Lecture 28: Finals Review Lecture

Lectured by Prof. Caldwell and scribed by Sunil Kothari

December 4, 2008

1 Review Lecture

Are there any questions about the HW ?

What we have is an over arching idea, that you can use the inductively defined datatypes to describe abstract syntax for various little languages. Then you can write an evaluator for these languages. Finally we got to the parser for the little languages.

A parser maps strings into abstract syntax - which is some sort of a tree. strings $\overset{\text{parser}}{\Rightarrow}$ Inductively defined datatype/abstract syntax $\overset{\text{evaluate}}{\Rightarrow}$ $\overset{\text{analyse}}{}$ output value.

This is a standard design for building software. You can use any language for this - functional as well as imperative.

We did an instance of this framework.

1. Calculator - HW 15
   (a) evaluator
2. Lambda calculus - HW 16
   (a) evaluator
3. Type Inference - HW 20
   (a) parse a lambda term and then do type inference analysis outputting the type if the term has one.

The thing is defining the intermediate languages is simple - just write down a data type.

Here's a strong possible exam question. You may have to build something in this framework. For example, build an evaluator for a small logic language.

$$
\begin{aligned}
Formula ::= & \ V\ String \\
& | \ F\ True \\
& | \ FF\ False \\
& | \ And\ Formula\ Formula \\
& | \ Or\ Formula\ Formula \\
& | \ Implies\ Formula\ Formula
\end{aligned}
$$

The equivalent grammar is:

$$
\begin{aligned}
Formula ::= & \ identifier \\
& | \text{"("} Formula \text{"\&"}\ Formula \text{")"} \\
& | \text{"("} Formula \text{""}\ Formula \text{")"} \\
& | \text{"("} Formula \text{" => "}\ Formula \text{")"} \\
& | \text{"}T\text{"} \\
& | \text{"}F\text{"}
\end{aligned}
$$

Q: Are we going to cover only the portion after midterm ?
A:Yes, but you are expected to know the material from the first half.
If you get a programming job, you will get to do this framework over and over again. It's pretty standard and its nicely done in functional world. 15 years ago, they were confined to purely academic setting.

We spent the last few weeks on type inference- and something like the same is happening in the Haskell.
- with a richer language.

We will start going through the HWs.

HW 13 was to give structural induction principle for STree.

```
data (Ord a) => Stree a = Null | Fork (Stree a) (Stree a)
```

The induction principle would be
$(P(Null) \wedge \forall t1 : Stree\ a.\forall x : a.\forall t2 : Stree\ a.(P(t1) \wedge P(t2)) => P(Fork\ t1\ x\ t2))$
$\Rightarrow \forall t : Stree\ a.P(t)$

That HW was write map and fold on Stree.

Then we had a lecture on Rose Trees. There might be a question on Rose trees.

```
data Rose a = Node a [Rose a]
```

One interesting thing is that there is no empty tree.

Some examples:

```
 1 (Node 1 [])

  1
 /  \
2    3   (Node a [(Node 2 []),Node 3 [], Node 4 []]
```

sumRose is defined as:

```
sumRose (Node k nodes) = k + foldl (+) 0  (map sumRose nodes)
```

flatten behaves as:

```
   1
 / | \
2  3  4    ---> [1,2,3,4]


     1
    /  \
   2    3  --->  [ 1,2,4,5,3]
  / \
 4   5
```

A short version is to say:

```
flatten (Node k nodes) = k: concatMap flatten nodes
```

But, there is another one:

```
flatten (Node k nodes) =  k: foldr (++) [] (map flatten nodes)
```

These are hard because list itself is an inductively defined data type. So when we write the functions over the rose trees we have to use the functions on the lists.

Lecture 18 and HW 15 go together. These have to do with writing a little evaluator. This had a little let clause, which makes a local binding.

The type of this thing

```
Exp = N Int | ...| Let String Exp Exp
```

$eval\ m\ (Let\ "x"\ (N\ 5)\ (Add\ (V\ "x")\ (V\ "x")))$
$\leadsto eval\ m'\ (Add\ (N\ 5)\ (N\ 5))$
$\leadsto eval\ m'\ (N5) + eval\ m'(N\ 5)$
$\leadsto 5 + 5$
$\leadsto 10$

where $m'z =$ if $z ==$ "x" then $(N\ 5)$ else$m\ z$.

Essentially, we are updating the function m to return $N\ 5$ when it sees x. Here $m : string \to Int$.

Basically, the HW was to extend it with boolean values.

Lecture 19 was going over – you probably wanna read this - how this overloading works in the let case.

In Lecture 19 (goes with HW16), we generalize this idea over to capture avoiding substitution. In the HW, you were supposed to add a spread term.

The abstract syntax for the lambda terms is:

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

The HW was to extend an evaluator to include spread-terms.

Lecture 21 ,lecture 22 and HW 17 were about unification. Remember that we were unifying types.

```
data Type = TyVar String | Arrow Type Type
```

Given two types $t1$ and $t2$, they are unifiable iff there exists a substitution σ such that $\sigma\ t1 = \sigma\ t2$. Remember σ is a substitution, mapping strings to types.

In the HW, you were to extend the unification algorithm to products.

In the HW 18 and Lecture 23 there was type inference. You were given the type inference algorithm and you were to extend to handle spread + pair terms and product types.

We have some context Γ, and we are trying to build the set of constraints. Once we get all the constraints, we use unification to get a substitution and apply that substitution to the type that we started with to get the actual type.

Rule for spread

$$\frac{\Gamma, E1 \vdash M : \alpha \times \beta \qquad \{x : \alpha, y : \beta\} \cup \Gamma, E2 \vdash N : \tau}{\Gamma, E1 \cup E2 \vdash spread(M; x, y.N) : \tau}$$

The type inference algorithm was by induction on the structure of the term. Remember E1 and E2 are propagated down to the conclusion from each one of the hypotheses.

HW 19 was to do build the derivation and then generate constraint sets and there will be a couple of these in the final.
The axiom rule:

$$\frac{}{\Gamma,\ \{\alpha = \tau\} \vdash x : \alpha} \text{ where } x : \tau \in \Gamma \qquad \text{(Var)}$$

The abs rule:

$$\frac{\Gamma \backslash x \cup \{x : \alpha\} \ \vdash \ M : \beta}{\Gamma\ , E \cup \{\tau = \alpha \to \beta\} \vdash \ \lambda x.M \ : \ \tau} \text{ where } \alpha \text{ and } \beta \text{ are fresh} \qquad \text{(Abs)}$$

Example derivation of $\lambda x.x$

$$\frac{\dfrac{}{[x : \alpha], \beta = \alpha \vdash x : \beta}\ \text{(Var)}}{[], \{\beta = \alpha, \tau = \alpha \to \alpha\} \vdash \lambda x.x : \tau}\ \text{(Abs)}$$

The resultant constraint set is $\{\alpha = \beta, \tau = \alpha \to \beta\}$ and the substitution is $\{\alpha := \beta, \tau := \beta \to \beta\}$
You won't be asked to do the unification but certainly some derivations.

Lec 25, Lec 26 were on Monads and parsers.

Monads were defined as:

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

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

(>>=) is a sequencing operator for monads and parsers were an instance of Monads.

When you show something is an instance of monad we get the do notation. The do notation for lists is like a looping structure which goes through each element of the list.
Parsers were defined as:

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

Applying a parser to a string is defined as:

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

Monad instances of parser is defined as:

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

And then we looked at some parsers:

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

  symbol:: String -> Parser String
  symbol xs = token (String xs)

  token:: Parse a -> Parser a
```