LiU-FP2016: Lecture 4

The Untyped λ-calculus: Operational Semantics and Reduction Orders

Henrik Nilsson

University of Nottingham, UK

Name Capture

Recall that

$$[x \mapsto t]F$$

means "substitute t for free occurrences of x in F.

$$[x \mapsto y](\lambda x.x) =$$

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

Substitution Caveats

We have seen that there are some caveats with substitution:

- Must only substitute for ***free*** variables:

$$[x \mapsto t](\lambda x.x) \neq \lambda x.t$$

- Must avoid ***name capture***:

$$[x \mapsto y](\lambda y.x) \neq \lambda y.y$$

"Substitution" almost always means ***capture-avoiding substitution***.

Capture-Avoiding Substitution (1)

$$
\begin{array}{rcl}
[x \mapsto s]y & = & \begin{cases} s, & \text{if } x \equiv y \\ y, & \text{if } x \not\equiv y \end{cases} \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) \\
[x \mapsto s](\lambda y.t) & = & \begin{cases} \lambda y.t, & \text{if } x \equiv y \\ \lambda y.[x \mapsto s]t, & \text{if } x \not\equiv y \wedge y \notin \mathrm{FV}(s) \\ \lambda z.[x \mapsto s]([y \mapsto z]t), & \text{if } x \not\equiv y \wedge y \in \mathrm{FV}(s), \\ & \quad \text{where } z \text{ is fresh} \end{cases}
\end{array}
$$

where s, t and indexed variants denote lambda-terms; x, y, and z denote variables; $\mathrm{FV}(t)$ denotes the free variables of term t; and $\equiv$ denotes syntactic equality.

Capture-Avoiding Substitution (2)

The condition “z is fresh” can be relaxed:

$$z \not\equiv x \wedge z \notin FV(s) \wedge z \notin FV(t)$$

is enough.

Capture-Avoiding Substitution (3)

A slight variation:

$$
\begin{array}{rcl}
[x \mapsto s]y & = & \begin{cases} s, & \text{if } x \equiv y \\ y, & \text{if } x \not\equiv y \end{cases} \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) \\
[x \mapsto s](\lambda y.t) & = & \begin{cases} \lambda y.t, & \text{if } x \equiv y \\ \lambda y.[x \mapsto s]t, & \text{if } x \not\equiv y \wedge y \notin \mathrm{FV}(s) \\ [x \mapsto s](\lambda z.[y \mapsto z]t), & \text{if } x \not\equiv y \wedge y \in \mathrm{FV}(s), \\ & \quad \text{where } z \notin \mathrm{FV}(s) \\ & \qquad \wedge\ z \notin \mathrm{FV}(t) \end{cases}
\end{array}
$$

Homework: Why isn’t $z \not\equiv x$ needed in this case?

α- and η-conversion

- ***Renaming bound variables*** is known as ***α-conversion***. E.g.

  $$(\lambda x.x) \underset{\alpha}{\leftrightarrow} (\lambda y.y)$$

- Note that $(\lambda x.F\ x)\ G \underset{\beta}{\rightarrow} F\ G$ if x ***not free*** in F.

  This justifies ***η-conversion***:

  $$\lambda x.F\ x \underset{\eta}{\leftrightarrow} F \quad \text{if } x \notin \mathrm{FV}(F)$$

Capture-Avoiding Substitution (4)

If we adopt the convention that terms that differ only in the names of bound variables are interchangeable in all contexts, then the following *partial* definition can be used as long as it is understood that an α-conversion has to be carried out if no case applies:

$$
\begin{array}{rcl}
[x \mapsto s]y & = & \begin{cases} s, & \text{if } x \equiv y \\ y, & \text{if } x \not\equiv y \end{cases} \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_1) \\
[x \mapsto s](\lambda y.t) & = & \lambda y.[x \mapsto s]t, \quad \text{if } x \not\equiv y \wedge y \notin \mathrm{FV}(s)
\end{array}
$$

Op. Semantics: Call-By-Value (1)

Abstract syntax:

t	$\to$		*terms:*
		x	*variable*
	$\mid$	$\lambda x.t$	*abstraction*
	$\mid$	$t\ t$	*application*

Values:

v	$\to$		*values:*
		$\lambda x.t$	*abstraction value*

Op. Semantics: Call-By-Value (2)

Call-by-value operational semantics:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-APP2)}$$

$$(\lambda x.t)\ v \longrightarrow [x \mapsto v]t \quad \text{(E-APPABS)}$$

Op. Semantics: Full β-reduction

Operational semantics for full β-reduction (non-deterministic). Syntax as before, but the syntactic category of values not used:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'} \qquad \text{(E-APP2)}$$

$$(\lambda x.t_1)\ t_2 \longrightarrow [x \mapsto t_2]t_1 \quad \text{(E-APPABS)}$$

Op. Semantics: Normal-Order

Normal-order operational semantics is somewhat awkward to specify. Like full β-reduction, except left-most, outermost redex first.

Op. Semantics: Call-By-Name

Call-by-name like normal order, but no evaluation under λ:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-APP1)}$$

$$(\lambda x.t_1)\ t_2 \longrightarrow [x \mapsto t_2]t_1 \quad \text{(E-APPABS)}$$

Note: Argument not evaluated “prior to call”!

Call-By-Value vs. Call-By-Name (1)

Exercises:

1. Evaluate the following term both by call-by-name and call-by-value:

$$(\lambda x.\lambda y.y)\ ((\lambda z.z\ z)\ (\lambda z.z\ z))$$

2. For some term t and some value v, suppose $t \overset{*}{\underset{\beta}{\rightarrow}} v$ in, say 100 steps. Consider $(\lambda x.x\ x)\ t$ under both call-by-value and call-by-name. How many steps of evaluation in the two cases? (Roughly)

Call-By-Value vs. Call-By-Name (2)

Questions:

- Do we get the same result (modulo termination issues) regardless of evaluation order?
- Which order is “better”?

The Church-Rosser Theorems (1)

Church-Rosser Theorem I:

For all λ-calculus terms t, t_1, and t_2 such that $t \overset{*}{\underset{\beta}{\rightarrow}} t_1$ and $t \overset{*}{\underset{\beta}{\rightarrow}} t_2$, there exists a term t_3 such that $t_1 \overset{*}{\underset{\beta}{\rightarrow}} t_3$ and $t_2 \overset{*}{\underset{\beta}{\rightarrow}} t_3$.

That is, β-reduction is ***confluent***.

This is also known as the “diamond property”.

The Church-Rosser Theorems (2)

Church-Rosser Theorem II:

If $t_1 \xrightarrow[\beta]{*} t_2$ and t_2 is a normal form (no redexes), then t_1 will reduce to t_2 under normal-order reduction.

Which Reduction Order? (1)

So, which reduction order is "best"?

- Depends on the application. Sometimes reduction under λ needed, sometimes not.
- Normal-order reduction has the best possible termination properties: if a term has a normal form, normal-order reduction will find it.

Which Reduction Order? (2)

- In terms of reduction steps (fewer is more efficient), none is strictly better than the other. E.g.:
  - Call-by-value may run forever on a term where normal-order would terminate.
  - Normal-order often duplicates redexes (by substitution of reducible expressions for variables), thereby possibly duplicating work, something that call-by-value avoids.

Lazy Evaluation (1)

Lazy evaluation is an ***implementation technique*** that seeks to combine the advantages of the various orders by:

- evaluate on demand only, but
- evaluate any one redex at most once (avoiding duplication of work)

Idea: ***Graph Reduction*** to avoid duplication by explicit sharing of redexes.

Lazy Evaluation (2)

Result: normal-order/call-by-need semantics, but efficiency closer to call-by-value (when call-by-value doesn't do unnecessary work). However, there are inherent implementation overheads of lazy evaluation.

Lazy evaluation is used in languages like Haskell.