathit{return}\ (x == y)
\end{aligned}
$$

Is the given type signature the most general type signature for *equal*? What would happen if the type signature would be omitted? •

20 (8 points)**.** Translate type classes into explicit evidence in the above function *equal*. Desugar the **do**-notation in the process [use the "simple" desugaring, without the possibility to pattern match on the left hand side of an arrow]. Define the dictionary types that are required – you may omit class methods that are not relevant to this example. You may also declare local abbreviations using **let**. •

21 (5 *bonus points*)**.** Haskell does not offer a scoping mechanism for instances. Instances are always exported from modules, even if nothing else is. Also, instances cannot be local. For example,

$$
\begin{aligned}
&\mathbf{let\ instance}\ \mathit{Eq}\ \mathit{Char}\ \mathbf{where} \\
&\qquad x == y = \mathit{ord}\ (\mathit{toUpper}\ x) == \mathit{ord}\ (\mathit{toUpper}\ y) \\
&\mathbf{in}\ \texttt{"hello"} == \texttt{"HeLlo"}
\end{aligned}
$$

(using *ord* and *toUpper* from *Data.Char*) is not legal Haskell.

Why do you think this decision has been made? Are there any problems you can think of? •

GADTs and kinds (14 points total)

Here is a variation of *GP*:

$$
\begin{array}{l}
\textbf{data}\ \mathit{GP}' :: * \to * \ \textbf{where} \\
\quad \mathit{Return} :: a \to \mathit{GP}'\ a \\
\quad \mathit{Bind} \quad :: \mathit{GP}'\ a \to (a \to \mathit{GP}'\ b) \to \mathit{GP}'\ b \\
\quad \mathit{Get}' \quad\ :: \mathit{GP}'\ \mathit{Int} \\
\quad \mathit{Put}' \quad\ :: \mathit{Int} \to \mathit{GP}'\ ()
\end{array}
$$

This is a GADT. The type GP' can trivially be made an instance of the classes *Monad* and *MonadState*:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ \mathit{GP}'\ \ \textbf{where} \\
\quad \mathit{return} = \mathit{Return} \\
\quad (\gg\!=) \ = \mathit{Bind}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{MonadState}\ \mathit{Int}\ \mathit{GP}'\ \ \textbf{where} \\
\quad \mathit{get}\ = \mathit{Get}' \\
\quad \mathit{put} = \mathit{Put}'
\end{array}
$$

22 (6 points)**.** A value of type *GP* can easily be transformed into a value of type GP' as follows:

$$
\begin{array}{l}
\mathit{gp2gp}' :: \mathit{GP}\ a \to \mathit{GP}'\ a \\
\mathit{gp2gp}'\ (\mathit{End}\ x) \quad = \mathit{Return}\ x \\
\mathit{gp2gp}'\ (\mathit{Get}\ f) \quad\ = \mathit{Get}' \gg\!= \lambda x \to \mathit{gp2gp}'\ (f\ x) \\
\mathit{gp2gp}'\ (\mathit{Put}\ n\ k) = \mathit{Put}'\ n \gg \mathit{gp2gp}'\ k
\end{array}
$$

Define a transformation in the other direction, i.e., a function

$$\mathit{gp}'\mathit{2gp} :: \mathit{GP}'\ a \to \mathit{GP}\ a$$

such that $\mathit{gp}'\mathit{2gp} \circ \mathit{gp2gp}' \equiv \mathit{id}$ for all values that do not contain $\perp$. Does $\mathit{gp2gp}' \circ \mathit{gp}'\mathit{2gp}$ also yield the identity? [No formal proof is required.] •

23 (4 points)**.** Do the monad laws hold for *GP* and GP'? [Give a counterexample if not, argue briefly if yes – no formal proof is required.]

Describe advantages and disadvantages of the two variants. •

24 (4 points)**.** Define type synonyms of kind

$$((* \to *) \to *) \to *$$

and

$$(* \to *) \to (* \to *) \to (* \to *)$$

without using any user-defined **data**types. •