# Lecture 25

Lectured by Prof. Caldwell and scribed by Sunil Kothari

November 20, 2008

## 1 Review

Are there any questions about the HW ?

You can use the constraint set to check your answer - by using the unification algorithm.

Basically, it's look at the proof rules and generate constraints.

## 2 Monads

We mentioned that IO was implemented in Haskell using this very abstract but powerful mechanism.

If you read Chapter 10, it's pretty dense. May be you don't have to read all of Chapter 10. It's very interesting stuff.

Just like you can do higher-order programming in visual basic. The monads will also show up in other languages.

Monads are defined as a type class in Haskell. When you have a type class you basically give the signature of the type operators that any class has to satisfy. But not all classes defined like this are valid monads.

Here's the Monad type class :

```
class Monad m where
  return :: a -> m a
```

Here we are looking at higher-order feature - m maps types to types. There are certain other features in Haskell type classes. All other functional languages have this small imperative feature for IO.

```
class Monad m where
  return :: a -> m a
```

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

bind is used as $p >>= q$. For example, consider the following expression in Haskell

```
getChar >>= (\c1 -> getchar >>=\c2 -> return <c1,c2>)
```

The type of the above expression is:

```
Hugs> :t  (getChar >>= (\c1 -> getChar >>= (\c2 -> return (c1,c2) )))
getChar >>= (\c1 -> getChar >>= (\c2 -> return (c1,c2))) :: IO (Char,Char)
Hugs>
```

The expression gets evaluated and waits for two char input from the user and returns a pair.

```
Hugs>  getChar >>= (\c1 -> getChar >>= (\c2 -> return (c1,c2) ))
gh
Hugs>
```

Remember we also had the do notation, so we can write the above as

```
do c1 <- getChar
   c2 <- getChar
   return <c1,c2>
```

Whenever you declared something an instance of monad you also get do notation.

Let's look at some other things which are monads.

There's that algebraic relationship with monads. If you ever want your own monads, you will have to ensure it satisfies the monad laws.

1. $p >>=$ return $= p$.
   For example, putChar $'a' >>=$ return = putChar $a$.
2. (return $e$) $>>= q = q\ e$
   For example, (return $'a'$) $>>=$ putchar = putChar $'a'$.
3. $(p >>= q) >>= r = p >>= s$ where, $s\ x = q\ x >>= r$. Another way to write this $(p >>= q) >>= r = p >>= (\backslash x- > q\ x >>= r)$

Laws 1 and 2 say that return is a right and left identity for bind.
IO monads are something which do something but it's all within. Whereas, lists live in the outside world.

```
all_pairs m n = do x <- m
                   y <- n
                   return (x,y)
```

We can instantiate lists as monads as follows:

```
instance Monad [] where
```

There is a little bit of magic going on here. As a type constructor for [] :: $Type \rightarrow Type$ so [] $a \rightsquigarrow [a]$

```
instance Monad [] where
   return x = [x]
   m >>= f  = concatMap f m
```

There are a few different ways to write *concatMap*

$$concatMap\ f \quad = \quad foldr\ ((++).f)\ []$$

or

$$\begin{aligned} concatMap\ f\ [] &= [] \\ concatMap\ f\ (x : xs) &= (f\ x) + +\text{concatMap}\ f\ xs \end{aligned}$$

$concatMap(\lambda x \rightarrow [x, x])[1, 2]$
$\rightsquigarrow (d1)$concatMap $d$ [2]
$\rightsquigarrow [1, 1] + +d2 + +$concatMap $d$ []
$\rightsquigarrow [1, 1] + +[2, 2] + +[]$
$\rightsquigarrow [1, 1, 2, 2]$
where, $d = (\lambda x \rightarrow [x, x])$

To show list is a monad we prove the monad laws.

**Theorem 1.** $\forall p : [a].p >>= return = p$

*Proof.* Choose an arb. $p \in [a]$. We must show

$$p >>= return$$

On the left
$\quad$ <<defn.>> $concatMap\ return\ p$
$\quad$ <<lemma below>> $= p$

□

We would like to show $concatMap\ return\ p = p$. So we need a lemma:

**Lemma 1.** $\forall m : [a].concatMap\ return\ m = m$

*Proof.* Proof by induction on the list $m$.

**Case** [ ]. concatMap return [] = [] (by definition of ConcatMap).

**Case(x:xs)** . Assume concatMap return $xs = xs$ We must show
$concatMap\ return\ (x : xs) = x : xs$

On the left
concatMap return $(x : xs)$
$=$ (return $x$) $+ +$concatMap return $xs$
$= x : ([] + + xs)$
$= x : xs$

□

To show the second law:
$\forall e : [a].\forall q : a \to [b].(return\ e) >>= q = q\ e$

*Proof.* Choose arb $e \in a$ and $q \in a \to [b]$ and show $(return\ e) >>= q = q\ e$
On the left
$(return\ e) >>= q$
<<def. of bind>> $\overset{=}{=}$ $concatMap\ q(return\ e)$
<<def. of return>> $\overset{=}{=}$ $concatMap\ q\ [e]$
<<def. of concatMap>> $\overset{=}{=}$ $q\ e + +\ concatMap\ q\ []$
$= q\ e + +\ []$
$= q\ e$

□

So what are we doing when we do these proofs ?
We are verifying that type class of monads needs two functions with proper signature. But will it do the correct thing ? Essentially it's guaranteeing that do notation will work out just fine.

Let's look at the type of bind
$(>>=) :: ma- > (a- > mb)- > mb$

Note that in $p >> q$, $>>$ is a sequencing operator and it means that do $p$ - ignore any result and then do $q$.

$$p >> q = p >>= (\_ \rightarrow q)$$

**Theorem 2.** $\forall p : [a], \forall q : a \rightarrow [b], r : b \rightarrow [c], (p >>= q) >>= r = p >>= (\backslash x \rightarrow qx >>= r)$

*By induction on p.* □

# 3 Functional Parsers

A *parser* is a function essentially from $string \rightarrow tree$.

Remember the terms are given by the following datatype:

```
data  Term = V String | Abs String Term | Ap Term Term
```

Remember, that our idea is to come up with a function *parse parse* $(\backslash x.x)y$ as $(Ap(Abs"x"(V"x"))(V"y"))$

```
newtype Parser = MkP (String -> (Tree, String))
```

If we define the Parser type as above a string eat some of it build a tree and return the string that is left after the build. But that's not sufficient. So we change our definition to

```
data Parser = MkP(String ->[(Tree,String)])
```

We want to parameterize the parser on type of tree. So here's the changed datatype:

```
data Parser a = MkP (String -> [(a,String)])
```

We define a function *apply* as:

```
apply :: Parser a ->String ->[(a,String)]
apply (MkP f) s = f s
```

Note that Bird does monad first and then parsers whereas Hutton does the other way.

```
instance Monad Parse where
   return x = (MkP f)
       where f s = [(x,s)]
   p >>= q  = MkP (\s -> case (apply p s)  of
                                [] -> []
                                [(v,out)] -> apply (f v) out)
```

```
item::Parser Char
item ::= MKp (\i -> case i of
                          []  -> []
                          (x:xs) -> [(x,xs)])


do y <- item
   item
   z<- item
   return (y,z)
```