Theory of Computation Assignment 7

November 16, 2005

This assignment is due on Tuesday, November 22, 2005.

1   Primitive Recursive Functions

In lecture I presented five schemas for defining primitive recursive functions. They are as follows:

1. [Zero] There is a constant function zero of every arity.

$$Z^k(x_1, \ldots, x_k) = 0$$

2. [Successor] There is a successor function of arity 1.

$$S(x) = x + 1$$

3. [Projection] There are projection functions for every argument position of every arity.

$$P_i^k(x_1, \ldots, x_k) = x_i \qquad \text{where } k > 0,\ i \leq k$$

4. [Composition (also called substitution)] The composition of the function f of arity k with functions $g_1, \ldots g_k$, each of arity l, defines a $f \circ_k^l [g_1 \ldots g_k]$ of arity l satisfying:

$$f \circ_k^l [g_1 \ldots g_k](x_1, \ldots, x_l) = f(g_1(x_1, \ldots, x_l), \ldots, g_k(x_1, \ldots, x_l))$$

5. [Primitive Recursion] The arity k function defined by primitive recursion from a function g of arity $k-1$ and a function h of arity $k+1$ is indicated $\mathrm{PR}^k[g, h]$. It satisfies:

$$
\begin{aligned}
\mathrm{PR}^k[g, h](0, x_2, \ldots, x_k) &= g(x_2, \ldots, x_k) \\
\mathrm{PR}^k[g, h](x + 1, x_2, \ldots, x_k) &= h(x, \mathrm{PR}^k[g, h](x, x_2, \ldots, x_k), x_2, \ldots, x_k)
\end{aligned}
$$

In lecture we showed how to define addition by primitive recursion:

$$\mathrm{PR}^2[P_1^1, S \circ_1^3 [P_2^3]]$$

1.1 Exercises

Using primitive recursion define:

1. Multiplication

2. If-then-else (e.g. $\text{ITE}(1, x, y) = x$, $\text{ITE}(0, x, y) = y$)

3. Or

4. And

5. Bounded quantification

   Define the bounded existential and universal quantifiers

   (a) $\text{BEQ}[\mathcal{P}] = \exists x < y.\mathcal{P}(x)$
   (b) $\text{BUQ}[\mathcal{P}] = \forall x < y.\mathcal{P}(x)$

   Define these as functions of y. My solution uses primitive recursion on y, so the order of arguments is y first, x second. It helps to define if-then-else and "boolean" functions (I use 0 for false and 1 for true) first.

2 Lambda Calculus

2.1 Syntax

Terms in the lambda calculus are either variables, lambda-abstractions, or applications.

Application associates to the left, that is mno is associated $((mn)o)$.

The dot in lambda abstraction is a left parenthesis that extends as far to the right as possible. Hence, $\lambda x.\lambda y.mno$ is $(\lambda x.(\lambda y.((mn)o)))$.

The first rule (α) states formally that variable names do not matter:

$$\lambda x.M = \lambda y.M[y/x] \quad \text{provided } y \text{ does not occur free in } M.$$

The second rule (β) embodies the basic mechanism for function application, in which the formal parameter is replaced by the actual parameter.

$$(\lambda x.M)N = M[N/x]$$

Both rules rely on substitution, which is defined as follows:

$$
\begin{aligned}
z[M/x] &= z \quad \text{if } x \neq z \\
x[M/x] &= M \quad \text{if } x = z \\
(\lambda y.N)[M/x] &= \lambda z.(N[z/y][M/x]) \quad \text{where } z \text{ is a new variable that is distinct from } x \text{ and does not occur in } N \text{ and is not free in } M. \\
(N_1 N_2)[M/x] &= (N_1[M/x])(N_2[M/x])
\end{aligned}
$$

The treatment of variable renaming prevents capture of bound variables. It is sometimes necessary to do all of the renaming suggested by the definition. Often in calculation it is useful to preserve names if they do not cause conflict.

2.2   Calculating with Church numerals

First, note that the Church numerals iterate application of the first argument to the second. Examples are:

$$
\begin{aligned}
0 &\;-\; \lambda s.\lambda z.z \\
1 &\;-\; \lambda s.\lambda z.sz \\
2 &\;-\; \lambda s.\lambda z.s(sz) \\
3 &\;-\; \lambda s.\lambda z.s(s(sz))
\end{aligned}
$$

The successor function is defined:

$$\lambda n.\lambda s.\lambda z.s(n\ s\ z)$$

Note how the successor function acts on the numeral 2:

$$
\begin{aligned}
&(\lambda n.\lambda s.\lambda z.s(nsz))(\lambda s.\lambda z.s(sz)) \\
&(\lambda s'.\lambda z'.s'((\lambda s.\lambda z.s(sz))s'z')) \\
&(\lambda s'.\lambda z'.s'(\lambda z.s'(s'z))z')) \\
&(\lambda s'.\lambda z'.s'(s'(s'z')))
\end{aligned}
$$

When programming with Church numerals it is important to think of each numeral as capable of driving a loop the iterates that number of times.

In the λ-calculus values are encoded by the control structures that analyze them. Booleans, thus, are represented by the equivalent of if-then-else:

$$true = \lambda t.\lambda f.t \qquad false = \lambda t.\lambda f.f$$

2.3   Exercises

1. Using the representation of Booleans given above define:

   (a) and

   (b) or

   (c) not

2. Use Church numerals to define addition and multiplication. See how the number n is represented by a loop that applies its first argument n times to its second argument. In the successor function above the first argument (s) gets applied one more time. That is the essence of the successor function.