Guest Lecture in Advanced Functional Programming

Monads and Effects in Purely Functional Programming

Johan Glimming

Department of Mathematics, Uppsala University

November 19, 2010

Outline of this Lecture

Introduction

Wadler wrote the following in his 1992 Marktoberdorf summerschool tutorial, discussing an application of monads:

"The discovery of such a simple solution comes as a surprise, considering the plethora of more elaborate solutions that have been proposed.

Why was this solution not discovered twenty years ago? One possible reason is that the data types involve higher-order functions in an essential way. The usual axiomatisation [...] involves only first-order functions, so perhaps it did not occur to anyone to search for an abstract data type based on higher-order functions.

That monads led to the discovery of the solution must count as a point in their favour."

Effects and Monads

Monads

Shall we be pure och impure? (Wadler 1992)

Haskell enjoys referential transparency because the value of an expression depends only on its free variables. There are no side-effects, no state, etc. Nor does it matter in what order we evaluate an expression (the so-called *Church-Rosser* property).

However, sometimes this is a limitation because we have to explicitly give all input and output parameters ("plumbing") and it is preferable to have a global state. There are also other situations where side-effects are useful ...

Monads

Monads constitute a mechanism for putting a number of operations in sequence, such that we know which operation should be evaluated before other operations. This is Haskell's answer to "how do I handle side-effects in a purely functional setting?" The answer is, as we shall see, to treat computations/effects as "first-class" values with which we can compute.

Monad = Computational Type

We introduce monads:

Monads

Monads encapsulate effects such as state changes, variable assignments, error handling, input and output into an abstract datatype.

Monads are defined in Haskell using functions, datatypes, and type classes.

History

"The abduction of the word *monad* from Leibniz was aided and abetted by the pun on monoid" (Bird).

- Moggi (1989) introduced monads as a way of structuring the denotational semantics of programming languages.
- Later, Wadler (1992) popularised monads as a programming tool for functional programming.

In fact, a monad is a mathematical notion which arises in category theory, and is similar to a monoid (but at higher order). However, no knowledge of categories is needed for learning about and using monads in Haskell. Monads can safely be understood or computational types, i.e. a monad *m a* should be thought of as a computation *m* producing something of type *a*.

Writing an Evaluator

Consider writing a small evaluator:

- If we decide to add error handling, all recursive calls must be changed as well as the type of the program. This would not necessarily be the case in an imperative language (exceptions!).
- If we want to add a counter that counts the number of evaluation steps, then we cannot – like for imperative languages – add just a global variable. Instead an *integer has to be passed around during execution, explicitly.*
- Adding an execution trace cannot be added whereas for imperative languages this is just a *side-effect*, e.g. printing.

Let us consider these three cases (without monads).

Basic Evaluator

We choose a simple tree structure for the terms that we will evaluate:

$$\textbf{data}\ \mathit{Term} = \mathit{Con}\ \mathit{Int} \mid \mathit{Div}\ \mathit{Term}\ \mathit{Term}$$

Here is how we would write a program that evaluates expressions/terms:

$$\begin{aligned}&\mathit{eval} :: \mathit{Term} \to \mathit{Int}\\ &\mathit{eval}\ (\mathit{Con}\ x) = x\\ &\mathit{eval}\ (\mathit{Div}\ t\ u) = (\mathit{eval}\ t)\ \mathit{div}\ (\mathit{eval}\ u)\end{aligned}$$

Basic Evaluator: a problem

Haskell uses *error* `"division by zero"` which is indistinguishable to a diverging program, although many programs print a message on screen and abort:

$$\begin{aligned}&> 4\ `\mathit{div}`\ 0\\ &***\mathit{Exception}: \mathit{divide\ by\ zero}\end{aligned}$$

We would like the evaluator to take care of such exceptional circumstances rather than diverging or producing a run-time error of this kind.

Evaluator with Exceptions

Suppose, therefore, that we want better error handling. We essentially use *Maybe*, but add a string in the case of *Nothing* (and rename the constructors):

$$\textbf{data}\ M\ a = \mathit{Raise}\ \mathit{String} \mid \mathit{Return}\ a$$

Hence, given $v :: M\ a$, we have either a value of type a or an exception.

Evaluator with Exceptions

An exception is "threaded" through the recursive evaluation after being raised. Note that we do not have with this *M* a mechanism to try again, merely to throw exceptions:

```
eval :: Term → M Integer
eval (Con n) = Return n
eval (Div a b) = case eval a of
  Raise s → Raise s
  Return n → case eval b of
    Raise t → Raise t
    Return m → if m == 0
      then Return (n div m)
      else Raise "Division by zero"
```


Evaluator with Tracing/Output

For tracing we tuple the result, i.e. we "thread" a string through the entire evaluation, appending output as we go:

```
data M a = MkOut (String, a)
eval :: Term → Out Int
eval (Con x) = MkOut (line (Con a) a, a)
eval (Div a b) =
  let (x, n) = eval a
      (y, m) = eval b
  in
  (x ++ y ++ line (Div t u) (a `div` b), a `div` b)
line :: Term → Int → Output
line t a = "eval (" ++ showterm t ++ ") <=" ++ showint a ++ "\n"
```


Monads: Id, Maybe, List, State

Monad Operations

The type class Monad is introduced, with two member functions for each instance *m a*:

- *return* :: $a \to m\ a$
- *bind* :: $m\ a \to (a \to m\ b) \to m\ b$

Encapsulating Effects

Suppose $m :: \star \to \star$ is a type constructor which instantiates the class *Monad*. Then

- $m\ ()$ denotes a type $()$ together with an effect of type m. Example: $putChar :: Char \to IO\ ()$ writes a character on the screen.
- $a \to m\ b$ denotes a function type, where the result is both a value b and an effect of type m. I.e. the evaluation of that function on an argument of type a not only produces a value of type b, but also allows for an effect of type m. Example: $readn :: Int \to IO\ String$ reads n characters (Bird pg. 328).

In this sense, $m\ a$ should be read as computation m yielding result a. In this lecture we will see how function of types such as $a \to m\ b$ can be composed similar to usual function composition. This ability is what makes a monad.

Recap: Types and Kinds

- Monomorphic types such as *Int*, *Bool*, *Unit* etc.
- Polymorphic types such as $[a]$, *Tree* a, etc.
- Monomorphic instances of polymorphic types, e.g. $[Int]$, $Tree\ [Bool]$.

We can view *Int*, *Bool* as nullary type constructors, whereas $[\,]$, *Tree*, etc, are unary constructors. A constructor is in fact a function from types to types.

$Int, Bool, [Int], [Bool], Tree\ Bool :: \star$
$[\,], Tree :: \star \to \star$

The symbol $\star$ is the type of types, and is called a kind. These higher-order types are used in Haskell together with type classes.

The *Functor* class

The *Functor* class demonstrates the use of high-order types:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

Note that f is applied here to one (type) argument, so should have kind $\star \to \star$. For example:

```
instance Functor Tree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)
```

Or, using the function *mapTree* previously defined:

```
instance Functor Tree where
  fmap = mapTree
```

Obviously, $[a]$ is also an instance of *Functor*.

The *Monad* class

Monads are, quite simply, instances of the following type class (subject to a suitable set of member functions):

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

The operation $>\!\!>\!\!=$ is pronounced bind and *return* is just return. The idea is that $>\!\!>\!\!=$ allows composition of effectful computations, and *return* allows all ordinary values to be injected into the effectful form in some standard way. I.e. every value is also a value with effect m, using *return*.

Monads explained

Let us stress this again: the two essential operations are $\gg=$ (pronounced: "bind") and *return*. Any instance of the class must provide these two operations/methods:

- *return* to give back a value without other effects/computations carried out.
- $\gg=$ to combine two computations (monads) when a value is passed from one computation to another (note that this is an infix binary operator in Haskell)

Note how that $\gg$, sequence, operator simply ignores any arguments passed from a previous computation.

Monad and Functor Laws

Functor laws:

$$fmap\ id = id$$
$$fmap\ (f \circ g) = fmap\ f \circ fmap\ g$$

Monad laws:

$$return\ a \gg= k = k\ a$$
$$m \gg= return = m$$
$$m \gg= (\lambda x \rightarrow k\ x \gg= h) = (m \gg= k) \gg= h$$

Note special case of last law:

$$m1 \gg (m2 \gg m3) = (m1 \gg m2) \gg m3$$

Connecting law:

$$fmap\ f\ xs = xs \gg= (return \circ f)$$

Id - the trivial monad

If we understand monads as "computational types" (or the type of an effect), then it is reasonable to expect a trivial kind of computation, namely a computation without any effect, i.e. pure functions:

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

Now $x \gg= (\lambda x \rightarrow f\ x) \gg= (\lambda y \rightarrow g\ y)$ is just $(g \circ f)\ x$, i.e. monads flip the arguments around in application. The result is (up to isomorphism) just the usual functions and composition of these. You can take this example as giving some intuition about how monads work. In more complex examples there is some extra information transferred between bind operations as well.

Id is a monad

Note that a computation involving the *Id* is no different from a value. *Id* is a monad for the following reason:

Proposition

Id as defined on the previous slide satisfies the monad laws.

This is a trivial exercise in algebraic manipulation of the involved two definitions.

Ids - a strictness monad

Working towards more interesting monads, we consider a variation of *Id* which forces the evaluation of each computation:

```
instance Monad Ids where
  return a = Ids a
  (Ids x) >>= f = Ids (f $! x)
```

Using this monad we sequence computations, but we also force each argument to be evaluated before it is passed on to the next computation ...

The *Maybe* Monad

Recall the *Maybe* data type:

```
data Maybe a = Just a
             | Nothing
```

It is both a *Functor* and a *Monad* by default in the Prelude:

```
instance Monad Maybe where
  Just x >>= k   = k x
  Nothing >>= k  = Nothing
  return x       = Just x
  fail s         = Nothing
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

We can check that the monad laws are satisfied by these definitions, i.e. that $>>=$ is "associative" and *return* "unit".

More *Maybe*

Suppose you have written some Haskell code for $f :: a \to Maybe\ b$ and $g :: b \to Maybe\ c$, similar to the evaluator we saw in the beginning with exceptions:

```
case (f x) of
  Nothing → Nothing
  Just y = case (g y) of
    Nothing → Nothing
    Just z → Just z
```

The fact that *Maybe* is a *Monad* instance, means that we can write this code much more concisely, structuring the program after the effect it produces. In this case, *the effect is a potential failure to compute a value*.

More *Maybe*

```
f x >>= λy →
  g y >>= λz →
    return z
```

Or more concisely:

```
f x >>= g
```

Compare this to ordinary composition $g\ (f\ x) = (g \circ f)\ x$ (which would not work here since the types are wrong). Note that

these two expressions are equal (assuming we have a monad, i.e. we have proved the monad laws). The proof requires monad laws and also relies on the validity of an "eta rule" $\lambda y \to g\ y = g$ (c.f. extensionality)

Even more *Maybe*

But often programmers prefer a concise notation. So we are allowed to write the following instead, which translates to the previous code:

```
do u ← f x
   z ← g y
   return z
```

But what does this **do** syntax mean more precisely?

do-notation

Monads can be used in Haskell to emulate imperative programming. In fact, a special notation, known as **do**-notation, is designed to allow precisely this (Peyton Jones and Wadler, 1993).

It is important to stress that monads are by themselves not an extension of the language as such. Rather it is a set of definitions of functions, datatypes, and type classes, serving a particular purpose, plus the extra syntactic sugar for **do** notation to allow nice sequencing of computations.

do-notation example

First do the "command" *putStr* (and do not collect its result), next do *getChar*, binding the value produced by that command to the variable *c*, finally return the value *c* as a result from the whole computation.

```
test = do putStr "Hello"
   c ← getChar
   return c
```

If $c :: a$, then $test :: IO\ a$ in this example. Note that we do not need to use *return* here – this should be intuitively clear since *return* gives no effect.

Syntactic sugar for monads

The "do" syntax in Haskell is shorthand for *Monad* operations, as captured by these rules:

1. **do** e ➨ e
2. **do** $e_1; e_2; ...; e_n$ ➨ $e_1 \gg$ **do** $e_2; ...; e_n$
3. **do** $pat \leftarrow e_1; ....; e_n$ ➨
   ```
   let ok pat = do e2; ...; en
       ok _ = fail "..."
   in e1 >>= ok
   ```
4. **do let** $decllist; e_2; ...; e_n$ ➨ **let** $decllist$ **in do** $e_2; ...; e_n$

Note the following special case:

3a. **do** $x \leftarrow e_1; e_2; ...e_n$ ➨ $e_1 \gg\!= \lambda x \rightarrow$ **do** $e2; ...; e_n$

do-notation desugared

"do" syntax can be completely eliminated using these rules:

```
do putStr "Hello"
   c ← getChar
   return c
putStr "Hello" ≫    -- by rule (2)
do c ← getChar
   return c
putStr "Hello" ≫    -- by rule (3a)
getChar ≫= λc → do return c
putStr "Hello" ≫    -- by rule (1)
getChar ≫= λc → return c
putStr "Hello" ≫    -- by currying
getChar ≫= return
```


The Monad Laws using *do*-notation

```
do x ← return a; k x      =  k a
do x ← m; return x        =  m
do x ← m; y ← k x; h y    =  do y ← (do x ← m; k x); h y
do m1; m2; m3             =  do (do m1; m2); m3
fmap f xs                 =  do x ← xs; return (f x)
```

For example, using the second rule above, the example given earlier can be simplified to just:

```
do putStr "Hello"
   getChar
```

or, after desugaring: *putStr* `"Hello"` ≫ *getChar*

Exercise: derive the above laws from the usual monad laws.

The *List* Monad

The *List* data type is also a *Monad*:

```
instance Monad [] where
   m ≫= k  = concat (map k m)
   return x = [x]
   fail x   = []
```

For example:

```
do x ← [1,2,3]
   y ← [4,5]
   return (x,y)
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

In other words, monadic binding takes a list of values, applies a function to each of them, collecting all generated results together. The *return* function creates a singleton list. $[1,2,3] \gg\!= (\lambda x \to [4,5] \gg\!= (\lambda y \to return\ (x,y)))$ is the expansion of the **do** sugaring!

List monad explained

The *List* monad's return method simply constructs a singleton list, i.e. injects a value into the obvious one-element list with this value.

The idea behind the implementation of ≫= for *List* is to apply an operation to all elements in the left hand argument (a list). While doing so we gather all possible values that arises, and in a sense we generate lists.

I.e. the previous computation (i) applies the right hand argument to all the elements of the left hand argument (the list), and (ii) gathers the resulting values together, here via generation of a singleton lists of pairs.

List - what it is

The *List* monad gives us an encapsulation of nondeterministic computations, i.e. computations that can generate many different results, rather than one. This means that we can simulate relations using functions, e.g. $a \to [b]$ assigns to each value in the domain a set (list) of values in the codomain.

List vs. List Comprehension

The following true equation shows the relationship:

do $x \leftarrow xs$
  $return\ (f\ x) = [f\ x \mid x \leftarrow xs]$

List - more than a monad

In fact, *List* has slightly more algebraic structure than what is provided by the *Monad* type class. *List* is also an instance of the *MonadPlus* class as follows:

instance *MonadPlus* [ ] **where**
  $mzero = [\,]$
  $mplus\ m\ n = m +\!\!+ n$

In this case, $\gg\!=$ distrbiutes through *mplus* and *mzero* is a zero "element". I.e. the following laws hold:

$zero \gg\!= k = zero$
$k \gg\!= \lambda a \to zero = zero$
$(m\ mplus\ n) \gg\!= k = (m \gg\!= k)\ mplus\ (n \gg\!= k)$

This structure will reappear when we consider parser monads in the next lecture (since these are based on the *List* monad).

State Monad: how it arises

Compare

$f :: (s, a) \to (s, b)$

to

$f' :: a \to (s \to (s, b))$

The second is the curried form of the first, i.e. $curry\ f = f'$. In fact, we have $(s, a) \to (s, b) \cong a \to (s \to (s, b))$ using *curry* and *uncurry*.

This is the *basic observation* that suggests to us how to define a monad $St\ b = (s \to (s, b))$ for some state s in a very natural way. The bind $\gg\!=$ operation will essentially be composition of f above.

State Monad

Suppose we have some given *State* type. We can then define this datatype:

```
data St a = St (State → (State, a))
```

I.e. given a previous state, we transform this state into a new state plus some additional output *a*. We can view the state monad as hiding an extra argument and extra result of type *State* during composition.

State Monads

We instantiate *Monad* and provide a binding operation that allows states to be transformed "behind the scene":

```
instance Monad St where
  return x       = St (λs → (s, x))
  (St m0) >>= f  = St (λs0 →
                     let (s1, a1) = m0 s0
                         (St m1)  = f a1
                         (s2, a2) = m1 s1
                     in (s2, a2))
```

- *return* returns the parameter paired with the same state.
- The operator $m \gg\!= f$ is a computation which runs m_0 on the initial state s_0 producing a result a_1 and a new state s_1, then $f :: a \to St\ a$ is applied to to the result a_1, giving a state transformer which is finally applied to s_1 producing the tuple $(s2, a2)$.

State Monads

This would be quite useless unless we had a way to actually read and write to the state. The simplest possible state change is perhaps:

```
tick :: St ()
tick = St (λx → ((), x + 1))
```

In Haskell there are lot of things available for state monads ...

```
class Monad m ⇒ MonadState s m where
  get :: m s
  put :: s → m ()
```


Polymorphic State Monad

The state monad can be made polymorphic by abstracting from the fixed state *State* that we previously used:

```
data St s a = St (s → (s, a))
instance Monad St where    -- same definition as before!
  return x       = St (λs → (s, x))
  (St m0) >>= f  = St (λs0 →
                     let (s1, a1) = m0 s0
                         (St m1)  = f a1
                         (s2, a2) = m1 s1
                     in (s2, a2))
```

Note the partial application of the type constructor *St* in the instance declaration. This works because St has kind $\star \to \star \to \star$, so *St s* has kind $\star \to \star$ ...

IO

Suppose we have these operations that implement an association list:

```
lookup :: a → [(a, b)] → Maybe b
update :: a → b → [(a, b)] → [(a, b)]
exists :: a [(a, b)] → Bool
```

A file system is just an association list mapping file names (strings) to file contents (strings):

```
type State = [(String, String)]
```

Then an extremely simplified *IO* monad is:

```
data IO a = IO (State → (State, a))
```

whose instance in *Monad* is exactly as on the preceding slide, replacing *St* with *IO*.

IO

All that remains is defining the domain-specific operations, such as:

```
readFile :: String → IO (Maybe String)
readFile s = IO (λfs → (fs, lookup s fs))
writeFile :: String → String → IO ()
writeFile s c = IO (λfs → (update s c fs, ()))
fileExists :: String → IO Bool
fileExists s = IO (λfs → (fs, exists s fs))
```

Variations include generating an error when readFile fails instead of using the Maybe type, etc.

A Monadic Evaluator

We can write a new evaluator which uses monads as a structuring principle. All that is needed, is to change the type of *eval* so that *m* is an instace of *Monad*.

```
eval :: Monad m ⇒ Term → m Int
eval (Con x) = return x
eval (Div t u) = do x ← eval t
   y ← eval u
   return (x div y)
```

Next, we have to make a few local changes for each extension of the evaluator that we have in mind, for example to handle raised exceptions. We do *not* have the rewrite the entire evaluator for each such change.

A Monadic Evaluator

```
do x ← eval t
   y ← eval u
   return (x div y)
```

is translated into the following

```
eval t >>= λx → eval u >>= λy → eval u >>= return (x div y)
```

Since lambda abstractions binds has lower precedence than application, this is the same thing as the following after brackets have been inserted:

```
eval t >>= (λx → ((eval u) >>= (λy → ((eval u) >>= return (x div y)))))
```

Note also that the type of *eval* provides some information. The result is *m Int*, which says that *eval* performs a computation that yields an integer as a result.

A Monadic Evaluator: Exceptions

For example, assuming the monad *m* is provided with an extra function $raise :: String \to Exc\ a$, the following local change is all we need:

```
if b ≡ 0
  then raise "divide by zero"
  else return (x div y)
```

In summary we have:

```
eval (Con t u) =
  do {x ← eval t; y ← eval u;
    if b ≡ 0
      then raise "zero divisor"
      else return (x div y)}
```


Evaluating with State

Indeed we can now pass a state around during evaluation, while also update (read and write) the state as we go. We settle for *tick*:

```
evalSt :: Term → St Int
evalSt (Con x) = return x
evalSt (Div t u) = do x ← evalSt u
  y ← evalSt t
  tick
  return (x div y)
```

You can think about how to e.g. perform an efficient operation on a tree datatype by threading a state through the computation, e.g. counting the number of visited nodes.

Summary

Summary

We have seen two prominent techniques for structuring purely functional programs:

- Decompose programs based the values that they consume/produce, using *fold* functions, such as *foldr* and *unfoldr* for lists and *foldBtree* for trees, or other similar structured recursion combinators suitable for the problem at hand, ideally with good algebraic properties.
- Structure a program after the effects they have on their environment, i.e. use monads to build functional programs.

In the next lecture we apply monads:

- Parsing text using recursive descent. Monads gives an algebra of parsers.
- Writing interpreters of languages using monads. This allows languages to “grow”...
- Composing effects, interpreters etc using so-called monad transformers.

Good luck with this weeks homework!