Lecture 7 **7.2.2025**

Today's agenda:

If-then-else construct in LC
Recursion in LC

LC has only three constructs:

$$M ::= x \quad | \quad \underset{\text{Abstraction}}{(\lambda x.\ M)} \quad | \quad \underset{\text{application}}{(M\ N)}$$

There are no numbers, data types, if-then-else, ... However, some of these can be easily encoded in LC.

Idea: try to write the meaning of the function using the above syntax.

If then else construct: *if (x > 0) then print x else print x + 1;*

We can generalize the structure as *if b then c1 else c2* where *b* can be either true or false.

Meaning: *if b = true then do c1 else do c2*

So we denote the above construct by IF that has three formal parameters:

b, c1, c2.

If the actual parameter for b is true (false) IF returns c1 (c2);

Can be generalized as: if b is true select the first argument among c1, c2; otherwise select the second argument.

We know that first = $\lambda x.\ \lambda y.\ x$ first M N = M second = $\lambda x.\ \lambda y.\ y$

Let true = first false = second

so first c1 c2 = true c1 c2 = c1 second c1 c2 = false c1 c2 = c2

IF = $\lambda b.\ \lambda c1.\ \lambda c2.\ b\ c1\ c2$

IF true C1 C2 = (($\lambda c1.\ \lambda c2.$ true c1 c2) C1) C2

= ($\lambda c2.$ true C1 c2) C2 = true C1 C2 = C1

Similarly, IF false C1 C2 = ($\lambda c1.\ \lambda c2.$ false c1 c2) C1 C2 = false C1 C2 = C2

So IF can be expressed as a pure lambda term. IF = $\lambda b.\ \lambda c1.\ \lambda c2.\ b\ c1\ c2$

IF = ($\lambda b.$ ($\lambda c1.$ ($\lambda c2.$ ((b c1) c2))))

Recursion is not provided explicitly in the syntax of LC.

In the early days of the development of PLs, some languages did not allow recursion (e.g., FORTRAN).

We need a way to encode recursion in LC. For this we develop the elementary ideas.

Self application sa = (λx. x x), so x is a function that takes x as an argument.

Recall that in the previous lecture we simplified SII to λz. z z which is sa.

What is sa sa?

sa sa = (λx. x x) (λx. x x)

$=_\beta$ (λx. x x) (λx. x x) = sa sa

Thus by beta reduction we get back sa sa.

This term corresponds to an infinite loop in LC, also referred to as a non-terminating reduction denoted by big omega Ω. But this term is useful when we want to define recursive functions.

Advantage of CBN:

(λy. z) N $=_\beta$ z (which is the normal form)

results by CBN and CBV semantics may be different.

What if N is non terminating? Eg, N = sa sa.

CBN will always return z irrespective of N.

But CBV, for N = sa sa, a computation would proceed that is an infinite loop. Thus the normal form is never found, although one exists.

Recursive function:

eg factorial function

factorial 0 =1 [basis]

factorial n = if n=0 then 1 else n* factorial (n-1)

factorial 3 = 3 * factorial 2 = 3 * 2 * factorial 1 = 3*2*1

when we call factorial with n=3 it calls itself with n=2 and so on.

So we get a call structure like g (g (g .....))) .... ---------B------- [this is our goal]

This is the meaning of recursion. See the similarity with sa sa. The same function is applied over and over again. Let us define a lambda term for factorial function.

$f = \lambda n.$ IF (n=0) 1 (n* f (n-1)) but it is recursive

idea: give a non-recursive definition and then *obtain the recursion*

$F = \lambda f. \lambda n.$ IF (n=0) 1 (n* f (n-1)) which is non-recursive but this is not factorial function

How to obtain the recursion?

We need a special lambda term, called Y combinator that was invented by Haskell B. Curry.

The Y combinator denoted by Y

Haskell Brooks Curry (1900-1982)

The term currying is named after him. Haskell PL is named after him. There are other PLs, Brook and Curry, that are also named after him.

$Y = \lambda t. (\lambda x. t (x x)) (\lambda x. t (x x))$ compare any subterm with $sa = \lambda x. x x$

Here sa is modified as: $sa' = (\lambda x. \underline{t} (x x))$ the t is introduced

Now we do sa' sa' and ensure that it is a closed term. So we obtain

$Y = \lambda t. (\lambda x. t (x x)) (\lambda x. t (x x))$

Let us apply Y to t, we get

$Y t = [\lambda t. (\lambda x. t (x x)) (\lambda x. t (x x))] t$

$=_\beta (\lambda x. t (x x)) (\lambda x. t (x x))$ ---A--

$=_\beta t ((\lambda x. t (x x)) (\lambda x. t (x x)))$ by CBN

$= t (Y t)$ by 'A' [now we can observe that Y t is recursive]

$= t (t (t (t ....) ....$ by unwinding compare this with B

We use it to encode recursive functions in LC. Now we claim that the lambda term for factorial function is Y F, i.e., f = Y F $(\lambda t. (\lambda x. t (x x)) (\lambda x. t (x x))) F = F (Y F)$

End of lecture

f = Y F = F (Y F)