INFOAFP – Exam

Andres Löh

Wednesday, 16 April 2008, 09:00–12:00

1

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.

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

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

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

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

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

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

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

2

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

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

Here, *scanl1* is defined as follows

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

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

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:

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

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

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

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:

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

(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:

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

•

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:

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

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,

```
let instance Eq Char where
      x == y = ord (toUpper x) == ord (toUpper y)
in "hello" == "HeLlo"
```

(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*:

```
data GP' :: * → * where
  Return :: a → GP' a
  Bind   :: GP' a → (a → GP' b) → GP' b
  Get'   :: GP' Int
  Put'   :: Int → GP' ()
```

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

```
instance Monad GP' where
  return = Return
  (>>=)  = Bind

instance MonadState Int GP' where
  get = Get'
  put = Put'
```

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

```
gp2gp' :: GP a -> GP' a
gp2gp' (End x)   = Return x
gp2gp' (Get f)   = Get' >>= λx -> gp2gp' (f x)
gp2gp' (Put n k) = Put' n >> gp2gp' k
```

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

```
gp'2gp :: GP' a -> GP a
```

such that $gp'2gp \circ gp2gp' \equiv id$ for all values that do not contain $\bot$. Does $gp2gp' \circ gp'2gp$ 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. •