Advanced Functional Programming

Tim Sheard

Monads part 2

Monads and Interpreters

Small languages

Many programs and systems can be though of as interpreters for "small languages"

Examples:

Yacc – parser generators

Pretty printing

regular expressions

Monads are a great way to structure such systems

Language 1

use a monad

```
eval1 :: T1 -> Id Value
```

use types

```
data Id x = Id x
```

Think about abstract syntax
Use an algebraic data type

```
eval1 (Add1 x y) =
      do {x' <- eval1 x
         ; y' <- eval1 y
         ; return (x' + y')}
eval1 (Sub1 x y) =
      do {x' <- eval1 x
         ; y' <- eval1 y
         ; return (x' - y')}
eval1 (Mult1 x y) =
      do {x' <- eval1 x
         ; y' <- eval1 y
         ; return (x' * y')}
eval1 (Int1 n) = return n
```

construct a purely functional interpreter

```
data T1 = Add1 T1 T1
        | Sub1 T1 T1
        | Mult1 T1 T1
        | Int1 Int
```

figure out what a value is

```
type Value = Int
```

Effects and monads

- When a program has effects as well as returning a value, use a monad to model the effects.
- This way your reference interpreter can still be a purely functional program
- This helps you get it right, lets you reason about what it should do.
- It doesn't have to be how you actually encode things in a production version, but many times it is good enough for even large systems

Monads and Language Design

Monads are important to language design because:

– The meaning of many languages include effects. It's good to have a handle on how to model effects, so it is possible to build the "reference interpreter"

– Almost all compilers use effects when compiling. This helps us structure our compilers. It makes them more modular, and easier to maintain and evolve.

– Its amazing, but the number of different effects that compilers use is really small (on the order of 3-5). These are well studied and it is possible to build libraries of these monadic components, and to reuse them in many different compilers.

An exercise in language specification

- In this section we will run through a sequence of languages which are variations on language 1.
- Each one will introduce a construct whose meaning is captured as an effect.
- We'll capture the effect first as a pure functional program (usually a higher order object, i.e. a function , but this is not always the case, see exception and output) then in a second reference interpreter encapsulate it as a monad.
- The monad encapsulation will have a amazing effect on the structure of our programs.

Monads of our exercise

```
data Id x = Id x

data Exception x = Ok x | Fail

data Env e x = Env (e -> x)

data Store s x = St(s -> (x,s))

data Mult x = Mult [x]

data Output x = OP(x,String)
```

Failure effect

```
eval2a :: T2 -> Exception Value

eval2a (Add2 x y) =
  case (eval2a x,eval2a y) of
    (Ok x', Ok y') -> Ok(x' + y')
    (_,_) -> Fail
eval2a (Sub2 x y) = ...
eval2a (Mult2 x y) = ...
eval2a (Int2 x) = Ok x
eval2a (Div2 x y) =
  case (eval2a x,eval2a y)of
    (Ok x', Ok 0) -> Fail
    (Ok x', Ok y') -> Ok(x' `div` y')
    (_,_) -> Fail
```

```
data Exception x
  = Ok x | Fail

data T2
  = Add2 T2 T2
  | Sub2 T2 T2
  | Mult2 T2 T2
  | Int2 Int
  | Div2 T2 T2
```

Another way

```
eval2a (Add2 x y) =
  case (eval2a x,eval2a y) of
    (Ok x', Ok y') -> Ok(x' + y')
    (_,_) -> Fail


eval2a (Add2 x y) =
  case eval2a x of
    Ok x' -> case eval2a y of
               Ok y' -> Ok(x' + y')
             | Fail -> Fail
    Fail -> Fail
```

Note there are several orders in which we could do things

Monadic Failure

```
eval2 :: T2 -> Exception Value
eval2 (Add2 x y) =
 do { x' <- eval2 x
    ; y' <- eval2 y
    ; return (x' + y')}
eval2 (Sub2 x y) =
 do { x' <- eval2 x
    ; y' <- eval2 y
    ; return (x' - y')}
eval2 (Mult2 x y) = ...
eval2 (Int2 n) = return n
eval2 (Div2 x y) =
 do { x' <- eval2 x
    ; y' <- eval2 y
    ; if y'==0
         then Fail
         else return
            (div x' y')}
```

```
eval1 :: T1 -> Id Value
eval1 (Add1 x y) =
      do {x' <- eval1 x
         ; y' <- eval1 y
         ; return (x' + y')}
eval1 (Sub1 x y) =
      do {x' <- eval1 x
         ; y' <- eval1 y
         ; return (x' - y')}
eval1 (Mult1 x y) = ...
eval1 (Int1 n) = return n
```

Compare with
language 1

environments and variables

```
eval3a :: T3 -> Env Map Value
eval3a (Add3 x y) =
 Env(\e ->
     let Env f = eval3a x
         Env g = eval3a y
     in (f e) + (g e))
eval3a (Sub3 x y) = ...
eval3a (Mult3 x y) = ...
eval3a (Int3 n) = Env(\e -> n)
eval3a (Let3 s e1 e2) =
 Env(\e ->
     let Env f = eval3a e1
         env2 = (s,f e):e
         Env g = eval3a e2
     in g env2)
eval3a (Var3 s) = Env(\ e -> find s e)
```

```
data Env e x
  = Env (e -> x)

data T3
  = Add3 T3 T3
  | Sub3 T3 T3
  | Mult3 T3 T3
  | Int3 Int
  | Let3 String T3 T3
  | Var3 String

Type Map =
    [(String,Value)]
```

Monadic Version

```
eval3 :: T3 -> Env Map Value
eval3 (Add3 x y) =
 do { x' <- eval3 x
    ; y' <- eval3 y
    ; return (x' + y')}
eval3 (Sub3 x y) = ...
eval3 (Mult3 x y) = ...
eval3 (Int3 n) = return n
eval3 (Let3 s e1 e2) =
 do { v <- eval3 e1
     ; runInNewEnv s v (eval3 e2) }
eval3 (Var3 s) = getEnv s
```

Multiple answers

```
eval4a :: T4 -> Mult Value
eval4a (Add4 x y) =
 let Mult xs = eval4a x
     Mult ys = eval4a y
 in Mult[ x+y | x <- xs, y <- ys ]
eval4a (Sub4 x y) = …
eval4a (Mult4 x y) = …
eval4a (Int4 n) = Mult [n]
eval4a (Choose4 x y) =
  let Mult xs = eval4a x
      Mult ys = eval4a y
  in Mult (xs++ys)
eval4a (Sqrt4 x) =
  let Mult xs = eval4a x
  in Mult(roots xs)
```

```
data Mult x
  = Mult [x]

data T4
   = Add4 T4 T4
   | Sub4 T4 T4
   | Mult4 T4 T4
   | Int4 Int
   | Choose4 T4 T4
   | Sqrt4 T4
```

```
roots [] = []
roots (x:xs) | x<0 = roots xs
roots (x:xs) =  y : z : roots xs
  where y = root x
        z = negate y
```

Monadic Version

```
eval4 :: T4 -> Mult Value
eval4 (Add4 x y) =
 do { x' <- eval4 x
    ; y' <- eval4 y
    ; return (x' + y')}
eval4 (Sub4 x y) = …
eval4 (Mult4 x y) = …
eval4 (Int4 n) = return n
eval4 (Choose4 x y) = merge (eval4a x) (eval4a y)
eval4 (Sqrt4 x) =
 do { n <- eval4 x
    ; if n < 0
        then none
        else merge (return (root n))
                   (return(negate(root n))) }
```

```
merge :: Mult a -> Mult a -> Mult a
merge (Mult xs) (Mult ys) = Mult(xs++ys)
none = Mult []
```

Print statement

```
eval6a :: T6 -> Output Value
eval6a (Add6 x y) =
 let OP(x',s1) = eval6a x
     OP(y',s2) = eval6a y
 in OP(x'+y',s1++s2)
eval6a (Sub6 x y) = ...
eval6a (Mult6 x y) = ...
eval6a (Int6 n) = OP(n,"")
eval6a (Print6 mess x) =
 let OP(x',s1) = eval6a x
 in OP(x',s1++mess++(show x'))
```

```
data Output x
  = OP(x,String)

data T6
 = Add6 T6 T6
 | Sub6 T6 T6
 | Mult6 T6 T6
 | Int6 Int
 | Print6 String T6
```

monadic form

```
eval6 :: T6 -> Output Value
eval6 (Add6 x y) =  do { x' <- eval6 x
                         ; y' <- eval6 y
                         ; return (x' + y')}
eval6 (Sub6 x y) =  do { x' <- eval6 x
                         ; y' <- eval6 y
                         ; return (x' - y')}
eval6 (Mult6 x y) = do { x' <- eval6 x
                         ; y' <- eval6 y
                         ; return (x' * y')}
eval6 (Int6 n) = return n
eval6 (Print6 mess x) =
 do { x' <- eval6 x
    ; printOutput (mess++(show x'))
    ; return x'}
```

Why is the monadic form so regular?

- The Monad makes it so.
  - In terms of effects you wouldn't expect the code for Add, which doesn't affect the printing of output to be effected by adding a new action for Print
- The Monad "hides" all the necessary detail.
- An Monad is like an abstract datatype (ADT).
  - The actions like Fail, runInNewEnv, getEnv, Mult, getstore, putStore and printOutput are the interfaces to the ADT
- When adding a new feature to the language, only the actions which interface with it need a big change.
  - Though the *plumbing* might be affected in all actions

Plumbing

<table>
<tr>
<td>
<pre>
case (eval2a x,eval2a y)of
    (Ok x', Ok y') ->
                Ok(x' + y')
    (_,_) -> Fail
</pre>
</td>
<td>
<pre>
Env(\e ->
  let Env f = eval3a x
      Env g = eval3a y
  in (f e) + (g e))
</pre>
</td>
</tr>
<tr>
<td>
<pre>
let Mult xs = eval4a x
    Mult ys = eval4a y
in Mult[x+y |
         x <- xs, y <- ys]
</pre>
</td>
<td>
<pre>
St(\s->
  let St f = eval5a x
      St g = eval5a y
      (x',s1) = f s
      (y',s2) = g s1
  in(x'+y',s2))
</pre>
</td>
</tr>
<tr>
<td>
<pre>
let OP(x',s1) = eval6a x
    OP(y',s2) = eval6a y
in OP(x'+y',s1++s2)
</pre>
</td>
<td>
The unit and bind of the monad abstract the plumbing.
</td>
</tr>
</table>

Adding Monad instances

When we introduce a new monad, we need to define a few things

1. The "plumbing"
   - The return function
   - The bind function
2. The operations of the abstraction
   - These differ for every monad and are the interface to the "plumbing", the methods of the ADT
   - They isolate into one place how the plumbing and operations work

The Id monad

```
data Id x = Id x

instance Monad Id where
  return x = Id x
  (>>=) (Id x) f = f x
```

There are no operations, and only the simplest plumbing

The Exception Monad

```
Data Exceptionn x = Fail | Ok x

instance Monad Exception where
  return x = Ok x
  (>>=) (Ok x) f = f x
  (>>=) Fail f = Fail
```

There only operations is Fail and the plumbing is matching against Ok

The Environment Monad

```
instance Monad (Env e) where
  return x = Env(\ e -> x)
  (>>=) (Env f) g = Env(\ e -> let Env h = g (f e)
                               in h e)

type Map = [(String,Value)]

getEnv :: String -> (Env Map Value)
getEnv nm = Env(\ s -> find s)
  where find [] = error ("Name: "++nm++" not found")
        find ((s,n):m) = if s==nm then n else find m

runInNewEnv :: String -> Int -> (Env Map Value) ->
              (Env Map Value)
runInNewEnv s n (Env g) =
   Env(\ m -> g ((s,n):m))
```

The Store Monad

```
data Store s x = St(s -> (x,s))
instance Monad (Store s) where
  return x = St(\ s -> (x,s))
  (>>=) (St f) g = St h
    where h s1 = g' s2 where (x,s2) = f s1
                             St g' = g x
getStore :: String -> (Store Map Value)
getStore nm = St(\ s -> find s s)
  where find w [] = (0,w)
        find w ((s,n):m) = if s==nm then (n,w) else find w m

putStore :: String -> Value -> (Store Map ())
putStore nm n = (St(\ s -> ((),build s)))
  where build [] = [(nm,n)]
        build ((s,v):zs) =
           if s==nm then (s,n):zs else (s,v):(build zs)
```

The Multiple results monad

```
data Mult x = Mult [x]

instance Monad Mult where
  return x = Mult[x]
  (>>=) (Mult zs) f = Mult(flat(map f zs))
     where flat [] = []
           flat ((Mult xs):zs) = xs ++ (flat zs)
```

The Output monad

```
data Output x = OP(x,String)

instance Monad Output where
  return x = OP(x,"")
  (>>=) (OP(x,s1)) f =
       let OP(y,s2) = f x in OP(y,s1 ++ s2)

printOutput:: String -> Output ()
printOutput s = OP((),s)
```

Further Abstraction

- Not only do monads hide details, but they make it possible to design language fragments
- Thus a full language can be constructed by composing a few fragments together.
- The complete language will have all the features of the sum of the fragments.
- But each fragment is defined in complete ignorance of what features the other fragments support.

The Plan

Each fragment will

1. Define an abstract syntax data declaration, abstracted over the missing pieces of the full language
2. Define a class to declare the methods that are needed by that fragment.
3. Only after tying the whole language together do we supply the methods.

There is one class that ties the rest together

```
class Monad m => Eval e v m where
   eval :: e -> m v
```

The Arithmetic Language Fragment

```
instance
  (Eval e v m,Num v)
     => Eval (Arith e) v m   where
  eval (Add x y) =
     do { x' <- eval x
        ; y' <- eval y
        ; return (x'+y') }
  eval (Sub x y) =
     do { x' <- eval x
        ; y' <- eval y
        ; return (x'-y') }
  eval (Times x y) =
     do { x' <- eval x
        ; y' <- eval y
        ; return (x'* y') }
  eval (Int n) = return (fromInt n)
```

```
class Monad m =>
  Eval e v m where
   eval :: e -> m v

data Arith x
  = Add x x
  | Sub x x
  | Times x x
  | Int Int
```

The syntax fragment

The divisible Fragment

```
instance
 (Failure m,
  Integral v,
  Eval e v m) =>
    Eval (Divisible e) v m where

 eval (Div x y) =
   do { x' <- eval x
      ; y' <- eval y
      ; if (toInt y') == 0
           then fails
           else return(x' `div` y')
      }
```

```
data Divisible x
  = Div x x
```

The syntax fragment

```
class Monad m =>
  Failure m where
    fails :: m a
```

The class with the necessary operations

The LocalLet fragment

```
data LocalLet x
  = Let String x x
  | Var String

class Monad m => HasEnv m v where
  inNewEnv :: String -> v -> m v -> m v
  getfromEnv :: String -> m v

instance (HasEnv m v,Eval e v m) =>
                Eval (LocalLet e) v m where
  eval (Let s x y) =
    do { x' <- eval x
       ; inNewEnv s x' (eval y)
       }
  eval (Var s) = getfromEnv s
```

The syntax fragment

The operations

The assignment fragment

```
data Assignment x
  = Assign String x
  | Loc String
```

The syntax fragment

```
class Monad m => HasStore m v where
  getfromStore :: String -> m v
  putinStore :: String -> v -> m v
```

The operations

```
instance (HasStore m v,Eval e v m) =>
            Eval (Assignment e) v m     where
  eval (Assign s x) =
    do { x' <- eval x
       ; putinStore s x' }
  eval (Loc s) = getfromStore s
```

The Print fragment

```
data Print x
  = Write String x
```

The syntax fragment

```
class (Monad m,Show v) => Prints m v where
  write :: String -> v -> m v
```

The operations

```
instance (Prints m v,Eval e v m) =>
             Eval (Print e) v m        where

  eval (Write message x)  =
    do { x' <- eval x
       ; write message x' }
```

The Term Language

```
data Term
  = Arith (Arith Term)
  | Divisible (Divisible Term)
  | LocalLet (LocalLet Term)
  | Assignment (Assignment Term)
  | Print (Print Term)

instance (Monad m, Failure m, Integral v,
          HasEnv m,v HasStore m v, Prints m v) =>
         Eval Term v m where
  eval (Arith x) = eval x
  eval (Divisible x) = eval x
  eval (LocalLet x) = eval x
  eval (Assignment x) = eval x
  eval (Print x) = eval x
```

Tie the syntax fragments together

Note all the dependencies

A rich monad

In order to evaluate `Term` we need a rich monad, and value types with the following constraints.

- `Monad m`
- `Failure m`
- `Integral v`
- `HasEnv m v`
- `HasStore m v`
- `Prints m v`

The Monad M

```
type Maps x = [(String,x)]
data M v x =
    M(Maps v -> Maps v -> (Maybe x,String,Maps v))

instance Monad (M v) where
 return x = M(\ st env -> (Just x,[],st))
 (>>=) (M f) g = M h
   where h st env = compare env (f st env)
         compare env (Nothing,op1,st1) = (Nothing,op1,st1)
         compare env (Just x, op1,st1)
            = next env op1 st1 (g x)
         next env op1 st1 (M f2)
           = compare2 op1 (f2 st1 env)
         compare2 op1 (Nothing,op2,st2)
           = (Nothing,op1++op2,st2)
         compare2 op1 (Just y, op2,st2)
           = (Just y, op1++op2,st2)
```

Language Design

- **Think only about Abstract syntax**
  this is fairly stable, concrete syntax changes much more often
- **Use algebraic datatypes to encode the abstract syntax**
  use a language which supports algebraic datatypes
- **Makes use of types to structure everything**
  Types help you think about the structure, so even if you use a language with out types. Label everything with types
- **Figure out what the result of executing a program is**
  this is your “value” domain. values can be quite complex
  think about a purely functional encoding. This helps you get it right. It doesn’t have to be how you actually encode things. If it has effects use monads to model the effects.

Language Design
(cont.)

Construct a purely functional interpreter for the abstract syntax.

This becomes your “reference” implementation. It is the standard by which you judge the correctness of other implementations.

Analyze the target environment

What properties does it have?

What are the primitive actions that get things done?

Relate the primitive actions of the target environment to the values of the interpreter.

Can the values be implemented by the primitive actions?

mutable variables

```
eval5a :: T5 -> Store Map Value
eval5a (Add5 x y) =
  St(\s-> let St f = eval5a x
              St g = eval5a y
              (x',s1) = f s
              (y',s2) = g s1
          in(x'+y',s2))
eval5a (Sub5 x y) = ...
eval5a (Mult5 x y) = ...
eval5a (Int5 n) = St(\s ->(n,s))
eval5a (Var5 s) = getStore s
eval5a (Assign5 nm x) = St(\s ->
  let St f = eval5a x
      (x',s1) = f s
      build [] = [(nm,x')]
      build ((s,v):zs) =
       if s==nm then (s,x'):zs
               else (s,v):(build zs)
  in (0,build s1))
```

```
data Store s x
 = St (s -> (x,s))

data T5
  = Add5 T5 T5
  | Sub5 T5 T5
  | Mult5 T5 T5
  | Int5 Int
  | Var5 String
  | Assign5 String T5
```

Monadic Version

```
eval5 :: T5 -> Store Map Value
eval5 (Add5 x y) =
  do {x' <- eval5 x
     ; y' <- eval5 y
     ; return (x' + y')}
eval5 (Sub5 x y) = ...
eval5 (Mult5 x y) = ...
eval5 (Int5 n) = return n
eval5 (Var5 s) = getStore s
eval5 (Assign5 s x) =
      do { x' <- eval5 x
         ; putStore s x'
         ; return x' }
```