Lecture 18

Lectured by Prof. Caldwell and scribed by Sunil Kothari

October 28, 2008

1 Overview

We will build a little language and make an interpreter for it (using inductive datatypes). There's a famous paper by Kerninghan and Ritchie, who were at Bell Labs, would create small languages for chemists and physicists. It happens to be really easy in functional languages.

We will give meaning to the syntax by giving meaning to the constructs using Haskell on the abstract syntax.

We have to worry about parsing and we will look at functional parsers, which map strings into abstract syntax.

The langauges we would look in this lecture are:

1. Calculator + variables + local binding construct + evaluator
2. Logic language + evaluator
3. Lambda calculus - build an evaluator

1.1 A simple calculator

What could be the expressions ?

```
Main> :t Add
Add :: Exp -> Exp -> Exp
Main> N 10
N 10
Main> Add (N 10) (N 20)
Add (N 10) (N 20)
Main>
```

So what is in the Exp type ?
$\{N0, N1, N\ (-1), Add\ N0\ N1, Add\ (Add\ N0\ N1), \ldots\}$
The point is that we have a tree structure. Every piece of syntax has a tree structure underneath. This is an alternate view of inductive data structures. A user of little calculator won't like to type this kind of form.

```
Main> (N 0) ‘Add‘ (N 1)
Add (N 0) (N 1)
Main>
```

We want to write "0+1". But that's a problem for the later. We will be building parsers. That is what is happening in programming languages. Our parser should be able to build a tree or give an indication what is wrong with the string that you typed in.

```
eval (N k) = k
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Mul e1 e2) = (eval e1 ) * (eval e2)


Main> eval (N 0)
0
Main> eval (N 4) ‘Add‘ (N 2)

Main> eval ((N 4) ‘Add‘ (N 2))
6
```

But this is boring and we want to add variables.

```
data Exp = N Int |V String | Add Exp Exp | Mul Exp Exp deriving (Eq,Show)


eval (N k) = k
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Mul e1 e2) = (eval e1 ) * (eval e2)
eval (V x)
```

But what is the meaning of a variable ?
We can include meaning (call here m) in our eval function as

```
eval m (N k) = k
eval m (Add e1 e2) = (eval m e1) + (eval m e2)
eval m (Mul e1 e2) = (eval m e1 ) * (eval m e2)
eval m (V x) = m x
```

What is this m argument gonna do ? What is the type of m ?
Look at m on a variable so m has the type

```
Main> :t eval
eval :: ([Char] -> Int) -> Exp -> Int
```

One way to define m is using a let in Haskell as:

```
Main> let m x = 10 in eval m (V "z")
10
Main> let m x = 11 in eval m (N 11)
11

Main> let m x = if x == "zzzz" then (-10) else 10 in eval m ((N 11) ‘Add‘ (V "zzzz"))
1
```

So we add let to our expression language, and what we want is that
$eval\ m$ (let $x = e1$ in ez) $\leadsto$ $eval\ m'\ ez$
where $m'\ z =$ if $z == x$ then ($eval\ m\ e1$) else $m\ z$

```
Main> Let "x" (N 1) (V "x")
Let "x" (N 1) (V "x")
Main> eval (
z -> 0)(Let "x" (N 1) (V "x"))
1
Main> eval (
z -> 0)(Let "x" (N 1) (Let "x" (N 2) (V "x")))
2
Main> eval (
z -> 0)(Let "x" (N 1) (Let "x" (N 2) (V "z")))
0
Main> eval (
z -> 0)(Let "x" ((V "x") ‘Add‘ (N 1))  (V "x"))
1
Main> eval (
_ -> 0)(Let "x" ((V "x") ‘Add‘ (N 1))  (V "x"))
1
```

Haskell has lexical scoping - the binding is closest to the nearest binder in the expression tree.

```
Main> let x = 1 in let x = x+ 1 in x
ERROR - C stack overflow
```

The example above shows that let by default in Haskell is recursive unlike in other functional languages like OCaml, ML.
In our little language we can do something as follows:

```
Main> eval (_ -> 0)(Let "x" (N 1) (Let "x" (Add (V "x") (N 1))  (V "x")))
2
Main> let x = 1 in let x = x+ 1 in 10
10
Main> let x = [] in let x = 1: x in take 12
take 12
Main> let x = [] in let x = 1: x in take 12 x
[1,1,1,1,1,1,1,1,1,1,1,1]
Main> take 12 [1..]
[1,2,3,4,5,6,7,8,9,10,11,12]
```

1.2 Logic language

The terms in this language are given as:

```
data Prop = FF | Neg Prop | Implies Prop Prop  deriving (Show, Eq)
```

Below are some examples:

```
Main> :t FF
FF :: Prop
Main> (Neg FF)
Neg FF
Main> (Neg FF) `Implies` FF
Implies (Neg FF) FF
Main> :t (Neg FF) `Implies` FF
Implies (Neg FF) FF :: Prop
Main>
```

We can define valuations for propositions as:

```
val FF = False
val (Neg p) =  not (val p)
val (Implies p1 p2) =  not (val p1) || val p2
```

Recall that the truth table for $P1 \Rightarrow P2$ is same as that of $\neg P1 \wedge P2$

P1	P2	¬ P1	¬ P1 ∨ P2	P1 ⇒ P2
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

Table 1: Truth table

val behaves as:

```
Main> :t val
val :: Prop -> Bool
Main> val ((Neg FF) ‘Implies‘ FF)
False
Main> val ((Neg FF) ‘Implies‘ (Neg FF))
True
```

We can add a function that counts falses in a proposition

```
falses FF = 1
falses (Neg p) =  (falses p)
falses (Implies p1 p2) =  (falses p1) +  (falses p2)
```

```
Main> :t val
val :: Prop -> Bool
Main> val ((Neg FF) ‘Implies‘ FF)
False
Main> val ((Neg FF) ‘Implies‘ (Neg FF))
True
```

1.3 Lambda calculus

```
data Lam = X String | Apply Lam Lam | Lambda String Lam deriving (Eq, Show)
```

The reduce function uses capture avoiding substitution but we don't worry about it in our definition of *reduce*. Recall that reduce works as :
$(\backslash x- > x)5 \leadsto x[x := 5]$
$(\backslash x- > x + 1)5 \leadsto (x + 1)[x := 5]$
So the function *reduce* is defined as follows:

```
reduce (X s) =  (X  s)
reduce (Apply (Lambda x m) n) = subst (x,n) m
reduce (Apply m n) = Apply (reduce m) (reduce n)
reduce (Lambda x m) =  (Lambda x m)
```

Here's an example of using *reduce*:

```
Main> reduce (Apply (Lambda "x" (X "x")) (Apply (X "y") (X "y")))
Apply (X "y") (X "y")
```