Advanced Functional Programming TDA342/DIT260

Patrik Jansson

2012-03-07

Contact:	Patrik Jansson, ext 5415.
Result:	Announced no later than 2012-03-27
Exam check:	Th 2012-03-29 and Fr 2012-03-30. 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.

(30 p)

Problem 1: Spec: use specification based development techniques

Instances of the *MonadState* class should satisfy the following four laws (where all unbound variables are implicitly forall-quantified and $skip = return\ ()$):

$$
\begin{array}{lll}
put\ s' \gg put\ s & == put\ s & \text{-- put-put} \\
put\ s \gg get & == put\ s \gg return\ s & \text{-- put-get} \\
get \gg\!= put & == skip & \text{-- get-put} \\
get \gg\!= \lambda s \to get \gg\!= \lambda s' \to k\ s\ s' & == get \gg\!= \lambda s \to k\ s\ s & \text{-- get-get}
\end{array}
$$

Consider the following implementation:

```
data S2 s a where
  Return :: a → S2 s a
  Bind   :: S2 s a → (a → S2 s b) → S2 s b
  Then   :: S2 s a → S2 s b → S2 s b
  Get    :: S2 s s
  Put    :: s → S2 s ()
instance Monad (S2 s) where {return = Return; (>>=) = Bind; (>>) = Then}
instance MonadState s (S2 s) where {get = Get; put = Put}
```

(15 p) **(a)** Implement a run function $runS2 :: S2\ s\ a \to (s \to (a, s))$ and prove (by equational reasoning) that the put-put and put-get laws hold if $(==)$ means "all runs are equal".

(15 p) **(b)** An alternative implementation is $S3\ s\ a$ where *Bind* and *Then* have been combined with *Get* and *Put*. Below is a partial implementation of an optimiser from *S2* to *S3*:

```
data S3 s a where
  Ret3    :: a → S3 s a
  GetBind :: (s → S3 s a) → S3 s a
  PutThen :: s → S3 s a   → S3 s a
opt :: S2 s a → S3 s a
opt (Return a)  = Ret3 a
opt Get         = get3
opt (Put s)     = put3 s
opt (Bind m f)  = removeBind m f
opt (Then m n)  = removeThen m n
put3 :: s → S3 s ()
put3 s = PutThen s (Ret3 ())
get3 :: S3 s s
get3 = GetBind Ret3
removeBind :: S2 s a → (a → S2 s b) → S3 s b
removeThen :: S2 s a → S2 s b → S3 s b
```

Implement *removeBind* and *removeThen* and motivate your definitions with the monad laws.

$$
\begin{array}{lll}
return\ x \gg\!= f & == f\ x & \text{-- Law 1} \\
m \gg\!= return & == m & \text{-- Law 2} \\
(m \gg\!= f) \gg\!= g & == m \gg\!= (\lambda x \to f\ x \gg\!= g) & \text{-- Law 3}
\end{array}
$$

Problem 2: DSL: design embedded domain specific languages (15 p)

A pretty-printing library has the following API (inspired by RWH Chapter 5):

$$
\begin{array}{lll}
\mathit{empty} & :: \mathit{Doc} & \\
\mathit{char} & :: \mathit{Char} \to \mathit{Doc} & \\
\mathit{text} & :: \mathit{String} \to \mathit{Doc} & \\
\mathit{line} & :: \mathit{Doc} & \text{-- newline} \\
(<>) & :: \mathit{Doc} \to \mathit{Doc} \to \mathit{Doc} & \text{-- append} \\
\mathit{union} & :: \mathit{Doc} \to \mathit{Doc} \to \mathit{Doc} & \text{-- a choice of two variants only differing in layout} \\
\mathit{prettys} & :: \mathit{Doc} \to [\mathit{String}] & \text{-- all layout variants in order of increasing width}
\end{array}
$$

The width of a string (which can contain newlines) is the length of its longest line. A usage example could be *prettys d2* with

$$
\begin{array}{l}
\mathit{d1} = \mathit{union}\ (\mathit{text}\ \texttt{"x<-m"})\ (\mathit{text}\ \texttt{"x <- m"}) \\
\mathit{d2} = \mathit{union}\ (\mathit{text}\ \texttt{"do \{"} <> \mathit{d1} <> \mathit{char}\ \texttt{';'} <> \mathit{text}\ \texttt{"f x\}"}) \\
\qquad\qquad\quad (\mathit{text}\ \texttt{"do "}\ <> \mathit{d1} <> \mathit{line} <> \mathit{text}\ \texttt{"   f x"})
\end{array}
$$

The four variants would look as follows (widths 7, 9, 13, 15):

```
do x<-m        do x <- m        do {x<-m;f x}        do {x <- m;f x}
   f x            f x
```

(a) Implement a datatype *Doc*, and the operations of the API. This is intended to be just a "proof-of-concept" or "model"-implementation, there is no need to be efficient. (10 p)

(b) Is your implementation deep or shallow? Are you using any monads (explain)? Would some of the API operations fit the *Monoid* type class (explain)? (5 p)

Problem 3: Types: read, understand and extend Haskell programs which use advanced type system features (15 p)

Some features of dependently typed languages like Agda can be simulated in Haskell using GADTs or type families.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Zero} \\
\textbf{data}\ \mathit{Suc}\ n \\[4pt]
\textbf{data}\ \mathit{Vec}\ a\ n\ \textbf{where} \\
\quad \mathit{Nil}\ \ :: \mathit{Vec}\ a\ \mathit{Zero} \\
\quad \mathit{Cons} :: a \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (\mathit{Suc}\ n) \\[4pt]
\textbf{type family}\ \ \mathit{Add}\ m\ \qquad n :: * \\
\textbf{type instance}\ \mathit{Add}\ \mathit{Zero}\ \quad n = n \\
\textbf{type instance}\ \mathit{Add}\ (\mathit{Suc}\ m)\ n = \mathit{Suc}\ (\mathit{Add}\ m\ n)
\end{array}
$$

(a) Give the signature and implementation of $(+\!\!+)$ for vector concatenation and explain why it type checks. Would it still type check with the alternative definition of type-level addition below? Why/why not? (5 p)

$$
\begin{array}{l}
\textbf{type family}\ \ \mathit{Add'}\ m\ n\ \qquad :: * \\
\textbf{type instance}\ \mathit{Add'}\ m\ \mathit{Zero}\ \quad = m \\
\textbf{type instance}\ \mathit{Add'}\ m\ (\mathit{Suc}\ n) = \mathit{Suc}\ (\mathit{Add'}\ m\ n)
\end{array}
$$

(b) Implement a GADT *Fin n* for unary numbers below *n* and a lookup function (5 p)

$$(!) :: \mathit{Vec}\ a\ n \to \mathit{Fin}\ n \to a$$

(c) Briefly explain the Curry-Howard correspondence for "false", "true", "implies", "and", "or". (5 p)

A Library documentation

A.1 Monoids

```haskell
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
```

Monoid laws (variables are implicitly quantified, and we write 0 for *mempty* and (+) for *mappend*):

$$0 + m == m$$
$$m + 0 == m$$
$$(m_1 + m_2) + m_3 == m_1 + (m_2 + m_3)$$

Example: lists form a monoid:

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

A.2 Monads and monad transformers

```haskell
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  fail   :: String -> m a
class MonadTrans t where
  lift :: Monad m => m a -> t m a
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Reader monads

```haskell
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 locally
  local :: (e -> e) -> m a -> m a
```

Writer monads

```haskell
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

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

Error monads

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

A.3 Some QuickCheck

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

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

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

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

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