Advanced Functional Programming TDA342/DIT260

Patrik Jansson

2013-03-16

Contact:	Patrik Jansson, ext 5415. Will answer questions after 1 and after 3 hours.
Result:	Announced no later than 2013-04-05
Exam check:	Mo 2013-04-08 and We 2013-04-10. 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, max: 60p PhD student: 36p to pass, max: 60p
Remember:	Read the full exam before starting (perhaps the easy stuff is near the end). Hand in the summary sheet (if you brought one) with the exam solutions. Start each problem on a new sheet of paper. Don’t write on the back of the paper. Write legibly.

(18 p)

Problem 1: Spec: use specification based development techniques

(7 p) **(a)** Imagine you should test an implementation of a function $sort :: Ord\ a \Rightarrow [a] \to [a]$. Implement a QuickCheck property which checks that the result is ordered and a permutation of the input.

(5 p) **(b)** Explain what "pure" (referentially transparent) means in a functional programming context and how it relates to equational reasoning.

(6 p) **(c)** Even though list concatenation is associative, that is $lhs == (as +\!\!+ bs) +\!\!+ cs == as +\!\!+ (bs +\!\!+ cs) == rhs$, it may still be good for performance to transform lhs to rhs. Explain why by expanding $head\ lhs$ and $head\ rhs$. You may assume that only case distinctions (pattern matching) takes time and that as contains at least one element.

```
(++) :: [a] -> [a] -> [a]
xs ++ ys = case xs of
  []        -> ys                -- ++.1
  (x : xs') -> x : (xs' ++ ys)   -- ++.2
```

(22 p)

Problem 2: DSL: design embedded domain specific languages

A DSL for symbolic algebra. The *Num*, *Fractional* and *Floating* classes in Haskell provide an API for several mathematical operations and the standard library provides instances for several base types like integers, floating point numbers and rationals. Your task here is to implement a DSL for symbolic expressions for the following subset of this API:

```
class (Eq a, Show a) => Num a where
  (+), (*)    :: a -> a -> a
  negate      :: a -> a
  fromInteger :: Integer -> a
class (Num a) => Fractional a  where
  (/)         :: a -> a -> a
class (Fractional a) => Floating a where
  pi          :: a
  exp, log    :: a -> a
  sin, cos    :: a -> a
```

(5 p) **(a)** Implement a type $Sym\ v$ as a deep embedding of the API & symbolic variables of type v.

(5 p) **(b)** Implement parts of a run function $eval :: Floating\ n \Rightarrow (v \to Maybe\ n) \to Sym\ v \to Maybe\ n$. It is enough to implement the cases for variables, $(+)$, $negate$, $fromInteger$, $(/)$ and exp.

(7 p) **(c)** Implement an algebraic simplification function $simp :: Sym\ v \to Maybe\ (Sym\ v)$ which applies the rules $0 * e == 0$, $sin\ pi == 0$, $log\ (exp\ e) == e$ bottom-up and which fails (with *Nothing*) on division by zero.

(5 p) **(d)** Is there a reasonable *Monad* instance for *Sym*? If so, implement *return* and sketch $(\gg\!=)$, otherwise explain why not.

Problem 3: Types: read, understand and extend Haskell programs (20 p)

A *generalised trie* for a type k is a parametrised datatype used to store a lookup table representing a partial function from k to some value type. (The term "partial function" here means "a function returning *Maybe a*".) The following code (from the Haskell wiki page on type families) implements generalised tries for finite types built from units, sums and pairs.

```haskell
class GMapKey k where
  data GMap k :: * -> *
  empty :: GMap k v
  lookup :: k -> GMap k v -> Maybe v
  insert :: k -> v -> GMap k v -> GMap k v
instance GMapKey () where
  data GMap () v       = GMU (Maybe v)
  empty                = GMU Nothing
  lookup () (GMU mv)   = mv
  insert () v (GMU _)  = GMU $ Just v
instance (GMapKey a, GMapKey b) => GMapKey (Either a b) where
  data GMap (Either a b) v          = GME (GMap a v) (GMap b v)
  empty                             = GME empty empty
  lookup (Left a)  (GME gm1 _gm2)   = lookup a gm1
  lookup (Right b) (GME _gm1 gm2)   = lookup b gm2
  insert (Left a)  v (GME gm1 gm2)  = GME (insert a v gm1) gm2
  insert (Right a) v (GME gm1 gm2)  = GME gm1 (insert a v gm2)
instance (GMapKey a, GMapKey b) => GMapKey (a, b) where
  data GMap (a, b) v       = GMP (GMap a (GMap b v))
  empty                    = GMP empty
  lookup (a, b) (GMP gm)   = lookupGMP a b gm           -- TODO
  insert (a, b) v (GMP gm) = GMP (insertGMP a b v gm)   -- TODO
```

Some examples to get a feeling for how it works:

```haskell
type Bit = Either () ()
o = Left (); i = Right ()
type Four = (Bit, Bit)
oo = (o, o); oi = (o, i); io = (i, o); ii = (i, i)
t0, t1, t2 :: GMap Four Int  -- ≃ (Maybe (Maybe Int, Maybe Int), Maybe (Maybe Int, Maybe Int))
t0 = empty                   -- ≃ (Nothing, Nothing)
t1 = insert oi 17 t0         -- ≃ (Just (Nothing, Just 17), Nothing)
t2 = insert io 38 t1         -- ≃ (Just (Nothing, Just 17), Just (Just 38, Nothing))
```

(a) Fully expand the type family application *GMap* (*Either* () (*Bit*, *a*)) *v*. You may ignore the constructors *GMU*, *GME* and *GMP* as I did in the comment after type signature for *t0*. (4 p)

(b) Give the type signatures for and implement *lookupGMP* and *insertGMP*. (6 p)

(c) Here are the *Functor* instances for *GMap* (), *GMap* (*Either a b*) and *GMap* (*a*, *b*): (10 p)

```haskell
instance Functor (GMap ()) where
  fmap = fmapGMU  -- TODO
instance (Functor (GMap a), Functor (GMap b)) => Functor (GMap (Either a b)) where
  fmap = fmapGME  -- TODO
instance (Functor (GMap a), Functor (GMap b)) => Functor (GMap (a, b)) where
  fmap = fmapGMP  -- TODO
```

Give type signatures for and implement *fmapGMU*, *fmapGME* and *fmapGMP*.

A Library documentation

A.1 Monoids

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monoid}\ a\ \textbf{where} \\
\quad \mathit{mempty} :: a \\
\quad \mathit{mappend} :: a \to a \to a
\end{array}
$$

Monoid laws (variables are implicitly quantified, and we write 0 for *mempty* and (+) for *mappend*):

$$
\begin{array}{l}
0 + m == m \\
m + 0 == m \\
(m_1 + m_2) + m_3 == m_1 + (m_2 + m_3)
\end{array}
$$

Example: lists form a monoid:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monoid}\ [a]\ \textbf{where} \\
\quad \mathit{mempty} \qquad\quad = [\,] \\
\quad \mathit{mappend}\ \mathit{xs}\ \mathit{ys} = \mathit{xs} +\!\!+\ \mathit{ys}
\end{array}
$$

A.2 Monads and monad transformers

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{return} :: a \to m\ a \\
\quad (\gg\!=)\ \ :: m\ a \to (a \to m\ b) \to m\ b \\
\quad \mathit{fail}\ \ \ \ :: \mathit{String} \to m\ a \\
\textbf{class}\ \mathit{MonadTrans}\ t\ \textbf{where} \\
\quad \mathit{lift} :: \mathit{Monad}\ m \Rightarrow m\ a \to t\ m\ a \\
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadPlus}\ m\ \textbf{where} \\
\quad \mathit{mzero} :: m\ a \\
\quad \mathit{mplus} :: m\ a \to m\ a \to m\ a
\end{array}
$$

Reader monads

$$
\begin{array}{l}
\textbf{type}\ \mathit{ReaderT}\ e\ m\ a \\
\mathit{runReaderT} :: \mathit{ReaderT}\ e\ m\ a \to e \to m\ a \\
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadReader}\ e\ m \mid m \to e\ \textbf{where} \\
\qquad \text{-- Get the environment} \\
\quad \mathit{ask} :: m\ e \\
\qquad \text{-- Change the environment locally} \\
\quad \mathit{local} :: (e \to e) \to m\ a \to m\ a
\end{array}
$$

Writer monads

$$
\begin{array}{l}
\textbf{type}\ \mathit{WriterT}\ w\ m\ a \\
\mathit{runWriterT} :: \mathit{WriterT}\ w\ m\ a \to m\ (a, w) \\
\textbf{class}\ (\mathit{Monad}\ m, \mathit{Monoid}\ w) \Rightarrow \mathit{MonadWriter}\ w\ m \mid m \to w\ \textbf{where} \\
\qquad \text{-- Output something} \\
\quad \mathit{tell} :: w \to m\ () \\
\qquad \text{-- Listen to the outputs of a computation.} \\
\quad \mathit{listen} :: m\ a \to m\ (a, w)
\end{array}
$$

State monads

$$\textbf{type}\ \mathit{StateT}\ s\ m\ a$$
$$\mathit{runStateT} :: \mathit{StateT}\ s\ m\ a \to s \to m\ (a, s)$$

$$\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \to s\ \textbf{where}$$

```
    -- Get the current state
  get :: m s
    -- Set the current state
  put :: s → m ()
```

Error monads

$$\textbf{type}\ \mathit{ErrorT}\ e\ m\ a$$
$$\mathit{runErrorT} :: \mathit{ErrorT}\ e\ m\ a \to m\ (\mathit{Either}\ e\ a)$$

$$\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadError}\ e\ m \mid m \to e\ \textbf{where}$$

```
    -- 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]
```