Advanced Functional Programming TDA342/DIT260

Patrik Jansson

2011-08-23

Contact:	Patrik Jansson, ext 5415.
Result:	Announced no later than 2011-09-07
Exam check:	Thursday 2011-09-08 and Friday 2011-09-09. Both at 12-13 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: Spec: use specification based development techniques**

(10 p) **(a)** Formulate QuickCheck properties and generators to test the correctness of a sorting function *mysort* :: [*Weekday*] → [*Weekday*].

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun deriving (Eq, Ord, Show, Enum)
```

(10 p) **(b)** The following function implements insertion into a sorted list.

```
insert :: Ord a => a -> [a] -> [a]     -- Line labels (for use in the proof)
insert x []       = [x]                -- ins.0
insert x (y : ys)
  | x <= y        = x : y : ys         -- ins.1a
  | otherwise     = y : insert x ys    -- ins.1b
length :: [a] -> Int
length []       = 0                    -- len.0
length (x : xs) = 1 + length xs        -- len.1
```

Prove the following using list induction: $\forall (x :: a)\ (ys :: [a]).\mathit{length}\ (\mathit{insert}\ x\ ys) == 1 + \mathit{length}\ ys$. Motivate your steps carefully.

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

Calender programs usually support reminders at certain dates so that we can remember weekly knitting lessons, yearly anniversaries or one-off AFP-exams. A reminder can be seen as a set of dates (*DateSet*) and a function *Date* → *Message*. This problem is only about the domain *DateSet* specifying sets of calendar dates. The type *DateSet* should support expressing

- singleton *DateSet* (contains one particular date),
- (unbounded) infinitely repeating daily, weekly, monthly and yearly *Datesets*,
- the *DateSet* of all days between a start and an end date (inclusive),
- union and intersection of *DateSets*.

An example (here in pseudo-code) of one *DateSet* value could be the intersection of the infinite repeat "every Monday" with the *DateSet* "between 2011-08-23 and 2011-12-20".

(13 p) **(a)** Implement an EDSL in Haskell for the domain of *DateSets* as described above. Implement

- a type *DateSet*,
- a function *isIn* :: *Date* → *DateSet* → *Bool*
- a function *upperBound* :: *DateSet* → *Date* which gives a good (but not necessarily least) upper bound,
- and a function *toList* :: *DateSet* → [*Date*] returning all dates in the set in increasing order. You can start from *epochD* = *readD* `"1970-01-01"` and end at (or before) the *upperBound*.

You don't need to define any *String* encoding or decoding (you can assume functions *readD* :: *String* → *Date* and *showD* :: *Date* → *String*). You may also use accessor functions *weekday*, *monthday*, *yearday* which all take a *Date* and return an *Int*, and a function *nextD* :: *Date* → *Date* to move to the next day.

(7 p) **(b)** Explain briefly the following EDSL terminology in general: deep embedding, shallow embedding, constructors, combinators and run function. Exemplify by referring to or contrasting with your implementation.

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

(Code borrowed from RWH, chapter 18.)

```
 {-# LANGUAGE FlexibleContexts #-}
import Control.Monad.Writer

newtype MaybeT m a = MaybeT {runMaybeT :: m (Maybe a)}

bindMT :: (Monad m) ⇒ MaybeT m a → (a → MaybeT m b) → MaybeT m b
x ‘bindMT‘ f = MaybeT $ runMaybeT x ≫= maybe (return Nothing) (runMaybeT ∘ f)

returnMT :: (Monad m) ⇒ a → MaybeT m a
returnMT a = MaybeT $ return (Just a)

failMT :: (Monad m) ⇒ t → MaybeT m a
failMT _ = MaybeT $ return Nothing

instance (Monad m) ⇒ Monad (MaybeT m) where
  return = returnMT
  (≫=)  = bindMT
  fail   = failMT

problem :: MonadWriter [String] m ⇒ m Int
problem = do
  tell ["I fail"]
  fail "oops"
  return 1738

type A = WriterT [String] Maybe
type B = MaybeT (Writer [String])

a :: A Int
a = problem

b :: B Int
b = problem
```

(a) What do *runWriterT a* and *runWriter* (*runMaybeT b*) evaluate to? Explain. (7 p)

(b) Implement *tellMT* and *listenMT* and give their type signatures. (13 p)

```
instance (Monoid w, MonadWriter w m) ⇒ MonadWriter w (MaybeT m) where
  tell   = tellMT
  listen = listenMT
```

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 0 for *mempty* and (+) for *mappend*):

$$\begin{aligned}&0 + m == m\\ &m + 0 == m\\ &(m_1 + m_2) + m_3 == m_1 + (m_2 + m_3)\end{aligned}$$

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 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

```
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

```
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

```
type StateT s m a
runStateT :: StateT s m a → s → m (a, s)
class Monad m ⇒ MonadState s m | m → s where
    -- Get the current state
  get :: m s
    -- Set the current state
  put :: s → m ()
```

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
    -- Throw an error
  throwError :: e → m a
    -- If the first computation throws an error, it is
    -- caught and given to the second argument.
  catchError :: m a → (e → m a) → m a
```

A.3 Some QuickCheck

```
  -- Create Testable properties:
        -- Boolean expressions: (∧), (|), ¬, ...
(==>) :: 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]
```