COMP3048: Lecture 8

A Versatile Design Pattern: Monads

Matthew Pike

University of Nottingham, Ningbo

Perspective (1)

- Design Pattern [Wikipedia]:

  [A] design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.

- Example: In an OO Language like Java or C#, operations on data are tied to classes. Thus:
  - Cannot (directly) add a new operation on data without changing ***all*** involved classes.
  - The code for an operation gets ***spread out*** across all involved classes.

Perspective (2)

- Solution: The ***Visitor*** pattern (or ***double dispatch***):
  - Allows operations to be defined separately from data classes and in one place.
  - Allows operations to be defined by simple “pattern matching” (case analysis).
- Not entirely trivial: takes a lecture to explain. See:

  `http://en.wikipedia.org/wiki/Visitor_pattern`

This Lecture

Functional languages provides separation between operations and data, and typically pattern matching too, “for free”.

However, handling ***effects*** in a ***pure*** language requires work because, by definition, there are no implicit effects in a pure language.

This lecture: A design pattern for effects.

A Blessing and a Curse

- The ***BIG*** advantage of pure functional programming is

  “everything is explicit;”

  i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.
- The ***BIG*** problem with pure functional programming is

  “everything is explicit.”

  Can really add a lot of clutter, especially in large programs.

Example: LTXL Identification (1)

`enterVar` inserts a variable at the given scope level and of the given type into an environment.

- Check that no variable with same name has been defined at the same scope level.
- If not, the new variable is entered, and the ***resulting environment*** is returned.
- Otherwise an ***error message*** is returned.

```
enterVar :: Id -> Int -> Type -> Env
              -> Either Env ErrorMsg
```

Example: LTXL Identification (2)

Goals of LTXL identification phase:

- Annotate each applied identifier occurrence with attributes of the corresponding variable declaration.
  I.e., map unannotated AST **`Exp ()`** to annotated AST **`Exp Attr`**.
- Report conflicting variable definitions and undefined variables.

```
identification ::
      Exp () -> (Exp Attr, [ErrorMsg])
```

Example: LTXL Identification (3)

```
identDefs l env [] = ([], env, [])
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'', ms1++ms2++ms3)
  where
    (e', ms1) = identAux l env e
    (env', ms2) =
        case enterVar i l t env of
            Left env' -> (env', [])
            Right m   -> (env,  [m])
    (ds', env'', ms3) =
      identDefs l env' ds
```

Example: LTXL Identification (4)

Error checking and collection of error messages arguably added a lot of clutter. The ***core*** of the algorithm is this:

```
identDefs l env [] = ([], env)
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'')
  where
    e'            = identAux l env e
    env'          = enterVar i l t env
    (ds', env'') = identDefs l env' ds
```

Example: A Simple Evaluator

```
data Exp = Lit Integer
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Div Exp Exp

eval :: Exp -> Integer
eval (Lit n)     = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Sub e1 e2) = eval e1 - eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Div e1 e2) = eval e1 `div` eval e2
```

Making the evaluator safe (1)

```
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = Just n
safeEval (Add e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 + n2)
```

Making the evaluator safe (2)

```
safeEval (Sub e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 - n2)
```

Making the evaluator safe (3)

```
safeEval (Mul e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 * n2)
```

Making the evaluator safe (4)

```
safeEval (Div e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 ->
                    if n2 == 0
                    then Nothing
                    else Just (n1 `div` n2)
```

Any common pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?

We note:

- Sequencing of evaluations.
- If one evaluation fail, fail overall.
- Otherwise, make result available to following evaluations.

Example: Numbering trees

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

numberTree :: Tree a -> Tree Int
numberTree t = fst (ntAux t 0)
    where
        ntAux (Leaf _)      n = (Leaf n, n+1)
        ntAux (Node t1 t2) n =
            let (t1', n')   = ntAux t1 n
            in let (t2', n'') = ntAux t2 n'
               in (Node t1' t2', n'')
```

Observations

- Repetitive pattern: threading a counter through a ***sequence*** of tree numbering ***computations***.
- It is very easy to pass on the wrong version of the counter!

Can we do better?

Sequencing evaluations (1)

Sequencing is common to both examples, with the outcome of a computation ***affecting*** subsequent computations.

```
evalSeq :: Maybe Integer
           -> (Integer -> Maybe Integer)
           -> Maybe Integer
evalSeq ma f =
    case ma of
        Nothing -> Nothing
        Just a  -> f a
```

Sequencing evaluations (2)

```
safeEval (Add e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 + n2)
evalSeq ma f =
    case ma of
        Nothing -> Nothing
        Just a  -> f a
```

Sequencing evaluations (3)

```
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = Just n
safeEval (Add e1 e2) =
    safeEval e1 `evalSeq` (\n1 ->
    safeEval e2 `evalSeq` (\n2 ->
    Just (n1 + n2)))
safeEval (Sub e1 e2) =
    safeEval e1 `evalSeq` (\n1 ->
    safeEval e2 `evalSeq` (\n2 ->
    Just (n1 - n2)))
```

Sequencing evaluations (4)

```
safeEval (Mul e1 e2) =
    safeEval e1 `evalSeq` (\n1 ->
    safeEval e2 `evalSeq` (\n2 ->
    Just (n1 - n2)))
safeEval (Div e1 e2) =
    safeEval e1 `evalSeq` (\n1 ->
    safeEval e2 `evalSeq` (\n2 ->
    if n2 == 0
    then Nothing
    else Just (n1 `div` n2)))
```

Aside: Scope rules of λ-abstractions

The scope rules of λ-abstractions are such that parentheses can be omitted:

```
safeEval :: Exp -> Maybe Integer

...

safeEval (Add e1 e2) =
    safeEval e1 `evalSeq` \n1 ->
    safeEval e2 `evalSeq` \n2 ->
    Just (n1 + n2)

...
```

Exercise 1: Inline `evalSeq` (1)

```
safeEval (Add e1 e2) =
  safeEval e1 `evalSeq` \n1 ->
  safeEval e2 `evalSeq` \n2 ->
  Just (n1 + n2)
```

$=$

```
safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just a -> (\n1 -> safeEval e2 ...) a
```

Exercise 1: Inline `evalSeq` (2)

=

```
safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> safeEval e2 `evalSeq` (\n2 -> ...)
```

=

```
safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
                 Nothing -> Nothing
                 Just a -> (\n2 -> ...) a
```

Exercise 1: Inline `evalSeq` (3)

=

```
safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
                 Nothing -> Nothing
                 Just n2 -> (Just n1 + n2)
```

`Maybe` viewed as a computation (1)

- Consider a value of type `Maybe a` as denoting a ***computation*** of a value of type `a` that ***may fail***.
- When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.
- I.e. ***failure is an effect***, implicitly affecting subsequent computations.
- Let’s adopt names reflecting our intentions.

Maybe viewed as a computation (2)

Successful computation of a value:

```
mbReturn :: a -> Maybe a
mbReturn = Just
```

Sequencing of possibly failing computations:

```
mbSeq :: Maybe a -> (a -> Maybe b) -> Maybe b
mbSeq ma f =
    case ma of
        Nothing -> Nothing
        Just a  -> f a
```

Maybe viewed as a computation (3)

Failing computation:

```
mbFail :: Maybe a
mbFail = Nothing
```

The safe evaluator revisited

```
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = mbReturn n
safeEval (Add e1 e2) =
    safeEval e1 `mbSeq` \n1 ->
    safeEval e2 `mbSeq` \n2 ->
    mbReturn (n1 + n2)
...
safeEval (Div e1 e2) =
    safeEval e1 `mbSeq` \n1 ->
    safeEval e2 `mbSeq` \n2 ->
    if n2 == 0 then mbFail
    else mbReturn (n1 `div` n2)))
```

Stateful Computations (1)

- A ***stateful computation*** consumes a state and returns a result along with a possibly updated state.
- The following type synonym captures this idea:

  ```
  type S a = Int -> (a, Int)
  ```

  (Only `Int` state for the sake of simplicity.)
- A value (function) of type `S a` can now be viewed as denoting a stateful computation computing a value of type `a`.

Stateful Computations (2)

- When sequencing stateful computations, the resulting state should be passed on to the next computation.
- I.e. ***state updating is an effect***, implicitly affecting subsequent computations.
  (As we would expect.)

Stateful Computations (3)

Computation of a value without changing the state:

```
sReturn :: a -> S a
sReturn a = \n -> (a, n)
```

Sequencing of stateful computations:

```
sSeq :: S a -> (a -> S b) -> S b
sSeq sa f = \n ->
    let (a, n') = sa n
    in f a n'
```

Stateful Computations (4)

Reading and incrementing the state:

```
sInc :: S Int
sInc = \n -> (n, n + 1)
```

Numbering trees revisited

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

numberTree :: Tree a -> Tree Int
numberTree t = fst (ntAux t 0)
    where
        ntAux (Leaf _)    =
            sInc `sSeq` \n -> sReturn (Leaf n)
        ntAux (Node t1 t2) =
            ntAux t1 `sSeq` \t1' ->
            ntAux t2 `sSeq` \t2' ->
            sReturn (Node t1' t2')
```

Observations

- The “plumbing” has been captured by the abstractions.
- In particular, there is no longer any risk of “passing on” the wrong version of the state!

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing identically structured abstractions that encapsulated the effects:
  - A type denoting computations
  - A combinator for computing a value without any effect
  - A combinator for sequencing computations
- In fact, both examples are instances of the general notion of a ***MONAD***.

Monads in Functional Programming

A monad is represented by:

- A type constructor

  ```
  M :: * -> *
  ```

  `M T` represents computations of a value of type `T`.

- A polymorphic function

  ```
  return :: a -> M a
  ```

  for lifting a value to a computation.

- A polymorphic function

  ```
  (>>=) :: M a -> (a -> M b) -> M b
  ```

  for sequencing computations.

Monads in Haskell (1)

In Haskell, the notion of a monad is captured by a ***Type Class***:

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
```

This allows the names of the common functions to be overloaded, and the sharing of derived definitions.

Monads in Haskell (2)

The Haskell monad class have two further methods with default instances:

```
(>>) :: m a -> m b -> m b
m >> k = m >>= \_ -> k

fail :: String -> m a
fail s = error s
```

The `Maybe` monad in Haskell

```
instance Monad Maybe where
    -- return :: a -> Maybe a
    return = Just

    -- (>>=) :: Maybe a -> (a -> Maybe b)
                -> Maybe b
    Nothing  >>= _ = Nothing
    (Just x) >>= f = f x
```

Monad-specific operations

To be useful, monads need to be equipped with additional operations specific to the effects in question. For example:

```
fail :: String -> Maybe a
fail s = Nothing

catch :: Maybe a -> Maybe a -> Maybe a
m1 `catch` m2 =
     case m1 of
          Just _   -> m1
          Nothing -> m2
```

The do-notation (1)

Haskell provides convenient syntax for programming with monads:

$$
\begin{array}{l}
\texttt{do} \\
\qquad \texttt{a <-}\ \ exp_1 \\
\qquad \texttt{b <-}\ \ exp_2 \\
\qquad \texttt{return}\ \ exp_3
\end{array}
$$

is syntactic sugar for

$$
\begin{array}{l}
exp_1\ \texttt{>>= \textbackslash a ->} \\
exp_2\ \texttt{>>= \textbackslash b ->} \\
\texttt{return}\ \ exp_3
\end{array}
$$

The do-notation (2)

Computations can be done solely for effect, ignoring the computed value:

$$\begin{array}{l}\texttt{do} \\ \quad exp_1 \\ \quad exp_2 \\ \quad \texttt{return}\ exp_3\end{array}$$

is syntactic sugar for

$$\begin{array}{l}exp_1\ \texttt{>>= \textbackslash\_ ->} \\ exp_2\ \texttt{>>= \textbackslash\_ ->} \\ \texttt{return}\ exp_3\end{array}$$

The HMTC Diagnostics Monad

```
D              :: * -> *      -- Instances: Monad.
emitInfoD      :: SrcPos -> String -> D ()
emitWngD       :: SrcPos -> String -> D ()
emitErrD       :: SrcPos -> String -> D ()
failD          :: SrcPos -> String -> D a
failNoReasonD  :: D a
failIfErrorsD  :: D ()
stopD          :: D a
runD           :: D a -> (Maybe a, [DMsg])
```

(Roughly: The actual HMTC impl. is more refined.)

Identification Revisited (1)

Recall:

```
enterVar :: Id -> Int -> Type -> Env
              -> Either Env String
```

Let's define a version using the Diagnostics monad:

```
enterVarD :: Id -> Int -> Type -> Env ->D Env
enterVarD i l t env =
    case enterVar i l t env of
        Left env'  -> return env'
        Right m    -> do
            emitErrD NoSrcPos m
            return env
```

Identification Revisited (2)

Now we can define a monadic version of `identDefs`:

```
identDefs :: Int -> Env -> [(Id,Type,Exp ())]
             -> D ([(Id,Type,Exp Attr)], Env)
identDefs l env [] = return ([], env)
identDefs l env ((i,t,e) : ds) = do
    e'           <- identAux l env e
    env'         <- enterVarD i l t env
    (ds', env'') <- identDefs l env' ds
    return ((i,t,e') : ds', env'')
```

Identification Revisited (3)

Compare with the "core" identified earlier!

```
identDefs l env [] = ([], env)
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'')
  where
    e'           = identAux l env e
    env'         = enterVar i l t env
    (ds', env'') = identDefs l env' ds
```

The monadic version is very close to ideal, without sacrificing functionality, clarity, or pureness!

Further Reading

- Graham Hutton. *Programming in Haskell*.
  `http://www.cs.nott.ac.uk/~pszgmh/pih.html`