Advanced Functional Programming TDA342/DIT260

Patrik Jansson

2011-03-16

Contact: Patrik Jansson, ext 5415.

Result: Announced no later than 2011-03-31

Exam check: Friday 2011-04-01 and Monday 2011-04-04. 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: Difference lists

A library for "Difference lists" has the following API (from Real World Haskell, Chapter 13):

```
newtype DList a = DL { unDL :: [a] -> [a] }
append   :: DList a -> DList a -> DList a
cons     :: a -> DList a -> DList a
empty    :: DList a
foldr    :: (a -> b -> b) -> b -> DList a -> b
fromList :: [a] -> DList a
map      :: (a -> b) -> DList a -> DList b
toList   :: DList a -> [a]
instance Functor DList where ...
instance Monoid (DList a) where ...
instance Eq a => Eq (DList a) where ...
```

(10 p) **(a)** Implement the API: *append*, *cons*, *empty*, *foldr*, *fromList*, *map*, *toList* + the *Functor*, *Monoid* and *Eq* instances.

(10 p) **(b)** State (including type signatures) the two *Functor* laws + the first and third *Monoid* laws for *DList* in the form of polymorphic QuickCheck properties, provide a *main* function which calls *quickCheck* on these properties and implement the *Arbitrary* (*DList a*) instance.

(20 p) Problem 2: Parallelism and Concurrency

(5 p) **(a)** Give a short explanation of the denotational and operational semantics of *pseq* and *par*.

(10 p) **(b)** Implement a function *transfer* (with type given below) which attempts to transfer some gold from the first balance to the other. It should use the *STM* monad to retry if the first balance would become negative. (The API for Software Transactional Memory (STM) is provided in Appendix A.)

```
newtype Gold = Gold Int
type Balance = TVar Gold
transfer :: Gold -> Balance -> Balance -> IO ()
```

(5 p) **(c)** Assume that in a particular run of the *transfer* function the first balance is too small so that the system should retry. Before retrying, the STM implementation will wait until one of the variables used in the atomic block is changed.

Why is that?

If the change of the first balance is a decrease, will the atomic block be retried?

Problem 3: An embedded language **(20 p)**

Here is a GADT representing an embedded DSL for simple (regexp-like) parsers and a partial implementation of a run function: *parse*. (Code borrowed from `http://www.haskell.org/haskellwiki/Generalised_algebraic_datatype`.)

```
data Parser tok a where
  Zero   ::                                          Parser tok ()
  One    ::                                          Parser tok ()
  Check  :: (tok -> Bool) ->                         Parser tok tok
  Satisfy :: ([tok] -> Bool) ->                      Parser tok [tok]
  Push   :: tok -> Parser tok a ->                   Parser tok a
  Plus   :: Parser tok a -> Parser tok b ->          Parser tok (Either a b)
  Times  :: Parser tok a -> Parser tok b ->          Parser tok (a, b)
  Star   :: Parser tok a ->                          Parser tok [a]

parse :: MonadPlus m => Parser tok a -> [tok] -> m a

  -- Zero always fails.
parse Zero _ = mzero

  -- One matches only the empty string.
parse One [] = return ()
parse One _ = mzero

  -- Check p matches a string with exactly one token t such that p t holds.
parse (Check p) [t] | p t = return t
parse (Check p) _         = mzero

  -- Satisfy p matches any string such that p ts holds.
parse (Satisfy p) xs = if p xs then return xs else mzero

  -- Push t p matches a string ts when p matches (t : ts).
parse (Push t p) ts = parse p (t : ts)

  -- Plus p q matches when either p or q does.
parse (Plus p q) ts = liftM Left  (parse p ts) `mplus`
                      liftM Right (parse q ts)

  -- Times p q matches the concatenation of p and q.
  -- Star p matches zero or more copies of p.
```

(a) Complete the definition of *parse* by filling in cases for *Times p q* and *Star p*. (10 p)

(b) Implement parts of a shallow embedding for the same EDSL: a type *P m tok a*, constructor functions *zero*, *one*, *check*, *satisfy*, *plus* and a run function *runP*. (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*):

$$0 + m \equiv m$$
$$m + 0 \equiv m$$
$$(m_1 + m_2) + m_3 \equiv m_1 + (m_2 + m_3)$$

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)
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]
```

A.4 STM

```
type STM a
instance Monad STM

  -- Run an STM computation. Behaves as if the entire
  -- computation is performed in one atomic step. If
  -- the computation is aborted (for instance, using retry),
  -- it will be reexecuted until it succeeds.
atomically :: STM a → IO a

  -- Abort a computation.
retry :: STM a

  -- If the first argument is aborted (using retry), the
  -- second argument will be executed. If that one also
  -- aborts the entire computation will be aborted.
orElse :: STM a → STM a → STM a

  -- Transaction variables.
type TVar a

newTVar :: a → STM (TVar a)
readTVar :: TVar a → STM a
writeTVar :: TVar a → a → STM ()

instance MonadPlus STM where
  mzero = retry
  mplus = orElse
```