Advanced Functional Programming TDA342/DIT260

Patrik Jansson

2015-03-17

Contact:	Patrik Jansson, ext 5415.
Result:	Announced no later than 2015-04-03
Exam check:	Mo 2015-04-13 and Tu 2015-04-14. 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.

(20 p)

Problem 1: DSL: design an embedded domain specific language

The following API (From RWH, chapter 9) is aimed at describing predicates for searching the file system in the style of the command-line utility `find`:

$$
\begin{aligned}
&\textbf{type } \mathit{InfoP}\ a = \mathit{FilePath} && \text{-- path to directory entry}\\
&\qquad\qquad \to \mathit{Permissions} && \text{-- permissions}\\
&\qquad\qquad \to \mathit{Integer} && \text{-- file size}\\
&\qquad\qquad \to \mathit{UTCTime} && \text{-- last modified}\\
&\qquad\qquad \to a
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{constP} &&:: a \to \mathit{InfoP}\ a\\
&\mathit{liftPc} &&:: (a \to b \to c) \to \mathit{InfoP}\ a \to b \to \mathit{InfoP}\ c\\
&\mathit{liftP2} &&:: (a \to b \to c) \to \mathit{InfoP}\ a \to \mathit{InfoP}\ b \to \mathit{InfoP}\ c\\
&\mathit{liftPath} &&:: (\mathit{FilePath} \to a) \to \mathit{InfoP}\ a\\
&(\&\&?) &&:: \mathit{InfoP}\ \mathit{Bool} \to \mathit{InfoP}\ \mathit{Bool} \to \mathit{InfoP}\ \mathit{Bool}\\
&(||?) &&:: \mathit{InfoP}\ \mathit{Bool} \to \mathit{InfoP}\ \mathit{Bool} \to \mathit{InfoP}\ \mathit{Bool}\\
&\mathit{pathP} &&:: \mathit{InfoP}\ \mathit{FilePath}\\
&\mathit{sizeP} &&:: \mathit{InfoP}\ \mathit{Integer}\\
&(==?) &&:: (\mathit{Eq}\ a)\ \ \Rightarrow \mathit{InfoP}\ a \to a \to \mathit{InfoP}\ \mathit{Bool}\\
&(>?) &&:: (\mathit{Ord}\ a) \Rightarrow \mathit{InfoP}\ a \to a \to \mathit{InfoP}\ \mathit{Bool}\\
&\textbf{type } \mathit{Predicate} = \mathit{InfoP}\ \mathit{Bool}\\
&\mathit{find} :: \mathit{FilePath} \to \mathit{Predicate} \to \mathit{IO}\ [\mathit{FilePath}] && \text{-- run function}
\end{aligned}
$$

This is an example of the intended use:

$$
\begin{aligned}
&\mathit{myTest} :: \mathit{InfoP}\ \mathit{Bool}\\
&\mathit{myTest} = (\mathit{liftPath}\ \mathit{takeExtension} ==?\ \texttt{".hs"})\ \&\&?\ (\mathit{sizeP} >?\ 100000)\\
&\mathit{test} :: \mathit{IO}\ [\mathit{FilePath}]\\
&\mathit{test} = \mathit{find}\ \texttt{"/home/patrikj/src"}\ \mathit{myTest}
\end{aligned}
$$

where $\mathit{takeExtension} :: \mathit{FilePath} \to \mathit{String}$ is from $\mathit{System.FilePath}$.

(12 p) **(a)** Implement the constructors and combinators (but not the run function): constP, liftPc, $\mathit{liftP2}$, $\mathit{liftPath}$, $(\&\&?)$, $(||?)$, pathP, sizeP, $(==?)$, $(>?)$.

One implementation of the run function (find) is as follows:

$$
\begin{aligned}
&\mathit{find}\ \mathit{path}\ p = \mathit{getRecursiveContents}\ \mathit{path} \gg\!= \mathit{filterM}\ \mathit{check}\\
&\quad \textbf{where } \mathit{check} :: \mathit{FilePath} \to \mathit{IO}\ \mathit{Bool}\\
&\qquad\ \mathit{check} = \ \{\text{-... some code using } p \text{ ... -}\}\\
&\mathit{getRecursiveContents} :: \mathit{FilePath} \to \mathit{IO}\ [\mathit{FilePath}]\\
&\mathit{getRecursiveContents}\ \mathit{fp} = \ \{\text{-... some code using } \mathit{forM} \text{ ... -}\}
\end{aligned}
$$

(8 p) **(b)** Implement $\mathit{filterM} :: \mathit{Monad}\ m \Rightarrow (a \to m\ \mathit{Bool}) \to [a] \to m\ [a]$ and $\mathit{forM} :: \mathit{Monad}\ m \Rightarrow [a] \to (a \to m\ b) \to m\ [b]$ using just return, $(\gg\!=)$, liftM and foldr. (No **do**-notation, no other library functions.) Provide the type of the first argument to foldr in both cases.

(20 p)

Problem 2: Spec: use specification based development techniques

The *list-fusion* law is as follows:

$$
\begin{aligned}
&\ \ f \circ \mathit{foldr}\ g\ b = \mathit{foldr}\ h\ c\\
&\Leftarrow && \text{-- (is implied by)}\\
&\ \ f\ b = c && \text{-- and}\\
&\ \ f\ (g\ x\ y) = h\ x\ (f\ y) && \text{-- for all } x \text{ and } y
\end{aligned}
$$

(10 p) **(a)** Give the polymorphic types for f, g, b, h, c and prove list-fusion for finite lists by induction.

(10 p) **(b)** For which g and b is $map\ p = foldr\ g\ b$? Use this form and list-fusion to prove that $foldr\ q\ r \circ map\ p$ can be computed as a single $foldr$. Start by giving the polymorphic types of all the one-letter variables.

Problem 3: Types: read, understand and extend Haskell programs which use advanced type system features **(20 p)**

Type classes in Haskell can be "translated away" using a concept called "dictionaries". Each monomorphic instance declaration is translated to a record (called a dictionary) containing the methods of the type class for this particular instance. The following code shows a simplified version of a manual "dictionary translation" for the Eq class and the instances $Eq\ Foo$ and $Eq\ a \Rightarrow Eq\ [a]$.

```haskell
type EqT a = a -> a -> Bool
data EqDict a = EqDict { eq :: EqT a }
  -- An arbitrary type just as an example:
data Foo = Foo String (Maybe String)   -- Personal id number + optional comment
  -- instance Eq Foo
eqFoo :: EqT Foo
eqFoo (Foo pa _ca) (Foo pb _cb) = pa == pb

eqFooD :: EqDict Foo
eqFooD = EqDict eqFoo

  -- instance Eq a => Eq [a]
eqListD :: EqDict a -> EqDict [a]
eqListD (EqDict eqa) = EqDict (eqLa eqa)
```

(a) Implement the function $eqLa :: EqT\ a \rightarrow EqT\ [a]$ which "lifts" equality on an element type a to equality on lists of a. (5 p)

The same technique can be used also for (a simplified version of) the $Monad$ class as follows.

```haskell
 {-# LANGUAGE RankNTypes #-}
type RetT  m = forall a.  a -> m a
type BindT m = forall a b. m a -> (a -> m b) -> m b
data MonadDict m = MonadDict { ret :: RetT m, bind :: BindT m }
  -- instance Monad Maybe
monadMaybeD :: MonadDict Maybe
monadMaybeD = MonadDict returnM bindM

returnM :: RetT Maybe
returnM = Just

bindM :: BindT Maybe
bindM mx f = maybe Nothing f mx
```

For $StateT$ the instance $Monad\ m \Rightarrow Monad\ (StateT\ s\ m)$ gets the following form:

```haskell
monadStateT :: MonadDict m -> MonadDict (StateT s m)
monadStateT (MonadDict retm bindm) = MonadDict (retStateT retm) (bindStateT bindm)
retStateT  :: RetT  m -> RetT  (StateT s m)
bindStateT :: BindT m -> BindT (StateT s m)

newtype StateT s m a = StateT { runStateT :: s -> m (a, s) }
```

(b) Implement $retStateT$ and $bindStateT$. (15 p)

A Library documentation

A.1 Monoids

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

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

$$\varnothing \diamond m == m == m \diamond \varnothing$$
$$(m_1 \diamond m_2) \diamond m_3 == m_1 \diamond (m_2 \diamond m_3)$$

Example: lists form 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
  fail   :: String → m a

class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → 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
  ask :: m e                          -- Get the environment
  local :: (e → e) → m a → m a       -- Change the environment locally
```

Writer monads

```
type WriterT w m a
runWriterT  ::                WriterT w m a → m (a, w)
execWriterT :: (Monad m) ⇒ WriterT w m a → m w

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

State monads

```
type StateT s m a
type State  s   a
runStateT :: StateT s m a → s → m (a, s)
runState  :: State  s   a → s →   (a, s)

class Monad m ⇒ MonadState s m | m → s where
  get :: m s                        -- Get the current state
  put :: s → m ()                   -- Set the current state
  state :: (s → (a, s)) → m a       -- Embed a simple state action into the monad
```

Error monads

```
type ErrorT e m a
runErrorT :: ErrorT e m a -> m (Either e a)

class Monad m => MonadError e m | m -> e where
  throwError :: e -> m a                      -- Throw an error
  catchError :: m a -> (e -> m a) -> m a      -- In catchError x h if x throws an error,
                                              -- it is caught and handled by h.
```

A.3 Some QuickCheck

```
  -- Create Testable properties:
        -- Boolean expressions: (&&), (||), not, ...
(==>) :: Testable p => Bool -> p -> Property
forAll :: (Show a, Testable p) => Gen a -> (a -> p) -> Property
        -- ... and functions returning Testable properties

  -- Run tests:
quickCheck :: Testable prop => prop -> IO ()

  -- Measure the test case distribution:
collect  :: (Show a, Testable p) => a      -> p -> Property
label    :: Testable p =>           String -> p -> Property
classify :: Testable p => Bool -> String -> p -> Property

collect x = label (show x)
label s   = classify True s

  -- Create generators:
choose    :: Random a => (a, a) -> Gen a
elements  :: [a]                -> Gen a
oneof     :: [Gen a]            -> Gen a
frequency :: [(Int, Gen a)]     -> Gen a
sized     :: (Int -> Gen a)     -> Gen a
sequence  :: [Gen a]            -> Gen [a]
vector    :: Arbitrary a => Int -> Gen [a]
arbitrary :: Arbitrary a =>        Gen a
fmap      :: (a -> b) -> Gen a  -> Gen b
instance Monad (Gen a) where ...

  -- Arbitrary — a class for generators
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]
```