Lecture 24

Lectured by Prof. Caldwell and scribed by Sunil Kothari

November 18, 2008

1 Review

Has anyone looked at the HW ? It's not that hard.
$\lambda f.\lambda x.\lambda y.f(x,y)$

In our language the expression would be
$(Abs\ "f"\ (Abs\ "x"\ (Abs\ "y"\ (Ap\ (V\ "f")(Pair\ (V\ "x")\ (V\ "y"))))))$

In the HW you are supposed to figure out few of the cases
$\lambda f.\lambda p.spread(p,x,y.f\ x\ y)$

is written as:
$(Abs\ f\ (Abs\ p\ (Spread\ (V"p")(x,y)(Ap(V"f")(V"x")(V"y"))))))$

The typechecker should be able to figure out that f is a function and p is a pair. So what is the type of $f\ x\ y$. So the type is $(a \to b \to c) \to ((a \times b) \to c)$

The one thing is to keep adding these new variables - if you generate any while type checking.

Recall that the term language is:

```
data Term = V string | Abs String Term | Ap Term Term | Spread Term (String, String) Term
```

Next we are headed to functional parsers. To check the type checker that we built and parse $\lambda f.\lambda p.spread(p,x,y.f\ x\ y)$ and turn it into
$(Abs"f"(Abs"p"(Spread(V"p")(x,y)(Ap(Ap(V\ "f")(V\ "x"))(V\ "y")))))$
There is this book by a guy named Mark Jason Dominus - Higher Order Programming in Perl, which we will use later in the course.

If you google the guy's name, the review for the book are great.

2 Functional Parsers

Phil Wadler designed the idea of Monads, which serve as basis for IO in Haskell.

Wadler was able to apply ideas of category theory to functional programming.

What's the issue with IO ?

When you do IO (send a file to printer - its not a function, it has got side effects). What's the functional value that gets returns ? IO is all side-effects. Something happens - but in what sense is this a function ??
This is Chapter 10 of Bird.

The idea of monads is quiet complicated to understand but it is easy to use. The idea is to make a package and side effects are encapsulated in that package. The type of commands (actions) in Haskell is given by IO ().
Recall the type unit "()" in Haskell has one element "()".

Example 1. $foo :: a \rightarrow ()$
$foo\ x = ()$ *where* () *the single value in the type unit.*

Note: the type Int has values $\{0, -1, 2, \ldots\}$. *The type unit* $() = \{()\}$ *has one value.*

What about putChar ?
putChar::Char → IO()

In Hugs and GHC the behavior is quite different

```
Hugs> putChar 'x'
x
Hugs>
```

Here's what GHC does:

```
Prelude> putChar 'x'
xPrelude>
```

IO discussed today is all with respect to stdin and stdout. There are versions of all these command for doing file IO. Let's consider *done* which is a no-op *i.e.* does nothing done:: IO()

```
Hugs> :t (>>)
(>>) :: Monad a => a b -> a c -> a c
Hugs>
```

We want to combine IO() actions $(>>) :: IO() \rightarrow IO() \rightarrow IO()$

```
write:: String -> IO()
write [] = done
write (c:cs) = putChar c >> write cs
```

So *write* "*xyz*" is computed as
putChar $x >>$ *putChar* $y >>$ *putChar* $z >>$ *done*.
One problem with this is it doesn't write a newline at the end.

```
writeln:: String -> IO()
writeln s = write s >> putchar '\n'
```

We can generalize to give type to *getChar* - which reads a single character from stdin - we generalize the IO type to return a character.

```
Hugs> :t getChar
getChar :: IO Char
Hugs>
```

In general, IO α is the type of commands (actions) which return something of type α.

Using this generalization - we generalize *done* -

```
Hugs> :t return
return :: Monad a => b -> a b
```

Then we can define
$done = return()$

Since $return() :: IO()$ so $done :: IO()$

```
Main> :t (>>)
(>>) :: Monad a => a b -> a c -> a c
Main>
```

Then $p >> q$ means do action p of type $IO\alpha$ throw away the return value and then do q of type IO β and return the value of type β.

```
Hugs> getChar >> return ()
x
Hugs>
```

Here we can read a character and can't pass it along. So we want to print the next in character since char is an ordered type.
How do I get the character read by *getChar* anywhere ??

well.. there's another operator. $p >> q$ throws away value returned by p.

We can generalize again. We have a new operator bind
$(>>=) :: IO\alpha \to (\alpha \to IO\beta) \to IO\beta$

```
Main> :t (>>=)
(>>=) :: Monad a => a b -> (b -> a c) -> a c
Main>

Main> getChar >>=putChar
xx
Main>
```

We can write a function which grabs a character from the user and gives the next character.

```
Main> getChar >>= f where f x = putChar (toEnum ((fromEnum x) + 1)::Char)
st
Main>
```

What it the type of f ?

```
Main> :t f
f :: Enum a => a -> IO ()
Main>
```

Here's a function which reads n characters from the user and returns a list.

```
readn 0 = return []
readn (n+1) = getChar >>= q
  where q c = readn n >>= r
     where r cs = return (c:cs)

Main> readn 4
were
Main> :t readn
readn :: Integral a => a -> IO [Char]
Main>
```

Now we write a function which reads one line at a time

```
readln :: IO String
readln = getChar >>= q
```

```
  where q c = if c == '\n'
                then return []
                else readln >>= r
                  where r cs = return (c:cs)

Main> readln
wyoming

Main>
```

All these nested wheres are ugly so we can use Haskell "do notation" for Monad operators *return* and *bind*.

```
readn 0 = return []
readn (n+1) = do c <- getChar
                 cs <- readn n
                 return (c:cs)
```

This can be read as follows: Grab the first character call that c and grab the rest an call as cs and then combine them into a list.
What about *readln* ?

```
readln = do c <- getChar
            if c == '\n'
               then return []
               else do cs <- readln
                       return c:cs
```

So, this do notation is a bit of syntactic sugar. This do notation works for any monads. This turns out that lists are also monads.

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

all_pairs [] n =  []
all_pairs (x:xs) n = map (\x -> (x,y)) n ++ all_pairs xs n
```

So how does the do notation work ?

```
do c <- getChar
   cs <- readn n
   return (c:cs)
```

can be written as

```
readn n (n+1) = do { c <- getChar;cs <- readn n; return (c:cs)}
```

In general, do statements are of the form

$$do\ C; r$$

where, C is a semi colon separated list of commands and r is an expression of type IO β- which is the type of entire do expression.

Each command in the list C takes the form
$x \leftarrow p$
where x is a variable (or a tuple of variables).

If p is of type $IO\alpha$ then we can write p instead of $x \leftarrow p$.

Translation

```
do {r} = r
do {x<- p; C;r} =  p>>= q where q x = do {C;r}


do x <- m
   y <- n
   return (x,y)

   m >>= \x ->(n >>\y return (x,y))
```