G54FOP: Lecture 11 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.

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

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

G54FOP: Lecture 11 - p.2/21

G54FOP: Lecture 11 - p.4/21

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*.

G54FOP: Lecture 11 – p.3/21

G54FOP: Lecture 11 - p.1/21

Capture-Avoiding Substitution (1)

where s, t and indexed variants denote lambda-terms; x, y, and z denote variables; 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:

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

is enough.

Capture-Avoiding Substitution (3)

A slight variation:

$$[x \mapsto s]y = \begin{cases} s, & \text{if } x \equiv y \\ y, & \text{if } x \neq y \end{cases}$$
$$[x \mapsto s](t_1 \ t_2) = ([x \mapsto s]t_1) \ ([x \mapsto s]t_2) \\ \begin{cases} \lambda y.t, & \text{if } x \equiv y \\ \lambda y.[x \mapsto s]t, & \text{if } x \neq y \land y \notin FV(s) \end{cases}$$
$$[x \mapsto s](\lambda y.t) = \begin{cases} x \mapsto s](\lambda z.[y \mapsto z]t), & \text{if } x \neq y \land y \in FV(s), \\ & \text{where } z \notin FV(s) \\ \land z \notin FV(t) \end{cases}$$
Homework: Why isn't $z \neq x$ needed in this case?

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:

$$[x \mapsto s]y = \begin{cases} s, & \text{if } x \equiv y \\ y, & \text{if } x \neq y \end{cases}$$
$$[x \mapsto s](t_1 \ t_2) = ([x \mapsto s]t_1) \ ([x \mapsto s]t_1) \\ [x \mapsto s](\lambda y.t) = \lambda y.[x \mapsto s]t, & \text{if } x \neq y \land y \notin FV(s) \end{cases}$$

G54EOP: Lecture 11 - p.8/21

$\alpha\text{-}$ and $\eta\text{-}\mathrm{conversion}$

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

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

• Note that $(\lambda x.F x) \ G \xrightarrow{\beta} F \ G$ if $x \ not free$ in F.

This justifies η -conversion:

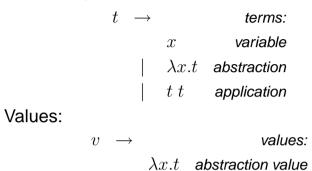
$$\lambda x.F \ x \underset{\eta}{\leftrightarrow} F \quad \text{if} \ x \notin FV(F)$$

G54FOP: Lecture 11 - p.7/21

G54FOP: Lecture 11 - p.5/21

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

Abstract syntax:



Op. Semantics: Full β **-reduction**

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

$$\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2}$$
 (E-APP1)
$$\frac{t_2 \longrightarrow t'_2}{t_1 t_2 \longrightarrow t_1 t'_2}$$
 (E-APP2)

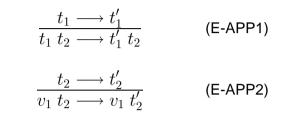
 $(\lambda x.t_1) t_2 \longrightarrow [x \mapsto t_2]t_1$ (E-APPABS)

C54FOP: | edure 11 – p.11/21

G54FOP: Lecture 11 - p.9/21

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

Call-by-value operational semantics:



 $(\lambda x.t) \ v \longrightarrow [x \mapsto v]t$ (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.

G54FOP: Lecture 11 – p.10/21

Op. Semantics: Call-By-Name

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

$$\frac{t_1 \longrightarrow t'_1}{t_1 \ t_2 \longrightarrow t'_1 \ t_2}$$
(E-APP1

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

Note: Argument not evaluated "prior to call"!

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"?

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 $t \stackrel{*}{\xrightarrow{\beta}} v$ 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)

The Church-Rosser Theorems (1)

Church-Rosser Theorem I:

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

That is, β -reduction is *confluent*.

This is also known as the "diamond property".

G54FOP: Lecture 11 – p.15/21

G54FOP: Lecture 11 - p.13/21

G54FOP: Lecture 11 - p.14/21

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.

G54FOP: Lecture 11 - p.18/21

G54FOP: Lecture 11 - p.20/21

Which Reduction Order? (2)

 In terms of reduction steps (fewer is more efficient), none is strictly better than the other. E.g.:

G54FOP: Lecture 11 - p.17/21

G54FOP: Lecture 11 - p.19/21

- 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.

