Advanced Functional Programming TDA342/DIT260

Patrik Jansson

2013-08-26

Contact:	Patrik Jansson, ext 5415.
Result:	Announced no later than 2013-09-13
Exam check:	Th 2013-09-19 and Fr 2013-09-20. 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: implement embedded domain specific languages

Part of the QuickCheck library is a domain specific language for expressing *Gen*erators of pseudo-random values. Your task is to implement a simplified version of this DSL. You *don't* need to handle *sized* generators, infinite values, exception handling or the *Property* part of QuickCheck. You only need to implement these operations:

```
elements  :: [a]                     -> Gen a
oneof     :: [Gen a]                 -> Gen a
frequency :: [(Int, Gen a)]          -> Gen a
sequence  :: [Gen a]                 -> Gen [a]
returnGen :: a                       -> Gen a
bindGen   :: Gen a -> (a -> Gen b)   -> Gen b
fmapGen   :: (a -> b) -> Gen a       -> Gen b
  -- A call to run g n r gives n (pseudo-)random values from the
  -- generator g using the (pseudo-)random source r.
run       :: Gen a -> Int -> StdGen -> [a]
instance Functor (Gen a) where fmap   = fmapGen
instance Monad   (Gen a) where return = returnGen; (>>=) = bindGen
```

(10 p) **(a)** The operations *elements*, *oneof*, *sequence* and *fmapGen* are *derived* operations, that is, they can be implemented in terms of the other operations without knowing the implementation of *Gen*. Show how. (You must treat *Gen* as an abstract type for this subproblem.)

(10 p) **(b)** Implement *Gen*, the remaining operations (*frequency*, *returnGen*, *bindGen*) and *run*.

You can assume this is in scope:

```
import Control.Monad.State as CMS
import System.Random (StdGen, next)
type Sem a = CMS.State StdGen a
  -- nextBoundedBy bound for 0 < bound returns a random result 0 ⩽ result < bound
nextBoundedBy :: Int -> Sem Int
```

or, alternatively, you could build your solution around

```
randomR :: Random a => (a, a) -> StdGen -> (a, StdGen)
```

which takes "a range (lo, hi) and a random number generator r, and returns a random value uniformly distributed in the closed interval $[lo, hi]$, together with a new generator." (Quote from *System.Random*.)

(20 p) Problem 2: Types: read, understand and extend Haskell programs which use advanced type system features

```
  -- file: RealWorldHaskell/ch18/CountEntriesT.hs
module CountEntriesT (listDirectory, countEntries) where
import System.Directory (doesDirectoryExist, getDirectoryContents)
import System.FilePath ((</>))
import Control.Monad (forM_, when, liftM)
import Control.Monad.Trans (liftIO)
import Control.Monad.Writer (WriterT, tell, execWriterT)
listDirectory :: FilePath -> IO [String]
listDirectory = liftM (filter notDots) ∘ getDirectoryContents
    where notDots p = p ≠ "." ∧ p ≠ ".."
```

```haskell
countEntries :: FilePath -> WriterT [(FilePath, Int)] IO ()
countEntries path = do
  contents <- liftIO . listDirectory $ path
  tell [(path, length contents)]
  forM_ contents $ \name -> do
    let newName = path </> name
    isDir <- liftIO . doesDirectoryExist $ newName
    when isDir $ countEntries newName
```

(a) A directory `T` has two subdirectories `A` and `D` which in turn contain files called `B`, `C` (in `A`) and `E` (in `D`). What is printed when *execWriterT* (*countEntries* `"T"`) $\gg=$ *print* is run? Explain! (10 p)

(b) Implement a variant so that *countEntriesMax n fp* recurses no deeper than n levels. (10 p)

Problem 3: Spec: use specification based development techniques (20 p)

This is the parser DSL interface from lecture 4:

```haskell
symbol :: P s s
pfail  :: P s a
(+++)  :: P s a -> P s a -> P s a
return :: a -> P s a
(>>=)  :: P s a -> (a -> P s b) -> P s b
```

The semantics of a parser of type $P\ s\ a$ is a function from a string of s to a multiset of results paired with the remaining parts of the input string. We use a multiset to capture the fact that we don't care about the order of the results.

The semantic function *sem* is specified as follows (we use list notation to denote multisets and (\/) for multiset union).

```haskell
sem :: P s a -> [s] -> [(a, [s])]
sem symbol     (s : ss) = [(s, ss)]                      -- sem.sym.1
sem symbol     []       = []                             -- sem.sym.2
sem pfail      ss       = []                             -- sem.pfail
sem (p +++ q)  ss       = sem p ss \/ sem q ss           -- sem.+++
sem (return x) ss       = [(x, ss)]                      -- sem.ret
sem (p >>= f)  ss       = [(y, ss'') | (x, ss')  <- sem p ss
                                     , (y, ss'') <- sem (f x) ss'   -- sem.bind
                          ]
```

Using this semantics it is possible to prove a number of useful laws about parsers and the laws can be used to derive an efficient implementation of the library. For two parsers p and q we define

$p == q$ iff $\forall ss.\ sem\ p\ ss == sem\ q\ ss$

Some parsing laws and lemmas:

```haskell
L5 :     (p +++ q) >>= f == (p >>= f) +++ (q >>= f)
L10 :    (symbol >>= f) +++ (symbol >>= g) == symbol >>= (\s -> f s +++ g s)
Lemma1 : sem (symbol >>= f) (s : ss) = sem (f s) ss
Lemma2 : sem (symbol >>= f) []       = []
```

You may use the lemmas and the specification of *sem* in your answers.

(a) What is the value of *sem* (*symbol* $\gg$ *symbol*) `"(h)"`? Prove it by equational reasoning. (10 p)

(b) Prove L10 with equational reasoning for the empty and the non-empty list case. (10 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 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:

```
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)
execWriterT :: (Monad m) => WriterT w m a -> m 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
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 the current state
  get :: m s
    -- Set the current state
  put :: s → m ()
    -- Embed a simple state action into the monad
  state :: (s → (a, s)) → m a
```

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]
```