Advanced Functional Programming TDA341/DIT260

Patrik Jansson

2010-08-24

Contact:	Patrik Jansson, ext 5415. Will drop by the exam hall around 15 and around 17 for questions.
Result:	Announced no later than 2010-09-10
Exam check:	Monday 2010-09-13 and Friday 2010-09-17. 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:	3: 24p, 4: 36p, 5: 48p, max: 60p G: 24p, VG: 48p
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.

(40 p) **Problem 1**

A domain specific language *Calc* (inspired by a simple calculator) has the following constructs:

- arithmetics: integer constants, binary $+$, $-$, $*$, $/$, unary invert ($1/$), negate ($0-$)
- state-changers: unary M (memory store), M+ (memory accumulate), and nullary MR (memory recall), MC (memory clear).

The calculator has one memory cell (of **type** *Value* = *Double*) and calculations may fail (end with an error value like *Left* `"Division by zero"`, etc.). Your task is to define a monad *CalcM*, a datatype *Calc* and an evaluator *eval* :: *Calc* → *CalcM Value* for a deep embedding of the calculator language, and some testing infrastructure (see below).

(20 p) **(a)** Define the datatype *Calc* and the evaluator *eval* :: *Calc* → *CalcM Value*. In this sub problem you may use methods from *Monad* and *MonadState* without defining them.

(10 p) **(b)** Define the monad *CalcM* either by using the monad transformers from the appendix or by providing your own *Monad* and *MonadState* instance declarations. Would it be possible to use *CalcM Value* as a shallow embedding?

(5 p) **(c)** Define the monad laws (for *return* and (≫=)) as QuickCheck properties. Give the types of the properties.

(5 p) **(d)** If you wanted to actually run checks of the monad laws for your *CalcM* monad, what more would you need to define and what kind of problems would you run into?

(20 p) **Problem 2**

The below program uses a GADT to express typed abstract syntax for a small language *Expr* (from lecture 7, spring 2010). Your task is to extend the language with a polymorphic if-then-else construct. (Don't repeat the given code unless something needs to change.)

(3 p) **(a)** Add an *If*-constructor to the datatype.

(3 p) **(b)** Add a corresponding case to the *eval* function.

(8 p) **(c)** Add a corresponding case to the *infer* function.

(6 p) **(d)** Add a corresponding case to the *showsPrecExpr* function and provide a type signature for *showsPrecExpr*. The if-then-else construct should have lowest precedence.

```
{-# LANGUAGE GADTs, ExistentialQuantification #-}
module Typed where
import qualified Expr as E   -- Corresponding lang. without GADTs (not shown)
data Expr t where
  Lit   :: (Eq t, Show t) => t                      -> Expr t
  (:+)  ::                   Expr Int -> Expr Int   -> Expr Int
  (:==) :: (Eq t, Show t) => Expr t   -> Expr t     -> Expr Bool
  -- TODO: a)
eOK :: Expr Int
eOK = If (Lit False) (Lit 1) (Lit 2 :+ Lit 1736)
eval :: Expr t -> t
eval (Lit x)      = x
eval (e1 :+ e2)   = eval e1 + eval e2
eval (e1 :== e2)  = eval e1 == eval e2
  -- TODO: b)
```

```
data Type t where
  TInt  :: Type Int
  TBool :: Type Bool

data TypedExpr = forall t. (Eq t, Show t) => Expr t ::: Type t

data Equal a b where
  Refl :: Equal a a

(=?=) :: Type s -> Type t -> Maybe (Equal s t)
TInt  =?= TInt  = Just Refl
TBool =?= TBool = Just Refl
_     =?= _     = Nothing

infer :: E.Expr -> Maybe TypedExpr
infer e = case e of
  E.LitN n -> return (Lit n ::: TInt)
  E.LitB b -> return (Lit b ::: TBool)
  r1 E.:+ r2 -> do
    e1 ::: TInt <- infer r1
    e2 ::: TInt <- infer r2
    return (e1 :+ e2 ::: TInt)
  r1 E.:== r2 -> do
    e1 ::: t1   <- infer r1
    e2 ::: t2   <- infer r2
    Refl        <- t1 =?= t2
    return (e1 :== e2 ::: TBool)
  -- TODO: c)

infixl 6 :+
infix  4 :==
infix  0 :::

instance Show (Type t) where
  show TInt = "Int"
  show TBool = "Bool"
instance Show TypedExpr where
  show (e ::: t) = show e ++ " :: " ++ show t
instance Show t => Show (Expr t) where
  showsPrec = showsPrecExpr

  -- TODO: d) type signature and ...
showsPrecExpr p e = case e of
    Lit n -> shows n
    e1 :+ e2  -> showParen (p > 2) $
      showsPrec 2 e1 . showString " + " . showsPrec 3 e2
    e1 :== e2 -> showParen (p > 1) $
      showsPrec 2 e1 . showString " == " . showsPrec 2 e2
  -- TODO: d) ... printing of the if-then-else construct
```

A Library documentation

A.1 Monoids

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
```

A monoid should satisfy the laws (where the variables are implicitly quantified):

$$\begin{aligned}\mathsf{mappend}\ \mathsf{mempty}\ m &= m\\ \mathsf{mappend}\ m\ \mathsf{mempty} &= m\\ \mathsf{mappend}\ (\mathsf{mappend}\ m_1\ m_2)\ m_3 &= \mathsf{mappend}\ m_1\ (\mathsf{mappend}\ m_2\ m_3)\end{aligned}$$

Example: for any type a lists of as 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
```

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 for a given computation
  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]
```