Advanced Functional Programming TDA342/DIT260

Patrik Jansson

2012-08-28

Contact: Patrik Jansson, ext 5415.

Result: Announced no later than 2012-09-16

Exam check: Th 2012-09-06 and Fr 2012-09-07. 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

A DSL for two-dimensional geometrical shapes has the following interface (from AFP lecture 2):

```
empty :: Shape
disc  :: Shape   -- disc with radius 1 around the origin
square :: Shape  -- square between (0,0) and (1,1)

translate  :: Vec   → Shape → Shape   -- shift the shape along a vector
scale      :: Vec   → Shape → Shape   -- magnify the shape by a vector
rotate     :: Angle → Shape → Shape   -- rotate the shape by an angle (around the origin)
union      :: Shape → Shape → Shape
intersect  :: Shape → Shape → Shape
difference :: Shape → Shape → Shape

inside     :: Point → Shape → Bool    -- run function: is the point inside the shape?
```

In your implementations you can assume this import is in scope:

```
import Matrix ( Vec, vecX, vecY   -- ::Vec → Double
              , Angle             -- = Double
              , Point             -- = Vec
              , sub, divide       -- ::Point → Vec → Point
              , rot               -- ::Angle → Point → Point
              )
```

Your task is to implement the following parts of this API for a deep and a for a shallow embedding.

(10 p) **(a) Deep:** implement *Shape*, *empty*, *disc*, *square*, *translate*, *union*, *intersect* and *inside*.

(10 p) **(b) Shallow:** implement *Shape*, *disc*, *scale*, *rotate*, *intersect*, *difference* and *inside*.

(20 p) Problem 2: Spec: use specification based development techniques

Consider the three (partially applied) type constructors *Either* e, $((,)\ e)$ and $((\to)\ e)$. Your implementation should be polymorphic in the given type e.

(10 p) **(a)** Provide a *Functor* instance (thus implement *fmap*) for each of them. Explain which of these are monads. Do you recognize any of these from Haskell?

(10 p) **(b)** State the functor laws and prove by equational reasoning that they hold for these types.

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

The *ListT* monad transformer adds "backtracking" to a given monad (if it is commutative).

```
newtype ListT m a = ListT {runListT :: m [a]}
instance (Monad m) ⇒ Monad (ListT m) where
  return = returnLT
  (>>=)  = bindLT
```

(10 p) **(a)** Provide type signatures for, and implement, *returnLT* and *bindLT*.

(10 p) **(b)** A *commutative monad* is any monad where we can replace the expression:

```
do a1 ← m1
   a2 ← m2
   f a1 a2
```

with

```
do a2 ← m2
   a1 ← m1
   f a1 a2
```

without changing the meaning.

Write a polymorphic QuickCheck property *commutative* which can test if a monad is commutative. Specialise your property to monomorphic types and to one of the functors from Problem 2 and implement a generator. Is that functor a commutative monad?

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)
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]
```