Advanced Functional Programming TDA342/DIT260

Patrik Jansson, Dan Rosén, Jonas Duregård

2014-03-11

Contact: Patrik Jansson, ext 5415, Dan Rosén, ext 1741.

Result: Announced no later than 2014-03-30

Exam check: Mo 2014-03-31 and Tu 2014-04-01. Both at 12.45-13.10 in EDIT 5468.

Aids: You may bring up to two pages (on one A4 sheet of paper) of pre-written notes - a "summary sheet". These notes may be typed or handwritten. They may be from any source. If this summary sheet is brought to the exam it must also be handed in with the exam (so make a copy if you want to keep it).

Grades: Chalmers: 3: 24p, 4: 36p, 5: 48p, max: 60p
GU: G: 24p, VG: 48p
PhD student: 36p to pass

Remember: Write legibly.
Don't write on the back of the paper.
Start each problem on a new sheet of paper.
Hand in the summary sheet (if you brought one) with the exam solutions.

Problem 1: Types: read, understand and extend Haskell programs which use advanced type system features

(20 p)

You have seen the standard Haskell definition of *Monad* in the course. Let's repeat the core of it here, and call it *Monad1* to be able to differentiate between it and another version below:

$$\begin{array}{l}\textbf{class}\ \mathit{Monad1}\ m\ \textbf{where}\\ \quad \mathit{return} :: a \to m\ a\\ \quad (\gg\!=)\ :: m\ a \to (a \to m\ b) \to m\ b\end{array}$$

In this task, you will show that another definition is just as expressive. It uses *join* as primitive instead of bind ($\gg\!=$), and it has a *Functor* constraint. We will call this class *Monad2*:

$$\begin{array}{l}\textbf{class}\ \mathit{Functor}\ m \Rightarrow \mathit{Monad2}\ m\ \textbf{where}\\ \quad \mathit{return} :: a \to m\ a\\ \quad \mathit{join}\quad :: m\ (m\ a) \to m\ a\end{array}$$

(5 p) **(a)** Without using do-notation, implement bind using this new monad definition:

$$\begin{array}{l}(\gg\!=) :: \mathit{Monad2}\ m \Rightarrow m\ a \to (a \to m\ b) \to m\ b\\ (\gg\!=) = ?\end{array}$$

Hint: Every Monad2 *is also a* Functor*!*

(5 p) **(b)** Implement *join1* using the standard monad definition:

$$\begin{array}{l}\mathit{join1} :: \mathit{Monad1}\ m \Rightarrow m\ (m\ a) \to m\ a\\ \mathit{join1} = ?\end{array}$$

Again, don't use do-notation.

If *Monad2* was how monads were defined in Haskell, the instances could also look a bit different.

(3 p) **(c)** Finish this *Maybe* instance of *Monad2* by implementing *join*:

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad2}\ \mathit{Maybe}\ \textbf{where}\\ \quad \mathit{return} = \mathit{Just}\\ \quad \mathit{join}\quad = ?\end{array}$$

Do it in this setting's most straightforward way (i.e. don't go via an implementation of ($\gg\!=$)).

(7 p) **(d)** Finish this *State* instance of *Monad2* by implementing *fmap* and *join*:

$$\begin{array}{l}\textbf{newtype}\ \mathit{State}\ s\ a = \mathit{State}\ \{\mathit{runState} :: s \to (a, s)\}\\ \textbf{instance}\ \mathit{Functor}\ (\mathit{State}\ s)\ \textbf{where}\\ \quad \mathit{fmap} = ?\\ \textbf{instance}\ \mathit{Monad2}\ (\mathit{State}\ s)\ \textbf{where}\\ \quad \mathit{return}\ a = \mathit{State}\ \$\ \lambda s \to (a, s)\\ \quad \mathit{join}\qquad = ?\end{array}$$

Again, do it in the most straightforward way for this setting.

Problem 2: Spec: use specification based development techniques (20 p)

This problem continues the previous problem's adventure about monads in terms of *join*.

Note: Even if you have not solved problem 1 you can still try to solve this problem.

The monad laws can also be expressed in terms of *join*, *fmap* and *return*:

Units:	$join \circ return = id = join \circ fmap\ return$
Associativity:	$join \circ fmap\ join = join \circ join$

(a) What is the type of *fmap return* at the use site in the units law and what is the type of the rightmost *join* in the associativity law? (5 p)

(b) Now, consider this implementation of the writer monad: (15 p)

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{Functor}\ ((,)\ w)\ \mathbf{where}\\
&\quad \mathit{fmap}\ f\ (w, a) = (w, f\ a)\\
&\mathbf{instance}\ \mathit{Monoid}\ w \Rightarrow \mathit{Monad2}\ ((,)\ w)\ \mathbf{where}\\
&\quad \mathit{return}\ a \qquad\quad = (\varnothing \qquad , a)\\
&\quad \mathit{join}\ (w, (w', a)) = (w \diamond w', a)
\end{aligned}
$$

Here, for brevity, we write $\varnothing$ for *mempty* and $\diamond$ for *mappend* just as in the appendix. The notation $((,)\ w)$ is a partial application of the pair type constructor. For example, $(((,)\ w)\ a)$ is the same type as (w, a).

Show the above unit and associativity laws for this writer monad by equational reasoning.

Problem 3: DSL: implement embedded domain specific languages (20 p)

This is a simple API for digital circuits of type C:

```
data C    -- To be defined
  -- Primitive operations
inv   :: C → C          -- inverter ("not"-gate)
ands  :: [C] → C        -- "and"-gate with zero or more inputs and one output
delay :: C → C          -- delay the output one step
  -- Derived operations
ors   :: [C] → C        -- "or"-gate with zero or more inputs and one output
xor   :: C → C → C      -- binary "xor"-gate
false, true, toggle :: C
  -- Run functions
run   :: C → [Bool]
show  :: C → String
```

The *run* function should return an infinite list of booleans representing the logic outputs of the circuit for all time steps. Here is a selection of the properties it should satisfy:

$$
\begin{aligned}
&\mathit{prop\_inv}\quad\ \ i\ c = \mathit{run}\ (\mathit{inv}\ c)\ !!\ i \neq \mathit{run}\ c\ !!\ i\\
&\mathit{prop\_delay0}\ \ c = \mathit{not}\ (\mathit{run}\ (\mathit{delay}\ c)\ !!\ 0)\\
&\mathit{prop\_delay}\ \ i\ c = \mathit{run}\ (\mathit{delay}\ c)\ !!\ (i+1) == \mathit{run}\ c\ !!\ i\\
&\mathit{prop\_true}\quad\ i\ \ = \mathit{run}\ \mathit{true}\ !!\ i\\
&\mathit{prop\_toggle}\ i\ \ = \mathit{run}\ \mathit{toggle}\ !!\ i == (i\ \text{`}\mathit{mod}\text{`}\ 2 == 1)
\end{aligned}
$$

(a) Implement the derived operations while keeping the type C abstract. (10 p)

(b) Implement the type C, the primitive operations and *run* using a deep embedding. (10 p)

A Library documentation

A.1 Monoids

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monoid}\ a\ \textbf{where} \\
\quad \mathit{mempty}\ \ :: a \\
\quad \mathit{mappend} :: a \to a \to a
\end{array}
$$

Monoid laws (variables are implicitly quantified, and we write $\varnothing$ for *mempty* and $(\diamond)$ for *mappend*):

$$
\begin{array}{l}
\varnothing \diamond m == m == m \diamond \varnothing \\
(m_1 \diamond m_2) \diamond m_3 == m_1 \diamond (m_2 \diamond m_3)
\end{array}
$$

Example: lists form a monoid:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monoid}\ [a]\ \textbf{where} \\
\quad \mathit{mempty} \qquad\quad = [\,] \\
\quad \mathit{mappend}\ \mathit{xs}\ \mathit{ys} = \mathit{xs} +\!\!+\ \mathit{ys}
\end{array}
$$

A.2 Monads and monad transformers

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{return} :: a \to m\ a \\
\quad (\gg\!=) \ \ :: m\ a \to (a \to m\ b) \to m\ b \\
\quad \mathit{fail} \quad\ :: \mathit{String} \to m\ a \\
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadPlus}\ m\ \textbf{where} \\
\quad \mathit{mzero} :: m\ a \\
\quad \mathit{mplus} :: m\ a \to m\ a \to m\ a
\end{array}
$$

Reader monads

$$
\begin{array}{ll}
\textbf{type}\ \mathit{ReaderT}\ e\ m\ a & \\
\mathit{runReaderT} :: \mathit{ReaderT}\ e\ m\ a \to e \to m\ a & \\
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadReader}\ e\ m \mid m \to e\ \textbf{where} & \\
\quad \mathit{ask} :: m\ e & \text{-- Get the environment} \\
\quad \mathit{local} :: (e \to e) \to m\ a \to m\ a & \text{-- Change the environment locally}
\end{array}
$$

Writer monads

$$
\begin{array}{ll}
\textbf{type}\ \mathit{WriterT}\ w\ m\ a & \\
\mathit{runWriterT}\ \ :: \qquad\qquad\quad\ \ \mathit{WriterT}\ w\ m\ a \to m\ (a, w) & \\
\mathit{execWriterT} :: (\mathit{Monad}\ m) \Rightarrow \mathit{WriterT}\ w\ m\ a \to m\ w & \\
\textbf{class}\ (\mathit{Monad}\ m, \mathit{Monoid}\ w) \Rightarrow \mathit{MonadWriter}\ w\ m \mid m \to w\ \textbf{where} & \\
\quad \mathit{tell} :: w \to m\ () & \text{-- Output something} \\
\quad \mathit{listen} :: m\ a \to m\ (a, w) & \text{-- Listen to the outputs of a computation.}
\end{array}
$$

State monads

$$
\begin{array}{ll}
\textbf{type}\ \mathit{StateT}\ s\ m\ a & \\
\textbf{type}\ \mathit{State}\ \ \ s\ \ \ \ a & \\
\mathit{runStateT} :: \mathit{StateT}\ s\ m\ a \to s \to m\ (a, s) & \\
\mathit{runState}\ \ :: \mathit{State}\ \ \ s\ \ \ \ a \to s \to \ \ \ \ (a, s) & \\
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \to s\ \textbf{where} & \\
\quad \mathit{get} :: m\ s & \text{-- Get the current state} \\
\quad \mathit{put} :: s \to m\ () & \text{-- Set the current state} \\
\quad \mathit{state} :: (s \to (a, s)) \to m\ a & \text{-- Embed a simple state action into the monad}
\end{array}
$$

Error monads

$$
\begin{array}{l}
\textbf{type } \mathit{ErrorT}\ e\ m\ a \\
\mathit{runErrorT} :: \mathit{ErrorT}\ e\ m\ a \to m\ (\mathit{Either}\ e\ a) \\
\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{MonadError}\ e\ m \mid m \to e\ \textbf{where} \\
\quad \mathit{throwError} :: e \to m\ a \qquad \text{-- Throw an error} \\
\quad \mathit{catchError} :: m\ a \to (e \to m\ a) \to m\ a \qquad \text{-- In } \mathit{catchError}\ x\ h \text{ if } x \text{ throws an error,} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{-- it is caught and handled by } h.
\end{array}
$$

A.3 Some QuickCheck

$$
\begin{array}{l}
\quad \text{-- Create Testable properties:} \\
\qquad\qquad \text{-- Boolean expressions: } (\wedge), (|), \mathit{not}, \ldots \\
(==>) :: \mathit{Testable}\ p \Rightarrow \mathit{Bool} \to p \to \mathit{Property} \\
\mathit{forAll}\ :: (\mathit{Show}\ a, \mathit{Testable}\ p) \Rightarrow \mathit{Gen}\ a \to (a \to p) \to \mathit{Property} \\
\qquad\qquad \text{-- ... and functions returning Testable properties}
\end{array}
$$

$$
\begin{array}{l}
\quad \text{-- Run tests:} \\
\mathit{quickCheck} :: \mathit{Testable}\ \mathit{prop} \Rightarrow \mathit{prop} \to \mathit{IO}\ ()
\end{array}
$$

$$
\begin{array}{l}
\quad \text{-- Measure the test case distribution:} \\
\mathit{collect}\ \ :: (\mathit{Show}\ a, \mathit{Testable}\ p) \Rightarrow a \qquad \to p \to \mathit{Property} \\
\mathit{label}\quad :: \mathit{Testable}\ p \Rightarrow \qquad\qquad \mathit{String} \to p \to \mathit{Property} \\
\mathit{classify} :: \mathit{Testable}\ p \Rightarrow \mathit{Bool} \to \mathit{String} \to p \to \mathit{Property} \\
\mathit{collect}\ x = \mathit{label}\ (\mathit{show}\ x) \\
\mathit{label}\ s\ \ = \mathit{classify}\ \mathit{True}\ s
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\quad \text{-- Create generators:}} \\
\mathit{choose} & :: \mathit{Random}\ a \Rightarrow (a, a) & \to \mathit{Gen}\ a \\
\mathit{elements} & :: [a] & \to \mathit{Gen}\ a \\
\mathit{oneof} & :: [\mathit{Gen}\ a] & \to \mathit{Gen}\ a \\
\mathit{frequency} & :: [(\mathit{Int}, \mathit{Gen}\ a)] & \to \mathit{Gen}\ a \\
\mathit{sized} & :: (\mathit{Int} \to \mathit{Gen}\ a) & \to \mathit{Gen}\ a \\
\mathit{sequence} & :: [\mathit{Gen}\ a] & \to \mathit{Gen}\ [a] \\
\mathit{vector} & :: \mathit{Arbitrary}\ a \Rightarrow \mathit{Int} & \to \mathit{Gen}\ [a] \\
\mathit{arbitrary} & :: \mathit{Arbitrary}\ a \Rightarrow & \quad \mathit{Gen}\ a \\
\mathit{fmap} & :: (a \to b) \to \mathit{Gen}\ a & \to \mathit{Gen}\ b \\
\multicolumn{3}{l}{\textbf{instance } \mathit{Monad}\ (\mathit{Gen}\ a)\ \textbf{where} \ldots}
\end{array}
$$

$$
\begin{array}{l}
\quad \text{-- Arbitrary — a class for generators} \\
\textbf{class } \mathit{Arbitrary}\ a\ \textbf{where} \\
\quad \mathit{arbitrary} :: \mathit{Gen}\ a \\
\quad \mathit{shrink}\quad :: a \to [a]
\end{array}
$$