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 \( \text{KI, I, SBI,...} \)

   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 \( \text{BS(BB)} \) because \( \text{BS(BB)} \bar{n} \bar{m} = \bar{n + m} \). 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 \( f \), \( \text{Bm} \bar{n} f \)

   We can see a general way to handle recursive definitions if we use \( \text{fix} \) in the \( \lambda \)-calculus to define addition.

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

   This can be used if we define \( x - 1 \) 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 \( \lambda \)-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. 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

\begin{verbatim}
let rec add x y = if x = 0 then y else add(x-1, y) + 1
\end{verbatim}

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 anonymous 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?

\begin{verbatim}
dd(\text{mult}(2, 2), \text{add}(20, 5))
\end{verbatim}

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

Note that equality has yet another meaning from its use in the \lambda-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).


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 \( \mu \)-operator is used as follows:

\begin{verbatim}
f(x_1, \ldots, x_n) = \mu y(g(x_1, \ldots, x_n, y) = 0) = \text{the least number } y \text{ such that } g(x_1, \ldots, x_n, y) = 0.
\end{verbatim}

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.
\[ a_3(x, y) = \exp(x, y) \]
\[ \exp(0, y) = 1 \quad \text{i.e. } y^0 = 1 \]
\[ \exp(S(x), y) = \text{mult}(\exp(x, y), y) \]

\[ \begin{aligned} 
\text{hypexp}(0, y) &= y \\
\text{hypexp}(S(x), y) &= \exp(\text{hypexp}(x, y), y) 
\end{aligned} \]

In general \( a_{n+1} \) is given by

\[ \begin{aligned} 
\text{hypexp}(0, y) &= y \\
\text{hypexp}(S(x), y) &= \exp(\text{hypexp}(x, y), y) 
\end{aligned} \]

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