Advanced Progamming Languages	**Lecture 9**
CS 6110 Spring 2015	**Mon. Feb. 9, 2015**

Lecture 9

Topics

1. Brief review of combinators for natural numbers, successor, and conditional.
2. Discussion of $\mathbb{N}$, the natural numbers.

Standard	0,	$S(0)$,	$S(S(0))$,...
Combinators	**KI**,	**I**,	**SBI**,...

   These are *number generators*, and we see the same with decimal numbers, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ... We learn how to add 1 at a very young age, $0 + 1 = 1$, $1 + 1 = 2$, $2 + 1 = 3$. What are the natural numbers? We see many kinds of generators: combinators, unary notation, binary, decimal, etc. *The numbers are an abstraction over number generators.* We will examine this topic when we study types.
3. What is the combinator to get recursion, and how is

   $$\begin{cases} add(0, y) = y \\ add(S(x), y) = S(add(x, y)) \end{cases}$$

   defined as a combinator?

   It turns out that we don't need to mimic the definition, we get *addition* as **BS(BB)** because **BS(BB)**$\bar{n}\bar{m} = \overline{n + m}$. *Multiplication* is essentially "bulit in" because of the representation of numbers. Multiplication of n by m is the $(m \cdot n)$ power of a function f, $\mathbf{B}\bar{m}\bar{n}f$

   We can see a general way to handle recursive definitions if we use fix in the λ-calculus to define addition.

   $$fix(\lambda(f.\lambda(x.\lambda(y.\ \underline{\text{if}}\ \ \text{x} = 0\ \ \underline{\text{then}}\ \ \text{y}\ \ \underline{\text{else}}\ \ \text{S(f(x} \dot{-} 1)\text{(y)))))))}$$

   This can be used if we define $x \dot{-} 1$ which is 0 if $x = 0$ and otherwise $x - 1$. This subtraction operator is called "monus". There is a complex combinator to do this. It was a nasty problem to define subtraction both in combinators and λ-calculus. I believe that Rosser solved it.
4. The primitive recursive function formalism in the style of Kleene will serve as a laboratory case of a *subrecursive programming language*. This is the kind of language used in the Coq proof assistant where, unlike most programming languages, the functions

are all total.[1] We will look at *partial functions* just below. These are the standard in languages such as OCaml, Lisp, Scala, Java, Nuprl-PL, etc.

One topic we can study for this PL is the use of environments and *named functions*. We typically want to name our functions as *add*, *mult*, *exp*, etc. rather than using the $fix(\lambda(f.\_\_))$ form. In OCaml we would define *add* roughly as

$$\underline{\text{let}}\ \underline{\text{rec}}\ add\, x\, y = \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ y\ \underline{\text{else}}\ add(x-1, y) + 1$$

When we write *add* as $fix(\lambda(f.\lambda(x.\lambda(y.\text{ if } x = 0 \text{ then } y \text{ else } f(x-1)y))))$, we call it an *annonymous function* - even if we use "*add*" as the bound variable f.

At the end of the lecture is a sequence of ever faster growing primitive recursive functions.

5. Illustrating previously mentioned concepts in the language of primitive recursion - call it PrimRec for now.

   What is the normal "calling convention" or *reduction rule* for PrimRec?

   $$add(mult(2,2), add(20,5))$$

   Do we expect lazy evaluation or call-by-value?

   Note that *equality* has yet another meaning from its use in the λ-equational theory.

   Also note, we could define simultaneously *(mutually) recursive functions*. We look at this later in the next problem set.

   We can also explore *computational complexity* issues in this language. We can define a function to count the number of "recursive calls" needed to compute $add(n, m)$.

6. Kleene's partial recursive functions.

   Kleene showed that by adding one more operator to his primitive recursive formalism, he could define all *general recursive functions* (Herbrand/Gödel recursive functions). This is an example of a topic at the intersection of Theory A and Theory B.

   The μ-operator is used as follows:

   $$f(x_1, ..., x_n) = \mu y(g(x_1, ..., x_n, y) = 0)$$
   $$= \text{the } \textit{least number } y \text{ such that } g(x_1, ..., x_n, y) = 0.$$

   Exercise: Write this definition using fix. If we require that for all x_i there is such a y we get the general *recursive functions*.

[1] We call this language CoqPL.

$a_3(x, y) = exp(x, y)$
$exp(0, y) = 1$ i.e. $y^0 = 1$
$exp(S(x), y) = mult(exp(x, y), y)$

$$\begin{cases} hypexp(0, y) = y \\ hypexp(S(x), y) = exp(hypexp(x, y), y) \end{cases}$$

In general a_{n+1} is given by

$$\begin{cases} a_{n+1}(0, y) = y \\ a_{n+1}(S(x), y) = a_n(a_{n+1}(x, y), y) \end{cases}$$

$\lambda(n, x, y.a_n(x, y))$ is a form of Ackermann's function, known to be non primitive recursive because it "grows too fast".