Rewriting logic semantics of the λ-calculus

We have already seen how parallel functional programming can be given a concurrent semantics in the rewrite theory RCL described in Lectures 7-8. Since combinatory logic is a machine-friendly notation, for implementation purposes, combinator semantics and, more generally, super-combinator semantics [see reference to the Peyton Jones' book in Lecture 5-6] is quite enough. However, CL is not user-friendly. Therefore, the following is a relevant question:

What is the rewriting logic semantics of the Lamdbda Calculus?

The answer to this question is non-trivial because the λ-calculus hides a lot of complexity behind it deceptively "simple" notation. This has two consequences:

1. It is machine-unfriendly, and therefore makes it difficult and error-prone to implement it directly.
2. To fully specify the \( \lambda \)-calculus as a rewrite theory, the complexities hidden under the \( \lambda \)-calculus notation should be made explicit.

Q: But where are those complexities hidden?
A: under the substitution notation

Recall that we wrote in Lecture 5-6 a the \( \beta \)-reduction rule as:

\[
\beta : (\lambda x. U) V \rightarrow U[x \mapsto V]
\]

that is we substitute any occurrence of \( x \) in \( U \) by \( V \).

In a more standard notation \( \{ x \mapsto V \} \) is often written \( [x := V] \), and the substitution is applied in front of \( [x := V] \), rather than \( [x := V] \), i.e., behind \( U \), i.e.,

we write:

\[
\beta : (\lambda x. U) \rightarrow [x := V] U
\]

So, where is the complexity? It is hidden in the fact that \( U \) may have two types of variables: (i) variable \( y \) bound by some \( \lambda y \), and (ii) unbound = free variables.
This may complicate matters quite a bit, because of the danger of so-called variable capture. Consider, for example, the λ-expression:

\[(\lambda x. (\lambda y. (y \ x)))(\lambda x. (x \ y))\]

which β-reduces to:

\[(\lambda x. (x \ y))(\lambda y. (y \ x))\]

since the variable \(x\) is free [unbound] in \(\lambda y. (y \ x)\), the naive application of the substitution \([x := \lambda x. (x \ y)]\) would yield:

\[(\lambda y. (y \ \lambda x. (x \ y)))\]

But this is wrong, because the occurrence of \(y\) in \(\lambda x. (x \ y)\) has now been bound [i.e., "captured"] by the outer \(\lambda y\). The reason why this is wrong, is that λ-abstractions of terms are considered equal up to renaming of variables, up to the so-called α-equivalence:
\[\alpha : \lambda x. U \equiv \lambda y. [x := y] U \]
if \(y \notin \text{fv}(U)\)

where \(\text{fv}(U)\), the set of free variables occurring in \(U\) has the obvious recursive definition:

\[\text{fv}(X) = X\]
\[\text{fv}(\lambda x. U) = \text{fv}(U) \setminus \{x\}\]
\[\text{fv}(U \ V) = \text{fv}(U) \cup \text{fv}(V)\]

where \(X\) ranges over variables \(X\) is a "meta-variable" and \(U\), \(V\) range over lambda terms \(U\) is a "meta-variable" of sort \(\text{lambda Term}\).

But this means that:

\[\lambda x. (\lambda y. (y \ x)) \equiv_{\alpha} \lambda x. (\lambda z. (z \ x))\]

and we would have obtained the desired result for the \(\alpha\)-equivalent expression application when \(\beta\)-reduced:

\[(\lambda x. (\lambda z. (z \ x)))\lambda x.(x,y) \rightarrow_{\beta} [x := \lambda x.(x,y)] \lambda z.(z \ x) = \]
\[= \lambda z. (z \ \lambda x.(x,y)) \quad \text{where now \(y\) is not captured!}\]
The long and short of it is that, because of bound variables and the problem of variable capture, the seeming innocent notation $[x:=v] U$ is quite tricky and complex. Can we spell it out? Yes!

$[x:=v] x = v$

$[x:=v] y = y$ if $x \neq y$

$[x:=v](U_1, U_2) = ([x:=v] U_1, ([x:=v] U_2))$

$[x:=v](\lambda x. U) = \lambda x. U$

$[x:=v](\lambda y. U) = \lambda y. [x:=v] U$

if $x \neq y$ and $y \notin \text{fv}(v)$ [no capture case]

$[x:=v](\lambda y. U) = \lambda z. [x:=v][y:=z] U$

if $x \neq y$ and $y \in \text{fv}(v) \land \text{fresh}(z)$ [capture avoidance]

where, intuitively, $Z$ is a "fresh" variable never appearing in either the free variable of $U$ or $V$, i.e., more precisely $\{Z\} \cap (\text{fresh}(U) \cup \text{fresh}(V)) = \emptyset$.

So, implementing $\lambda$-calculus substitution application, and the generation of fresh variables, is quite subtle. However, it can be done, giving rise to a rewrite
theory parametric on a choice of a data type for names, as explained in the Appendix A1 of this lecture. Such an appendix also covers the possibility of adding to the lambda calculus the η-rule:

\[ \eta: (\lambda x. u) \lambda x. (u \ x) \rightarrow u \]

if \( x \notin \text{fv}(u) \)

which captures an intution of extensibility: two functional expressions \( F_1 \) and \( F_2 \) are extensionally equal [specify the same function] iff for any argument \( V \) we have:

\[ F_1 \ V \ \equiv_{\alpha} \ F_2 \ V \]

what \( \eta \) captures is the idea that, as functional expressions, \( u \) and \( \lambda x. (u \ x) \) are extensionally equal, provided \( x \notin \text{fv}(u) \), i.e.:

\[ (\forall V) \ (\lambda x. u \ x) \ V \ \equiv_{\alpha} \ u \ V \]

However, adding \( \eta \) is optional: usually it is not added in implementations.