Advanced Progamming Languages **Lecture 5**
CS 6110 Spring 2015 **Fri. Jan. 30, 2015**

Lecture 5

Topics

1. Discuss homework problem 2, the topic is subtle because few people have an intuition for it; big mistakes in the past. *But* we have proof methods. In this case, by induction on the term structure - which is given inductively, or on the sign of terms.

   In Problem 2, if we assume $x \neq y$ and x *is not free in* b, then the result is true. At least I think I can prove it.

2. An evaluation function for the λ-terms.

   If you know OCaml or Coq it is fun to write this. You can do that for extra credit. Here we look at one approach and rasie some deep questions.

3. What kind of semantic theory or standard mathematics might help us understand the pure lambda calculus?

   How can we explain $\lambda(x.(ap(x;x)))$ and $\Omega = ap(\lambda(x.ap(x;x));\lambda(x.ap(x;x)))$.

4. Combinators are an even more primitive basis for computation. They can express the λ-calculus and conversely. The computational basis is very elegant but not widely studied.

An evaluation function.
In Lecture 4 we wrote a simple "recursive" evaluator.

```
$eval(l) =$ if $l = \lambda(x.b)$ then $l$
            if $l = ap(f;a)$ then
            if $eval(f) = \lambda(x.b)$
              then $eval(b[a/x])$
              else $abort(msg)$
```

This evaluator could return an open term. We use safe substitution rather than the variable convention. This leads in practice to serious issues with renaming - it is the largest part of the Nuprl evaluator code.

Could we write this as an extension to the Lambda Theory? Just add more equations that include *eval*?

$$eval(\lambda(x.b)) = \lambda(x.b)$$

$$eval(ap(f;a)) = eval(ap(eval(f);a))$$

$$eval(ap(\lambda(x.b);a) = eval(b[a/x])$$

$$eval(eval(t)) = eval(t)$$

$$eval(x) = abort(msg)$$

These equations would provide an evaluator. It would be neater to write the "recursive form". This could be done by adding one more operator for recursion. But it can also be done by building this operator as we show later.

Note, the *eval* equations can result in an endless evaluation sequence of λ-terms because of the recursive nature of *eval*.

Here is how we can express the recursive character in an operator.

Writing the evaluator in a functional form.
Suppose we had an operator fix, a notion from Lisp, with this structure and evaluation rule.

$$fix(\lambda(f.b)) \downarrow b[f(x(\lambda(f.b)))/f]$$

Consider the example which illustrates how we can do recursive functions this way. We start with an informal definition of addition recursively:

$$add(0, y) = y$$
$$add(S(x), y) = S(add(x, y))$$

As a recursive function in an applied λ-calculus the form can be this:

first define $\lambda(f.\lambda(x.\lambda(y.\text{ if } x = 0 \text{ then } y \text{ else } f(x-1, y))))$

then apply fix. To make this more compact, write the function as

$$\lambda(f.\lambda(x.\lambda(y.b(f, x, y))))$$

The recursive version is

$$fix(\lambda(f.\lambda(x.\lambda(y.b(f, x, y)))))$$

What sort of "semantic theory" might be possible for the pure lambda calculus? How can we make sense of self application as allowed in $\lambda(x.ap(x;x))$? This is not allowed in "standard mathematics" and certainly not in set theory. ** Consider this "cardinality" argument.

Suppose V is the set of *values* for the theory. We would need to talk about the function space $V \rightarrow V$, and in classical set theory this set has a higher cardinality than V, e.g.

$|V \to V| > |V|$. We teach this to first year college students. For example, if the values are natural numbers $\mathbb{N}$, then we teach that $\mathbb{N} \to \mathbb{N}$ has a larger cardinality.

Dana Scott created a denotational semantics for λ-cardinality that overcame this difficulty by considering only "continuous" functions $V \to V$. One might try restricting to "computable" functions, but the λ-definable functions are already computable.

Combinators.
Another approach to a function based theory of mathematics was developed by Curry. Instead of λ-terms, he used combinators, another system of computable functions. Its signature feature is the avoidance of bound variables. Here is a taste of that theory.

The idea of combinators goes back to Schönfinkel 1924 in a 12 page article *Über die Bausteine de Math Logik*. The idea was extensively developed by Haskell Curry and his colleagues in a subject called *Combinatory Logic*.

Schönfinkel's idea is that logic is about the relationships among the logical operators. For example

$$\sim P \Rightarrow (P \Rightarrow Q)$$

is not about propositions P and Q denoted by these variables but about the relationship between the logical operators, $\sim$ *and* $\Rightarrow$. He called his approach a *function calculus*, application of functions is the only operation.

This approach was similar to Church's approach to the foundations of mathematics. He based math on an *intensional* notion of functions as rules of computation.

(Both Church and Curry did well in the history of computer science, and Chuch well in logic - especially in inspiring and producing students - Turing, Kleene, Rosser, etc. and in influencing people like McCarthy.)

Curry's function notation is simplicity itself - **no binding!** He used single letter combinators such as these:

$\mathbf{K}xy$	$kxy = x$
$\mathbf{S}xyz$	$Sxyz = xz(yz)$
$\mathbf{I}x$	$Ix = x$
$\mathbf{W}fx$	$wfx = fxx$
$\mathbf{Y}f$	$yf = fyf$
$\mathbf{C}fxy$	y in λ notation is $\lambda(f.\lambda(x.f(ap(x;x)))\lambda(x.f(ap(x;x))))$
$\Phi\ fghx$	$\Phi fghx = f(gx)(hx)$

We can define all the combinators from S and K.

I=SKK	**C=S(BBS)(KK)**
B=S(KS)K	**Φ= B(BS)B**
W=SS(KI)	