Advanced Functional Programming TDA341/DIT260

Patrik Jansson

2010-03-10

Contact:	Patrik Jansson, ext 5415.
Result:	Announced no later than 2010-03-29
Exam check:	Monday 2010-03-29 and Monday 2010-04-12. Both at 12-13 in EDIT 6125.
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:	3: 24p, 4: 36p, 5: 48p, max: 60p G: 24p, VG: 48p
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.

(20 p)

Problem 1

Consider a DSL for vectors with the following API (similar to the Haskell list operations):

$$
\begin{array}{ll}
\textbf{data}\ \mathit{Vector}\ a & \text{-- to be implemented} \\
\multicolumn{2}{l}{\textbf{type}\ \mathit{Ix} = \mathit{Int}; \textbf{type}\ \mathit{Length} = \mathit{Int}}
\end{array}
$$

$$
\begin{array}{ll}
(+\!\!+) & :: \mathit{Vector}\ a \to \mathit{Vector}\ a \to \mathit{Vector}\ a \\
\mathit{drop} & :: \mathit{Ix} \to \mathit{Vector}\ a \to \mathit{Vector}\ a \\
\mathit{fromFun} & :: \mathit{Length} \to (\mathit{Ix} \to a) \to \mathit{Vector}\ a \\
\mathit{fromList} & :: [a] \to \mathit{Vector}\ a \\
\mathit{head} & :: \mathit{Vector}\ a \to a \\
\mathit{index} & :: \mathit{Vector}\ a \to \mathit{Ix} \to a \\
\mathit{length} & :: \mathit{Vector}\ a \to \mathit{Length} \\
\mathit{splitAt} & :: \mathit{Ix} \to \mathit{Vector}\ a \to (\mathit{Vector}\ a, \mathit{Vector}\ a) \\
\mathit{tail} & :: \mathit{Vector}\ a \to \mathit{Vector}\ a \\
\mathit{take} & :: \mathit{Ix} \to \mathit{Vector}\ a \to \mathit{Vector}\ a
\end{array}
$$

(4 p) **(a)** Classify the operations as constructors, combinators and run functions. Motivate.

(4 p) **(b)** Implement the Vector API using $V\ a$ as the implementation of $\mathit{Vector}\ a$:

$$\textbf{data}\ V\ a = V\ \{\mathit{length} :: \mathit{Length}, \mathit{index} :: \mathit{Ix} \to a\}$$

(4 p) **(c)** Give a deep embedding of Vector as a datatype $D\ a$ with at least 3 constructors and implement *drop*, *length*, *splitAt* and *take*. Identify which operations are primitive and which are derived.

(4 p) **(d)** Specify at least three non-trivial properties (or laws) of the Vector API. Express them as QuickCheck properties (see Appendix A.3 for a reminder).

(4 p) **(e)** Implement a QuickCheck generator for values of type $D\ a$. Avoid generating infinite vectors.

(20 p)

Problem 2

Given monads m and n it is possible to define a product monad $\mathit{Prod}\ m\ n$ as

$$\textbf{newtype}\ \mathit{Prod}\ m\ n\ a = \mathit{Prod}\ \{\mathit{unProd} :: (m\ a, n\ a)\}$$

(5 p) **(a)** Implement the *Monad* instance for $\mathit{Prod}\ m\ n$. You may find these helpers convenient:

$$
\begin{array}{l}
\mathit{fstP} :: \mathit{Prod}\ m\ n\ a\ \ \to m\ a \\
\mathit{fstP} = \mathit{fst} \circ \mathit{unProd} \\
\mathit{sndP} :: \mathit{Prod}\ m\ n\ a \to n\ a \\
\mathit{sndP} = \mathit{snd} \circ \mathit{unProd}
\end{array}
$$

(10 p) **(b)** Prove the three monad laws for your instance:

$$
\begin{array}{lrcl}
\text{Left identity} & \mathit{return}\ a \gg\!= f & \equiv & f\ a \\
\text{Right identity} & \mathit{mx} \gg\!= \mathit{return} & \equiv & \mathit{mx} \\
\text{Associativity} & (\mathit{mx} \gg\!= f) \gg\!= g & \equiv & \mathit{mx} \gg\!= (\lambda x \to f\ x \gg\!= g)
\end{array}
$$

You may use the following lemmas (which hold for (at least) total values):

$$
\begin{array}{lrcl}
\text{Surjective pairing} & p & \equiv & (\mathit{fst}\ p, \mathit{snd}\ p) \\
\text{Eta expansion} & f & \equiv & \lambda x \to f\ x \\
\mathit{fstP}\text{-distributes} & \mathit{fstP}\ (f\ x) \gg\!= (\mathit{fstP} \circ g) & \equiv & \mathit{fstP}\ (f\ x \gg\!= g) \\
\mathit{sndP}\text{-distributes} & \mathit{sndP}\ (f\ x) \gg\!= (\mathit{sndP} \circ g) & \equiv & \mathit{sndP}\ (f\ x \gg\!= g)
\end{array}
$$

(5 p) **(c)** Does "Surjective pairing" hold for all $p :: (a, b)$ in Haskell? Motivate why or why not. Does "Eta expansion" hold for all $f :: a \to b$ in Haskell? Motivate why or why not.

Problem 3

(20 p)

Consider the following Haskell program:

$$
\begin{array}{l}
\textbf{import}\ \mathit{Prod} \quad \text{-- contains the } \mathit{Prod}\ m\ n \text{ monad instance from Problem 2}\\
\textbf{import}\ \mathit{qualified}\ \mathit{Control.Monad.Identity}\ \mathit{as}\quad \mathit{CMI}\\
\textbf{import}\ \mathit{qualified}\ \mathit{Control.Monad.State}\quad \mathit{as}\quad \mathit{CMS}\\
\textbf{import}\ \mathit{qualified}\ \mathit{Control.Monad.Error}\quad \mathit{as}\quad \mathit{CME}\\
\textbf{instance}\ (...) \Rightarrow \mathit{CMS.MonadState}\ \ s\ (\mathit{Prod}\ m\ n)\ \textbf{where}\ ... \quad \text{-- omitted}\\
\textbf{instance}\ (...) \Rightarrow \mathit{CME.MonadError}\ e\ (\mathit{Prod}\ m\ n)\ \textbf{where}\ ... \quad \text{-- omitted}\\
\textbf{type}\ \mathit{Store} = \mathit{Integer}\\
\textbf{type}\ \mathit{Err}\quad = \mathit{String}\\
\textbf{newtype}\ \mathit{Eval1}\ a = \mathit{Eval1}\ \{\mathit{unEval1} :: \mathit{CMS.StateT}\ \mathit{Store}\ (\mathit{CME.ErrorT}\ \mathit{Err}\ \mathit{CMI.Identity})\ a\}\\
\quad \textbf{deriving}\ (\mathit{Monad}, \mathit{CMS.MonadState}\ \mathit{Store}, \mathit{CME.MonadError}\ \mathit{Err})\\
\textbf{newtype}\ \mathit{Eval2}\ a = \mathit{Eval2}\ \{\mathit{unEval2} :: \mathit{CME.ErrorT}\ \mathit{Err}\ (\mathit{CMS.StateT}\ \mathit{Store}\ \mathit{CMI.Identity})\ a\}\\
\quad \textbf{deriving}\ (\mathit{Monad}, \mathit{CMS.MonadState}\ \mathit{Store}, \mathit{CME.MonadError}\ \mathit{Err})\\
\mathit{startStateFrom} :: \mathit{Monad}\ m \Rightarrow \mathit{state} \to \mathit{CMS.StateT}\ \mathit{state}\ m\ a \to m\ a\\
\mathit{startStateFrom} = \mathit{flip}\ \mathit{CMS.evalStateT}\\
\mathit{emptyStore} :: \mathit{Store}\\
\mathit{emptyStore} = 0\\
\mathit{runEval1} :: \mathit{Eval1}\ a \to \mathit{Either}\ \mathit{Err}\ a\\
\mathit{runEval1} = \mathit{CMI.runIdentity} \circ \mathit{CME.runErrorT} \circ \mathit{startStateFrom}\ \mathit{emptyStore} \circ \mathit{unEval1}\\
\mathit{runEval2} :: \mathit{Eval2}\ a \to \mathit{Either}\ \mathit{Err}\ a\\
\mathit{runEval2} = \mathit{CMI.runIdentity} \circ \mathit{startStateFrom}\ \mathit{emptyStore} \circ \mathit{CME.runErrorT} \circ \mathit{unEval2}\\
(\twoheadrightarrow\!\!\!\!\twoheadleftarrow) :: (a1 \to a2) \to (b1 \to b2) \to (a1, b1) \to (a2, b2)\\
f \twoheadrightarrow\!\!\!\!\twoheadleftarrow g = \lambda(a, b) \to (f\ a, g\ b)\\
\textbf{type}\ \mathit{Test} = \mathit{Prod}\ \mathit{Eval1}\ \mathit{Eval2}\\
\mathit{check} :: \mathit{Test}\ a \to (\mathit{Either}\ \mathit{Err}\ a, \mathit{Either}\ \mathit{Err}\ a)\\
\mathit{check} = (\mathit{runEval1} \twoheadrightarrow\!\!\!\!\twoheadleftarrow \mathit{runEval2}) \circ \mathit{unProd}\\
\mathit{test1} :: (\mathit{CME.MonadError}\ \mathit{Err}\ m, \mathit{CMS.MonadState}\ \mathit{Store}\ m) \Rightarrow m\ \mathit{Store}\\
\mathit{test1} = (\textbf{do}\ \mathit{CMS.put}\ 1738; \mathit{CME.throwError}\ \texttt{"hello"}; \mathit{CMS.get})\\
\quad `\mathit{CME.catchError}`\ \lambda e \to \mathit{CMS.get}\\
\mathit{main} = \mathit{print}\ (\mathit{check}\ \mathit{test1})
\end{array}
$$

(a) What would the types *Eval1*, *Eval2* look like without using anything from *Control.Monad.*∗ (expand out the types and simplify away the **newtype**s)? (4 p)

(b) What does *main* print? Motivate. (6 p)

(c) At what type is *test1* used in *main*? Why is it defined with a more general type? (4 p)

(d) Use monad transformers to extend the original *Eval1*, *runEval1* to *Eval3*, *runEval3* adding read-only access to an environment *Env*. Annotate the definition of *runEval3* with the types at the intermediate stages of the "composition pipeline". (For the "pipeline" $f \circ g \circ h$ that would be the return types of g and h.) (6 p)

A Library documentation

A.1 Monoids

```
class Monoid a where
  mempty :: a
  mappend :: a → a → a
```

A monoid should satisfy the laws

$$\begin{aligned}
\mathsf{mappend}\ \mathsf{mempty}\ m &= m \\
\mathsf{mappend}\ m\ \mathsf{mempty} &= m \\
\mathsf{mappend}\ (\mathsf{mappend}\ m_1\ m_2)\ m_3 &= \mathsf{mappend}\ m_1\ (\mathsf{mappend}\ m_2\ m_3)
\end{aligned}$$

List is a monoid:

```
instance Monoid [a] where
  mempty = []
  mappend xs ys = xs ++ ys
```

A.2 Monads and monad transformers

```
class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b
class MonadTrans t where
  lift :: Monad m ⇒ m a → t m a
```

Reader monads

```
type ReaderT e m a
runReaderT :: ReaderT e m a → e → m a
class Monad m ⇒ MonadReader e m | m → e where
    -- Get the environment
  ask :: m e
    -- Change the environment for a given computation
  local :: (e → e) → m a → m a
```

Writer monads

```
type WriterT w m a
runWriterT :: WriterT w m a → m (a, w)
class (Monad m, Monoid w) ⇒ MonadWriter w m | m → w where
    -- Output something
  tell :: w → m ()
    -- Listen to the outputs of a computation.
  listen :: m a → m (a, w)
```

State monads

$$
\begin{array}{l}
\textbf{type}\ \mathit{StateT}\ s\ m\ a \\
\mathit{runStateT} :: \mathit{StateT}\ s\ m\ a \rightarrow s \rightarrow m\ (a, s) \\
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \rightarrow s\ \textbf{where} \\
\quad\quad \text{-- Get the current state} \\
\quad \mathit{get} :: m\ s \\
\quad\quad \text{-- Set the current state} \\
\quad \mathit{put} :: s \rightarrow m\ ()
\end{array}
$$

Error monads

$$
\begin{array}{l}
\textbf{type}\ \mathit{ErrorT}\ e\ m\ a \\
\mathit{runErrorT} :: \mathit{ErrorT}\ e\ m\ a \rightarrow m\ (\mathit{Either}\ e\ a) \\
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadError}\ e\ m \mid m \rightarrow e\ \textbf{where} \\
\quad\quad \text{-- Throw an error} \\
\quad \mathit{throwError} :: e \rightarrow m\ a \\
\quad\quad \text{-- If the first computation throws an error, it is} \\
\quad\quad \text{-- caught and given to the second argument.} \\
\quad \mathit{catchError} :: m\ a \rightarrow (e \rightarrow m\ a) \rightarrow m\ a
\end{array}
$$

A.3 Some QuickCheck

$$
\begin{array}{ll}
\multicolumn{2}{l}{\quad \text{-- Create Testable properties:}} \\
\multicolumn{2}{l}{\quad\quad\quad \text{-- Boolean expressions: } (\wedge), (|), \neg, \ldots} \\
(==>) & :: \mathit{Testable}\ p \Rightarrow \mathit{Bool} \rightarrow p \rightarrow \mathit{Property} \\
\mathit{forAll} & :: (\mathit{Show}\ a, \mathit{Testable}\ p) \Rightarrow \mathit{Gen}\ a \rightarrow (a \rightarrow p) \rightarrow \mathit{Property} \\
\multicolumn{2}{l}{\quad\quad\quad \text{-- ... and functions returning Testable properties}}
\end{array}
$$

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

$$
\begin{array}{llll}
\multicolumn{4}{l}{\quad \text{-- Measure the test case distribution:}} \\
\mathit{collect} & :: (\mathit{Show}\ a, \mathit{Testable}\ p) \Rightarrow a & \rightarrow p \rightarrow \mathit{Property} \\
\mathit{label} & :: \mathit{Testable}\ p \Rightarrow & \mathit{String} \rightarrow p \rightarrow \mathit{Property} \\
\mathit{classify} & :: \mathit{Testable}\ p \Rightarrow \mathit{Bool} \rightarrow & \mathit{String} \rightarrow p \rightarrow \mathit{Property}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{collect}\ x & = \mathit{label}\ (\mathit{show}\ x) \\
\mathit{label}\ s & = \mathit{classify}\ \mathit{True}\ s
\end{array}
$$

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

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