Advanced Progamming Languages	**Lecture 4**
CS 6110 Spring 2015	**Wed. Jan. 28, 2015**

Lecture 4

Topics

1. Brief review of capture - OCaml example
2. Barendregt's equational theory $\wedge_\alpha$, Chapter 2, 2.1.4, λ
   (He mentions names: λ-calculus, $\lambda\beta$-calculus, λk-calculus
3. An evaluator for λ-terms - denotational, relationship to set theory
4. Combinators

---

1. Review of capture and substitution

   Our key example can be written in OCaml over the integers as:

   $$ap(\lambda(y.ap(\lambda(x.\lambda(y.b(x,y))); a(y))); c)$$

   which reduces to:

   $$\lambda(y.b(a(c), y))$$

   We can write this numerically in OCaml and you can execute the program.

   $ap(\lambda(y.ap(\lambda(x.\lambda(y.x+y)); y*z)); 2)$

   $(fun\, y \rightarrow (fun\, x \rightarrow fun\, y \rightarrow (x+y))(y*3))\ 2$

   $(int \rightarrow int)$

   apply to 2 then 3 get 9

   $(fun\, x \rightarrow fun\, y \rightarrow (x+y))\ 6$

   $fun\, y \rightarrow (6+y)\ 3$

   $6+3$

   9

   Lecture 2 from CS6110 2012 gives the details of safe substitution. PS1 deals with this topic as well and asks you to write out safe substitution for your account of λ-terms.

2. Lambda Theory

Barendregt presents a small formal *equational theory* of λ-terms based on his syntax. Here are his axioms (page 23, Chapter 2) in a different order. We take M, N, L, Z, to be λ-terms.

Eq 1. Reflexivity: $M = N$

Eq 2. Symmetry: $(M = N) \Rightarrow (N = M)$

Eq 3. Transivity: $M = N,\ N = L \Rightarrow M = L$

Eq Ap. $M = N \Rightarrow MZ = NZ$ equals applied to equals

Ap Eq. $M = N \Rightarrow ZM = ZN$ application to equals

$M = N \Rightarrow \lambda x.M = \lambda x.N$

β $(\lambda x.M)N = M[N/x]$ β-conversion (lazy application)

α $M \equiv_\alpha N$ iff N results from M by a sequence of changes of bound variable. We also write $M =_\alpha N$. This is called alpha equality.

This Lambda Theory treats a weak notion of computational equality, a step by step treatment of computation without regard to whether the computation terminates.

There is an even more syntactic theory that omits the β rule. That is a theory of *structural equality.*

An evaluation function for λ-terms

Lisp, built by McCarthy at MIT, was the first programming language to implement the λ-calculus, defined at Princeton by Church. One of McCarthy's key steps was writing an *evaluator* for the language. The ML language adopted this notion. OCaml has an evaluator. The problem for us is that it executes a typed λ-calculus, so we can't experiment with all expressions such as $\lambda(x.xx)\lambda(x.xx)$, more fully

$$ap(\lambda(x.ap(x;x));\lambda(x.ap(x;x)))$$

Here is a lazy evaluator based on the β-reduction rule:

$ap(\lambda(x.b);a) \downarrow b[a/x]$	Barendregt writes using
$\lambda(x.b)a = b[a/x]$	the variable convention.

This evaluation rule is given the name *lazy evaluation* or *call-by-name* evaluation. It is lazy because we don't bother evaluating the argument a before we substitute it "by name" for x.

Here is the lazy evaluator written recursively:

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

The evaluator must deal with any *closed* λ-*expression*.

$$
\begin{aligned}
eval(l) = {} & \textbf{if } l = \lambda(x.b) \textbf{ then } l \\
& \textbf{if } l = ap(f;a) \\
& \quad \textbf{then if } eval(f) = \lambda(x.b) \\
& \qquad\quad \textbf{then } eval(b[a/x]) \\
& \qquad\quad \textbf{else } \text{abort}
\end{aligned}
$$

This is a recursive function. Can we write it as a λ-term?