Advanced Functional Programming TDA342/DIT260

Patrik Jansson, Jonas Duregård

2014-08-25

Contact: Patrik Jansson, ext 5415.

Result: Announced no later than 2014-09-12

Exam check: Mo 2014-09-15 and Tu 2014-09-16. Both at 12.45-13.00 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.

(25 p)

Problem 1: DSL: design an embedded domain specific language

This assignment is about design and implementation of an embedded language for “ASCII art”. The language should be compositional, that is, enable building complex images by combining simpler images. Here is one example of what the language should be able to express (but you need not implement the rendering).

```
+-+   +--------+
|1|   |+---+   |
|7|   ||hi!|   |
|3|   |+---+   |
|8|   |        |
+-+   |+------+|
      ||Patrik||
      |+------+|
      +--------+
```

Typical components are: horizontal text, vertical text, framed boxes, relative placement (above, beside, etc.).

(5 p) **(a)** Design an API for the above embedded language. This should consist of: suitable names of types, names and type of operations for constructing, combining and “running”. For each operation, describe briefly what it is supposed to do. Keep the role of each operation as simple as possible but add enough combinators to allow describing the picture above. (Note that this part does not ask for any implementation code, only names and type signatures.)

Implement the example above in terms of your API.

(5 p) **(b)** Which of the operations in your API are primitive and which are derived? Give definitions of the derived operations in terms of the primitive operations.

(5 p) **(c)** What properties (or laws) do your functions have? Mention at least three non-trivial such ways in which your functions interact.

(5 p) **(d)** Describe what a *shallow* implementation could look like. Give a type definition and describe (in words or code) what each of your primitive operations, and your run function, would do.

(5 p) **(e)** Describe what a *deep* implementation could look like. Give a type definition and describe (in words or code) what each of your primitive operations, and your run function, would do.

Problem 2: Spec: use specification based development techniques **(15 p)**

Below is an attempt at a QuickCheck test suite for $qsort :: Ord\ a \Rightarrow [a] \rightarrow [a]$.

$$
\begin{array}{l}
prop\_minimum\ xs = head\ (qsort\ xs) == minimum\ xs \\
prop\_ordered\ xs = ordered\ (qsort\ xs) \\
\quad \mathbf{where}\ ordered\ [\,] \qquad\qquad = True \\
\qquad\quad ordered\ (x : y : xs) = x \leqslant y\ \&\&\ ordered\ (y : xs) \\
prop\_permutation\ xs = permutation\ xs\ (qsort\ xs) \\
\quad \mathbf{where}\ permutation\ xs\ ys = null\ (xs \setminus\setminus ys)\ \&\&\ null\ (ys \setminus\setminus ys)
\end{array}
$$

(a) Find and correct at least one bug per property in the test suite. (5 p)

(b) Write a *main* function which tests the three properties (for lists of integers) using QuickCheck. (5 p)

(c) Write a sized generator ($sizedList :: Gen\ a \rightarrow Gen\ [a]$) for random lists. Make sure the list length is random (but bounded by the current size). (5 p)

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

(a) Define a GADT (Generalised Algebraic DataType) $Expr\ t$ representing the well-typed terms of a simple expression language with character and integer literals, function application and the built-in operations $Nil :: Expr\ [a]$, $Cons :: Expr\ (a \rightarrow [a] \rightarrow [a])$, $Length :: Expr\ ([a] \rightarrow Int)$, $Replicate :: Expr\ (Int \rightarrow a \rightarrow [a])$. Note that there are no variables in the language. (5 p)

(b) Define a function $eval :: Expr\ t \rightarrow t$ to compute the value of an expression. (5 p)

(c) Implement the derived operation $stringLit :: String \rightarrow Expr\ String$ (with no change to *Expr*). (4 p)

(d) Extend the language with numbered variables representing strings and call the new language *Expr2* with evaluator (6 p)

$$
\begin{array}{l}
eval2 :: (Name \rightarrow Maybe\ String) \rightarrow Expr\ t \rightarrow Maybe\ t \\
\mathbf{type}\ Name = Int
\end{array}
$$

What is changed in *Expr2* compared to *Expr* and in the function *eval2* compared to *eval*?

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*):

$$\varnothing \diamond m == m == m \diamond \varnothing$$
$$(m_1 \diamond m_2) \diamond m_3 == m_1 \diamond (m_2 \diamond m_3)$$

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 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

$$\begin{array}{l}
\textbf{type}\ \mathit{ErrorT}\ e\ m\ a \\
\mathit{runErrorT} :: \mathit{ErrorT}\ e\ m\ a \to m\ (\mathit{Either}\ e\ a) \\
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadError}\ e\ m \mid m \to e\ \textbf{where} \\
\quad \mathit{throwError} :: e \to m\ a \qquad\qquad\qquad \text{-- Throw an error} \\
\quad \mathit{catchError} :: m\ a \to (e \to m\ a) \to m\ a \quad \text{-- In } \mathit{catchError}\ x\ h \text{ if } x \text{ throws an error,} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{-- it is caught and handled by } h.
\end{array}$$

A.3 Some QuickCheck

$$\begin{array}{l}
\quad \text{-- Create Testable properties:} \\
\qquad\qquad \text{-- Boolean expressions: } (\&\&), (\|), \mathit{not}, \ldots \\
(==>) :: \mathit{Testable}\ p \Rightarrow \mathit{Bool} \to p \to \mathit{Property} \\
\mathit{forAll}\ :: (\mathit{Show}\ a, \mathit{Testable}\ p) \Rightarrow \mathit{Gen}\ a \to (a \to p) \to \mathit{Property} \\
\qquad\qquad \text{-- ... and functions returning Testable properties}
\end{array}$$

$$\begin{array}{l}
\quad \text{-- Run tests:} \\
\mathit{quickCheck} :: \mathit{Testable}\ \mathit{prop} \Rightarrow \mathit{prop} \to \mathit{IO}\ ()
\end{array}$$

$$\begin{array}{llll}
\multicolumn{4}{l}{\quad \text{-- Measure the test case distribution:}} \\
\mathit{collect} & :: (\mathit{Show}\ a, \mathit{Testable}\ p) \Rightarrow a & & \to p \to \mathit{Property} \\
\mathit{label} & :: \mathit{Testable}\ p \Rightarrow & \mathit{String} & \to p \to \mathit{Property} \\
\mathit{classify} & :: \mathit{Testable}\ p \Rightarrow \mathit{Bool} \to & \mathit{String} & \to p \to \mathit{Property}
\end{array}$$

$$\begin{array}{ll}
\mathit{collect}\ x & = \mathit{label}\ (\mathit{show}\ x) \\
\mathit{label}\ s & = \mathit{classify}\ \mathit{True}\ s
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\quad \text{-- Create generators:}} \\
\mathit{choose} & :: \mathit{Random}\ a \Rightarrow (a, a) & \to \mathit{Gen}\ a \\
\mathit{elements} & :: [a] & \to \mathit{Gen}\ a \\
\mathit{oneof} & :: [\mathit{Gen}\ a] & \to \mathit{Gen}\ a \\
\mathit{frequency} & :: [(\mathit{Int}, \mathit{Gen}\ a)] & \to \mathit{Gen}\ a \\
\mathit{sized} & :: (\mathit{Int} \to \mathit{Gen}\ a) & \to \mathit{Gen}\ a \\
\mathit{sequence} & :: [\mathit{Gen}\ a] & \to \mathit{Gen}\ [a] \\
\mathit{vector} & :: \mathit{Arbitrary}\ a \Rightarrow \mathit{Int} & \to \mathit{Gen}\ [a] \\
\mathit{arbitrary} & :: \mathit{Arbitrary}\ a \Rightarrow & \quad \mathit{Gen}\ a \\
\mathit{fmap} & :: (a \to b) \to \mathit{Gen}\ a & \to \mathit{Gen}\ b \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Monad}\ (\mathit{Gen}\ a)\ \textbf{where} \ldots}
\end{array}$$

$$\begin{array}{l}
\quad \text{-- Arbitrary — a class for generators} \\
\textbf{class}\ \mathit{Arbitrary}\ a\ \textbf{where} \\
\quad \mathit{arbitrary} :: \mathit{Gen}\ a \\
\quad \mathit{shrink}\quad\ :: a \to [a]
\end{array}$$