Advanced Functional Programming TDA342/DIT260

Anton Ekblad and Patrik Jansson

2015-08-24

Contact:	Anton Ekblad (070 7579 070), Patrik Jansson (031-7725415).
Result:	Announced no later than 2015-09-09
Exam check:	Th 2015-09-10 and Fr 2015-09-11. 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

In the TV show *Robot Wars*, robots fight each other to death using various close-range weapons. For this task, you are going to design and implement a DSL for expressing strategies for those robots. For simplicity, we assume that robot battles are fought on a flat surface, free of obstacles.

An example of such a strategy:

```
while enemies alive do
    if robot in line of sight then
        if right in front of robot then
            Attack!
            Move backward by 0.5 metres.
        else
            Move forward by 0.1 metres.
        end if
    else
        Turn left by 2 degrees.
    end if
end while
```

(5 p) **(a)** Design an API for expressing Robot Wars strategies such as the one above. Your API should consist of suitable names of types, as well as names and types of functions used to construct and combine strategies. For each function, give a short description of its purpose. Ensure that your DSL is powerful enough to express the above strategy.

Use your API to implement the above strategy.

(3 p) **(b)** State which of the operations in your API are derived and which are primitive. Give definitions of the derived operations in terms of the primitive ones. Try to minimize the number of primitive operations.

(3 p) **(c)** State three laws about how the functions of your API relate to each other.

(3 p) **(d)** Give a deep embedding of the primitive operations in your API. Give a type definition and describe what each of the type's constructors are supposed to do.

(6 p) **(e)** You have been given the following low level robot control library:

```
-- Gives the distance, in metres, to the nearest object in the robot's line of sight.
-- Returns Nothing if no object is spotted.
opticalSensor :: IO (Maybe Double)

-- Wait the specified number of seconds before resuming execution.
wait :: Double → IO ()

-- Instantly accelerate robot to x metres per second; negative values moves backward.
setSpeed :: Double → IO ()

-- Turn left by the given number of degrees.
turn :: Double → IO ()

-- Gives the total number of live robots left in the arena.
liveRobots :: IO Int

-- Attack whatever is in front of the robot.
attack :: IO ()
```

Use the above library to implement a "run" function for your deep embedding, and give the type of your run function.

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

Stringly typed programming is a practice where strings are used to hold non-string data - numbers or enumeration values, for instance - and functions operate over these strings rather than over more strictly defined data types. This practice is common in shell scripts but, thankfully, not in the functional programming world.

For this task, you are going to implement code to convert functions over an arbitrary number of *Int*s to stringly typed functions, and back again. That is, a function *string* which converts an n-ary function $f :: Int \to \ldots \to Int$ into an n-ary function $f' :: String \to \ldots \to String$, and a function *unstring* which is the inverse of *string*.

The following code gives a type class *Stringly* which uses associated types to accomplish this:

```
class Stringly f where
  type Str f
  string :: f → Str f
  unstring :: Str f → f
```

Additionally, here are some examples and invariants to explain how the type class is used:

```
add :: Int → Int → Int
add = (+)

stringly_add :: String → String → String
stringly_add = string add

unstringly_add :: Int → Int → Int
unstringly_add = unstring stringly_add

prop_inverse :: (Int → Int) → Int → Bool
prop_inverse = λf x → f x == unstring (string f) x

prop_preserves_semantics :: (Int → Int) → Int → Bool
prop_preserves_semantics = λf x → show (f x) == string f (show x)
```

(a) Using associated types, give the *Stringly* instances required to implement *string* and *unstring* as specified above. (10 p)

Hint: you should need at most two instances.

(b) In many cases, multiparameter type classes can be used to solve the same problems as associated types. The following code gives a type class to provide the same functionality as *Stringly*, this time using multiparameter type classes: (10 p)

```
class Stringly2 f s where
  string2 :: f → s
  unstring2 :: s → f
```

Using multiparameter type classes, give the *Stringly2* instances required to implement *string2* and *unstring2* as specified above.

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

A binary search tree is a binary tree with the following invariant: the value at the root of the tree is *larger than or equal to* all values in the *left* subtree, and *smaller than or equal to* all values in the *right* subtree.

Below is a data type definition for such a tree.

data *Tree a* = *Nil* | *Tree a* (*Tree a*) (*Tree a*)

(5 p) **(a)** Write a QuickCheck property which expresses the invariant given above for the *Tree* type.

Hint: it is easier to check if a list is ordered than to check if a tree is...

(7 p) **(b)** Write a sized QuickCheck generator (*sizedTree* :: *Ord a* ⇒ *Gen a* → *Gen* (*Tree a*)) for the *Tree* type. Generated trees must preserve the invariant given above.

Do *not* just generate arbitrary trees and filter out the ones that don't satisfy the invariant. If your generator uses *suchThat*, you are on the wrong track.

Hint: one possible solution involves generating lists of elements, sorted or otherwise...

(3 p) **(c)** Make *Tree* an instance of QuickCheck's *Arbitrary* type class and write a *main* function which uses QuickCheck to test that your generator preserves the invariant for trees of integers.

(5 p) **(d)** Since the *Tree* type is a functor, we add an appropriate *Functor* instance for it.

```
instance Functor Tree where
  fmap _ Nil = Nil
  fmap f (Tree x l r) = Tree (f x) (fmap f l) (fmap f r)
```

Recall that all functors must obey the two *functor laws*:

$$fmap\ id = id$$

$$fmap\ (f \circ g) = fmap\ f \circ fmap\ g$$

Using equational reasoning, show that the functor laws hold for the above *Functor* instance.

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*):

```
∅ ⋄ m == m == m ⋄ ∅
(m₁ ⋄ m₂) ⋄ m₃ == m₁ ⋄ (m₂ ⋄ m₃)
```

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]
```