Advanced Functional Programming
TDA341/DIT260

Ulf Norell

March 11, 2009

Contact:	Ulf Norell, ext 1054.
Aids:	Pencil, food and drink.
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.
	Write your name on each sheet of paper.

Problem 1 (16 p)

Consider the following library for describing strategies for navigating through a maze:

```
type S  -- the type of strategies

forward      :: S           -- Try to take one step forward
left         :: S           -- Turn left 90 degrees
idle         :: S           -- Do nothing
ifObstructed :: S -> S -> S -- Use the first strategy if the way
                            -- forward is blocked, otherwise use
                            -- the second strategy.
(>>>)        :: S -> S -> S -- Perform two strategies in sequence
```

A simple strategy is to move forward until there is an obstruction and then turn left and try to move in that direction instead. Using this library this strategy can be implemented as follows:

```
simple = ifObstructed (left    >>> simple)
                      (forward >>> simple)
```

(a) (3 p)

Before looking at how to implement the library, use the functions above to implement the following strategies

```
right    :: S -- Turn 90 degrees right
turn180  :: S -- Turn around
backward :: S -- Try to take one step backwards
```

If there are no obstructions, `backward >>> forward` should have the same behaviour as `idle`.

Here is a deep embedding of the library.

```
data S = Forward S        -- Forward s  == forward >>> s
       | TurnLeft S       -- TurnLeft s == left >>> s
       | IfObstructed S S
       | Idle

forward      = Forward Idle
left         = TurnLeft Idle
idle         = Idle
ifObstructed = IfObstructed
```

(b) **(5 p)**

Give an implemention of (>>>) for the deep embedding.

The next step is to run our strategies in actual mazes. For this purpose the following library has been provided for us. It defines a type of *mazes* and a type of *ants* that can move through the mazes.

```
type Maze
type Ant  -- contains the location of an ant and
          -- the direction it's facing

  -- Move an ant one step forward, ignoring obstructions
moveAnt    :: Ant -> Ant
  -- Turn an ant 90 degrees left
leftAnt    :: Ant -> Ant
  -- Check if there is an obstruction in front of the ant
obstructed :: Maze -> Ant -> Bool
```

Using this library we can implement a run function `navigate` for our strategies. It takes a maze, an ant, and a strategy and computes where the ant ends up if it uses the given strategy to navigate through the maze.

```
navigate :: Maze -> Ant -> S -> Ant
navigate maze ant (Forward s)
  | obstructed maze ant = navigate maze ant s
  | otherwise           = navigate maze (moveAnt ant) s
navigate maze ant (TurnLeft s)
                        = navigate maze (leftAnt ant) s
navigate maze ant Idle  = ant
navigate maze ant (IfObstructed s t)
  | obstructed maze ant = navigate maze ant s
  | otherwise           = navigate maze ant t
```

Note that an ant trying to move through an obstruction will simply remain in its old position.

(c) **(8 p)**

Give a shallow implementation of the strategy library using the library for ants and mazes. You should give the definition of the type of strategies `S` and the implementations of `forward`, `ifObstructed`, `(>>>)`, and `navigate`.

Problem 2 (10 p)

Consider the following definition of a writer monad:

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

(a) **(5 p)**

Give the definitions for `return` and `>>=` in the following monad instance:

```
instance Monoid w => Monad (Writer w) where
  ...
```

See the appendix for an explanation of the `Monoid` class.

(b) **(5 p)**

Implement the functions `tell` and `listen` from the `MonadWriter` class.

```
-- Output something
tell   :: w -> Writer w ()

-- Listen to the outputs of the argument.
-- listen m should have the same outputs as m
listen :: Writer w a -> Writer w (a, w)
```

Problem 3 (14 p)

Below is part of the type inference algorithm for an expression language without variables. The ... shows where things have been omitted that are not relevant for the problem.

```
data Expr = LitN Int
          | LitB Bool
          | If Expr Expr Expr
          | ...
  deriving Show

data Type = TInt | TBool | ...
  deriving (Show, Eq)

newtype TC a = TC { runTC :: Either String a }
  deriving (Monad, MonadError String)

-- Report a type error.
typeError :: String -> TC a
typeError s = throwError s

-- Infer the type of an expression. Fails if the expression
-- is not well-typed.
infer :: Expr -> TC Type
...
infer (LitN n) = return TInt
infer (LitB b) = return TBool
infer (If a b c) = do
  ta  <- infer a
  unless (ta == TBool) $ typeError "bad condition"
  tb  <- infer b
  tc  <- infer c
  unless (tb == tc) $ typeError "bad branches"
  return tb
```

Here are a few example runs of the algorithm:

```
*Main> runTC $ infer $ If (LitN 3) (LitB True) (LitB False)
Left "bad condition"
*Main> runTC $ infer $ If (LitB False) (LitB True) (LitN 3)
Left "bad branches"
*Main> runTC $ infer $ If (LitB False) (LitN 1) (LitN 3)
Right TInt
```

Now we want to add (immutable) variables to the language by adding two new constructors to the expression type:

```
data Expr = ... -- all the stuff from before
          | Var Name
          | Let Name Expr Expr -- Let x e1 e2 declares a variable
                               -- x with value e1 scoping over e2
                               -- Note: x is not in scope in e1.
```

We are given the following types and functions to work with variables:

```
type Name     -- Variable names
type Context  -- Contains the variables in scope and their types

-- Find the type of a variable in the given context
varType :: Name -> Context -> Maybe Type

-- Extend a context with a new variable
addVar :: Name -> Type -> Context -> Context
```

(a) **(4 p)**

Change the implementation of the `TC` monad to allow us to keep track of the current context during type checking. See the appendix for monad transformers that might be helpful. Make sure not to break the existing implementations of `typeError` and `infer` when changing the implementation of `TC`.

(b) **(6 p)**

Implement the following functions for your improved monad

```
-- Lookup the type of a variable in the context
lookupVar :: Name -> TC Type

-- Extend the context locally
extendContext :: Name -> Type -> TC a -> TC a
```

(c) **(4 p)**

Add cases for the constructors `Var` and `Let` to the `infer` function.

Problem 4 (10 p)

Consider the following type of expressions with explicit application

```
data Expr = Lit Int
          | Plus
          | App Expr Expr -- the application of a function
                          -- expression to an argument
```

In this language the expression $1 + 2$ is modelled as

```
App (App Plus (Lit 1)) (Lit 2)
```

or using infix notation for `App`:

```
(Plus ‘App‘ Lit 1) ‘App‘ Lit 2
```

The following terms are valid elements of the `Expr` type but they don't correspond to well-formed expressions: `Lit 1 ‘App‘ Lit 2` and `App Plus Plus`.

(a) **(6 p)**

Define a generalised datatype (GADT) `Expr t` whose elements correspond only to well-formed expressions of type `t`. For instance

```
(Plus ‘App‘ Lit 1) ‘App‘ Lit 2 :: Expr Int
App Plus (Lit 1)               :: Expr (Int -> Int)
```

(b) **(4 p)**

Implement an evaluator `eval :: Expr t -> t` for your expressions.

Problem 5 (10 p)

Using the STM library (see appendix), we can represent a lock as transaction variable containing a value indicating whether the lock is open or not.

```
data LockState = Locked | Unlocked
type Lock = TVar LockState
```

The idea is that if one thread takes the lock (locks it) then any other thread that tries to take the lock has to wait until the first thread unlocks it. This mechanism can be used to ensure that a thread has exclusive access to some shared data.

(a) **(4 p)**

Implement the following basic interface:

```
-- Create a new lock which starts out in the unlocked state.
newLock :: STM Lock

-- Wait for a lock to become unlocked. Causes
-- it to become locked.
lock :: Lock -> STM ()

-- Causes a locked lock to become unlocked. The behaviour of
-- unlock in case the lock is already unlocked is not
-- specified and you can choose to do whatever you want.
unlock :: Lock -> STM ()
```

(b) **(2 p)**

Define a function

```
criticalSection :: Lock -> IO a -> IO a
```

protecting an IO computation by a lock, making sure that it is not executed until we have managed to take the lock.

(c) **(4 p)**

Define a function

```
lockAny :: [Lock] -> STM Lock
```

that waits on all of the given locks. As soon as any of them becomes unlocked `lockAny` should lock that lock and return it (so that we can unlock it once we're done).

Hint: Use `orElse`.

A Library documentation

A.1 Monoids

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

A monoid should satisfy the laws

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

List is 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

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 STM

```
type STM a
instance Monad STM

-- Run an STM computation. Behaves as if the entire
-- computation is performed in one atomic step. If
-- the computation is aborted (for instance, using retry),
-- it will be reexecuted until it succeeds.
atomically :: STM a -> IO a

-- Abort a computation.
retry :: STM a

-- If the first argument is aborted (using retry), the
-- second argument will be executed. If that one also
-- aborts the entire computation will be aborted.
orElse :: STM a -> STM a -> STM a

-- Transaction variables.
type TVar a

newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```