CS 524

Rewriting Logic Semantics of the λ-Calculus

Lecture 14b

José Meseguer

We have already seen how parallel functional programming can be given a concurrent semantics in the rewrite theory $\mathcal{R}_{CL}$ 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 5-6b] is quite enough. However, CL is not user-friendly. Therefore, the following is a relevant question:

> What is the rewriting logic semantics of the Lambda Calculus?

The answer to this question is non-trivial because the λ-calculus hides a lot of complexity behind its 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 λ-calculus as a rewrite theory, the complexities hidden under the λ-calculus notation should be made <u>explicit</u>.

<u>Q</u>: But were are those complexities hidden?
<u>A</u>: under the <u>substitution</u> notation

Recall that we wrote in Lecture 5-6a the <u>β-reduction rule</u> as:

$$\beta : (\lambda x. U)\, V \longrightarrow U\{x \mapsto V\}$$

that is we <u>substitue</u> any occurrence of x in U by V.
In a more <u>standard notation</u> $\{x \mapsto V\}$ is often written $[x := V]$, and the substitution is applied <u>in front of</u> U, i.e., <u>behind</u> U; i.e., [<u>before</u>] U, rather than [<u>after</u>] U, i.e.,

we write:

$$\beta : (\lambda x. U) \longrightarrow [x := V]\, U$$

So, where is the <u>complexity</u>? It is hidden in the fact that U may have two types of variables: (i) variables y <u>bound</u> by some $\lambda y. -$, and (ii) <u>unbound</u> = <u>free</u> variables.

This may complicate matters quite a lot, because of the danger of so-called *variable capture*. Consider, for example, the λ-expression:

$$\left(\lambda x.\left(\lambda y.(y\ x)\right)\right)\lambda x.(x\ y)$$

which β-reduces to:

$$[x := \lambda x.(x\ y)]\left(\lambda y.(y\ x)\right)$$

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*:

$$\alpha : \quad \lambda x . U \equiv \lambda y . \, [x := y] \, U$$

$$\text{if } \; y \notin fv(U)$$

where $fv(U)$, the set of <u>free</u> [unbound] variables occurring in U has the obvious recursive definition:

$$fv(X) = X$$
$$fv(\lambda X . U) = fv(U) \setminus \{X\}$$
$$fv(U \; V) = fv(U) \cup fv(V)$$

$\boxed{\text{of } \cancel{\text{for}} \text{ sort } \underline{\text{Var}}}$

where X ranges over variables [is a "meta variable"] and U, V range over lambda terms [is a "meta-variable" of sort <u>Lambda Term</u>].

But this means that:

$$\lambda x . \underbrace{(\lambda y . (y \; x))} \equiv_{\alpha} \lambda x . \underbrace{(\lambda z . \; (z \; x))}$$

and <u>we would have obtained the desired result</u> for the α-equivalent expression application when β-reduced:

$$(\lambda x . (\lambda z . \; (z \, x))) \, \lambda x . (x, y) \xrightarrow[\beta]{} [x := \lambda x . (x, y)] \; \lambda z . (z \; x) =$$

$$= \lambda z . (z \;\; \lambda x . (x \; y)) \quad \text{where now } y \text{ is } \underline{\text{not}} \; \underline{\text{captured}} \, !$$

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!

$$[X := V]\, X = V$$

$$[X := V]\, Y = Y \quad \text{if} \quad X \neq Y$$

$$[X := V](U_1\ U_2) = ([X := V]\, U_1)\ ([X := V]\, U_2)$$

$$[X := V](\lambda X.\, U) = \lambda X.\, U$$

$$[X := V](\lambda Y.\, U) = \lambda y.\ [X := V]\, U$$

$$\text{if } X \neq Y \wedge Y \notin fv(V) \quad \text{[no capture case]}$$

$$[X := V](\lambda Y.\, U) = \lambda Z.\ [X := V][Y := Z]\, U$$

$$\text{if } X \neq Y \wedge Y \in fv(V) \wedge fresh(Z) \quad \text{[capture avoidance]}$$

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

So, implementing λ-calculus substitution application, and the generation of fresh variables is quite subtle. However, it can be done, giving rise to a rewrite theory <u>parametric</u> on a choice of a data type for names, as explained in the <u>Appendix A 1</u> of this lecture. Such an appendix also covers the possibility of adding to the lambda calculus the η-rule:

$$\eta : \cancel{(\lambda x . U)}\ \lambda x . (U\, x) \to U \quad \text{if } x \notin fv(U)$$

which captures an intuition of <u>extensionality</u>: two functional expressions F_1 and F_2 are <u>extensionally equal</u> [specify the ~~...~~ <u>same function</u>] iff for any argument V we have:

$$F_1\, V \equiv_\alpha F_2\, V$$

what η captures is the idea that, as functional expressions, U and $\lambda x . (U\, x)$ are <u>extensionally equal</u>, provided $x \notin fv(U)$, i.e.,

$$(\forall V)\ (\lambda x . U\, x)\, V \equiv_\alpha U\, V$$

However, adding η is <u>optional</u>: usually is <u>not</u> added in implementations.