Lambda Calculus : Semantics

CS3100 Fall 2019

Review

Last time

- Lambda Calculus: Syntax

Today

- Lambda Calculus: Semantics
  - Reductions, Church-Rosser Theorem.

β-reduction

- Lambda Calculus we have been looking so far is **untyped**.
  - No static semantics, only dynamic semantics!
- A term of the form $(\lambda x.\ M)\ N$ is called a **β-redex**.
- The act of evaluating lambda calculus terms is called **β-reduction**.
  - β-reduction replaces $(\lambda x.\ M)\ N$ with $M[N/x]$.
- A term without β-reduxes is said to be in **β-normal form**.

β-reduction, formally

$$\frac{}{(\lambda x.\ M)\ N \rightarrow_\beta M[N/x]} \qquad \frac{M \rightarrow_\beta M'}{\lambda x.\ M \rightarrow_\beta \lambda x.\ M'}$$

$$\frac{M \rightarrow_\beta M'}{M\ N \rightarrow_\beta M'\ N} \qquad \frac{N \rightarrow_\beta N'}{M\ N \rightarrow_\beta M\ N'}$$

Example

$$
\begin{aligned}
& (\lambda\ x\ .\ x\ x)\ \ ((\lambda x.\ y)\ z) \\
\rightarrow_\beta\ & ((\lambda\ x\ .\ y)\ \ z\ )\ \ ((\lambda x.\ y)\ z) \\
\rightarrow_\beta\ & y\ ((\lambda\ x\ .\ y)\ \ z\ ) \\
\rightarrow_\beta\ & y\ y
\end{aligned}
$$

Example

$$
\begin{aligned}
& (\lambda\ x\ .\ x\ x)\ \ ((\lambda x.\ y)\ z) \\
\rightarrow_\beta\ & ((\lambda x.\ y)\ z)\ \ ((\lambda\ x\ .\ y)\ \ z\ ) \\
\rightarrow_\beta\ & ((\lambda\ x\ .\ y)\ \ z\ )\ y \\
\rightarrow_\beta\ & y\ y
\end{aligned}
$$

Example

$$
\begin{aligned}
& (\lambda x.\ x\ x)((\lambda\ x\ .\ y)\ \ z\ ) \\
\rightarrow_\beta\ & (\lambda\ x\ .\ x\ x)\ \ y \\
\rightarrow_\beta\ & y\ y
\end{aligned}
$$

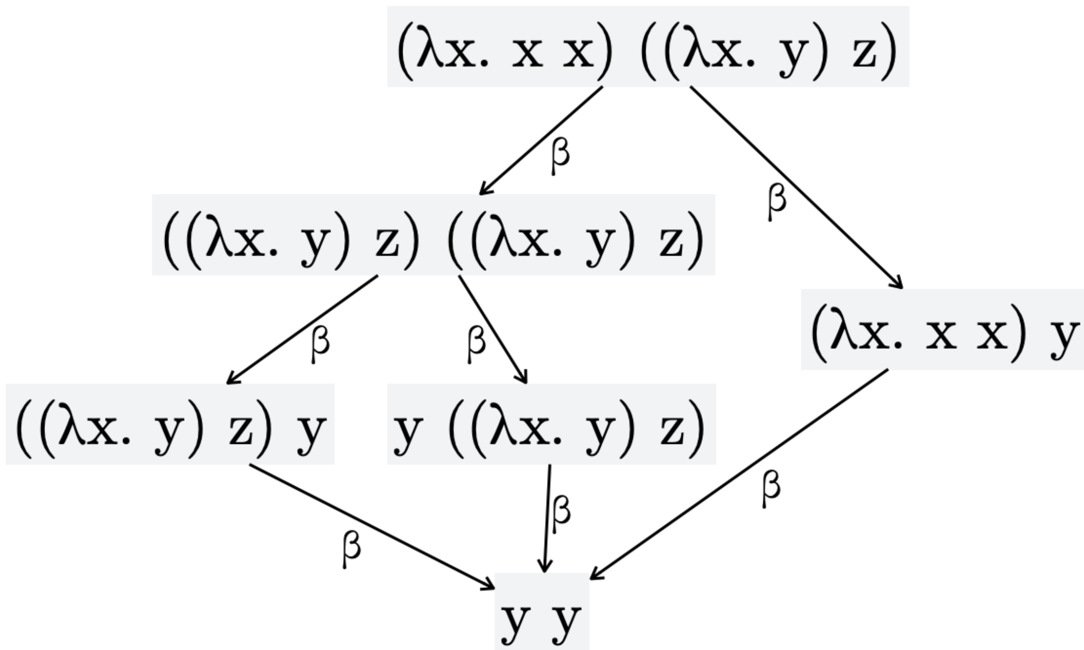
Many steps of β-reduction

$$\frac{M =_\alpha M'}{M \rightarrow_{\beta^*} M'}$$

$$\frac{M \rightarrow_\beta M' \quad M' \rightarrow_{\beta^*} M''}{M \rightarrow_{\beta^*} M''}$$

Church-Rossser Theorem

If $M \rightarrow_{\beta^*} M_1$ and $M \rightarrow_{\beta^*} M_2$ then there exists an M' such that $M_1 \rightarrow_{\beta^*} M'$ and $M_2 \rightarrow_{\beta^*} M'$.



β-normal form

- "β-normal form" ⇒ "contains no reduxes"
- **Theorem** (Uniqueness of β-normal forms). If $M \rightarrow_{\beta^*} N_1$ and $M \rightarrow_{\beta^*} N_2$ and N_1 and N_2 are in β-normal form, then $N_1 =_\alpha N_2$.

- **Proof.** By Church-Rosser, obtain an N such that $N_1 \rightarrow_{\beta^*} N$ and $N_2 \rightarrow_{\beta^*} N$. But N_1 and N_2 are in β-normal form. Hence, $N =_\alpha N_1 =_\alpha N_2$.

β-equivalence

$M_1 =_\beta M_2$ iff there exists an M' such that $M_1 \rightarrow_{\beta^*} M'$ and $M_2 \rightarrow_{\beta^*} M'$.

Possible Non-termination

Some terms do not have a normal form

$$\begin{array}{rl} \Omega & = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \\ & \rightarrow_\beta (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \\ & \rightarrow_\beta (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \\ & \rightarrow_\beta \ldots \end{array}$$

Such terms are said to **diverge**.

Possible Non-termination

Other terms may or may not terminate based on the redux chosen to reduce.

$$\begin{array}{rl} & (\lambda\ x\ .\ y)\ \ ((\lambda x.\, x\ x)\ (\lambda x.\, x\ x)) \\ \rightarrow_\beta & y \end{array}$$

$$\begin{array}{rl} & (\lambda x.\, y)\ ((\lambda\ x\ .\, x\ x)\ \ (\lambda x.\, x\ x)\ ) \\ \rightarrow_\beta & (\lambda x.\, y)\ ((\lambda\ x\ .\, x\ x)\ (\lambda x.\, x\ x)\ ) \\ \rightarrow_\beta & \ldots \end{array}$$

Reduction Strategies

- Several different reduction strategies have been studied for lambda calculus.
- The β reduction we have seen so far is known as **full β-reduction**
  - Any redex in the term can be reduced at any time.

Full β-reduction formally

$$\frac{}{(\lambda x.\, M)\ N \rightarrow_\beta M[N/x]} \qquad \frac{M \rightarrow_\beta M'}{\lambda x.\, M \rightarrow_\beta \lambda x.\, M'}$$

$$\frac{M \rightarrow_\beta M'}{M\ N \rightarrow_\beta M'\ N} \qquad \frac{N \rightarrow_\beta N'}{M\ N \rightarrow_\beta M\ N'}$$

- There may be multiple applicable rules for a term.
  - The reduction is said to be non-deterministic.

Full β-reduction

For example, we can choose to reduce the innermost redex first:

$$\begin{array}{ll} & (\lambda x.\, x)((\lambda x.\, x)\; (\lambda z.\, (\lambda x.\, x)\; z)) \\ =_\alpha & id\; (id\; (\lambda z.\;\; id\; z\;)) \\ \rightarrow_\beta & id\; (\; id\; (\lambda z.\, z)\;) \\ \rightarrow_\beta & id\; (\lambda z.\, z) \\ \rightarrow_\beta & \lambda z.\, z \end{array}$$

Normal order strategy

Reduce leftmost, outermost redex first.

$$\begin{array}{ll} & id\; (id\; (\lambda z.\, id\; z)) \\ \rightarrow_{\hat{\beta}} & id\; (\lambda z.\, id\; z) \\ \rightarrow_{\hat{\beta}} & \lambda z.\;\; id\; z \\ \rightarrow_{\hat{\beta}} & \lambda z.\, z \end{array}$$

In [1]:

```
#use "init.ml"
```

```
Findlib has been successfully loaded. Additional directive
s:
  #require "package";;      to load a package
  #list;;                   to list the available packages
  #camlp4o;;                to load camlp4 (standard synta
x)
  #camlp4r;;                to load camlp4 (revised syntax)
  #predicates "p,q,...";;   to set these predicates
  Topfind.reset();;         to force that packages will be
reloaded
  #thread;;                 to enable threads

val eval_cbv : ?log:bool -> string -> string = <fun>
val eval_cbn : ?log:bool -> string -> string = <fun>
val eval_normal : ?log:bool -> string -> string = <fun>
```

In [2]:

```
eval_normal ~log:true "(\\x.x) ((\\x.x) (\\z.(\\x.x) z))"
```

```
= (λx.x) (λz.(λx.x) z)
= λz.(λx.x) z
= λz.z
```

Out[2]:

```
- : string = "λz.z"
```

Normal order strategy, formally

$$\frac{}{(\lambda x.\, M)\ N \rightarrow_{\hat{\beta}} M[N/x]} \qquad \frac{M \neq \lambda x.\, M_1 \quad M \rightarrow_{\hat{\beta}} M'}{M\ N \rightarrow_{\hat{\beta}} M'\ N}$$

$$\frac{M \neq \lambda x.\, M_1 \quad M \nrightarrow_{\hat{\beta}} \quad N \rightarrow_{\hat{\beta}} N'}{M\ N \rightarrow_{\hat{\beta}} M\ N'} \qquad \frac{M \rightarrow_{\hat{\beta}} M'}{\lambda x.\, M \rightarrow_{\hat{\beta}} \lambda x.\, M'}$$

- Rules are deterministic. (how?)

Call-by-name strategy

- Call-by-name is even more restrictive.
  - Deterministic
  - No reduction under abstraction.

$$\begin{array}{ll} & id\ (id\ (\lambda z.\, id\ z)) \\ \rightarrow_{\beta N} & id\ (\lambda z.\, id\ z) \\ \rightarrow_{\beta N} & \lambda z.\, id\ z \\ \nrightarrow_{\beta N} & \end{array}$$

In [3]:

```
eval_cbn ~log:true "(\\x.x) ((\\x.x) (\\z.(\\x.x) z))"
```

```
= (λx.x) (λz.(λx.x) z)
= λz.(λx.x) z
```

Out[3]:

```
- : string = "λz.(λx.x) z"
```

Call-by-name, formally

$$\frac{}{(\lambda x.\, M)\; N \rightarrow_{\beta N} M[N/x]} \qquad \frac{M \rightarrow_{\beta N} M'}{M\; N \rightarrow_{\beta N} M'\; N}$$

- Arguments not reduced unless they appear on the function position.
  - Is a win if arguments not used.
  - Same reduxes may need to be reduced multiple times.

$$\begin{array}{rl} & (\lambda\ x\ .\ (x\ y)\ (x\ z))\ \ ((\lambda x.\, x)\ a) \\ \rightarrow \beta N & (\ (\lambda x.\, x)\ a\ \ y)\ \ (\ (\lambda x.\, x)\ a\ \ z) \end{array}$$

Call-by-need

- In order to avoid recomputing redexes, use a variant of call-by-name called **call-by-need**
- Idea: Tree reductions ⇒ Graph reductions.
  - Always substitute terms by **reference**
  - Redexes are reduced only once.
- Also known as **lazy evaluation**
  - Used by Haskell and Miranda.
  - Lazy features also present in OCaml, Perl 6.

Call-by-value

Always reduce functions and then arguments before application.

$$
\begin{array}{ll}
 & id\ (\ id\ (\lambda z.\, id\ z)\ ) \\
\rightarrow_{\beta V} & id\ (\lambda z.\, id\ z) \\
\rightarrow_{\beta V} & \lambda z.\, id\ z \\
\not\rightarrow_{\beta V} &
\end{array}
$$

In [4]:

```
eval_cbv ~log:true "(\\x.x) ((\\x.x) (\\z.(\\x.x) z))"
```

Out[4]:

```
- : string = "λz.(λx.x) z"
```

Call-by-value, formally

$$
\frac{M \rightarrow_{\beta V} M'}{M\ N \rightarrow_{\beta V} M'\ N} \qquad \frac{M \not\rightarrow_{\beta V} \qquad N \rightarrow_{\beta V} N'}{M\ N \rightarrow_{\beta V} M\ N'}
$$

$$
\frac{N \not\rightarrow_{\beta V}}{(\lambda x.\, M)\ N \rightarrow_{\beta V} M[N/x]}
$$

- Also known as **strict evaluation**
  - Used by almost all lanugages, including OCaml.

Normalization

Given a term and a reduction strategy, the term is said to normalise under that reduction strategy if reducing that term leads to a β-normal form.

Weak Normalisation: A term is said to weakly normalise under a given reduction strategy if there exists some sequence of reductions leading to a β-normal form.

Strong Normalisation: A term is said to strongly normalise under a given reduction strategy if every reduction leads to a β-normal form.

No distinction between weak and strong if the reduction is **deterministic** (normal order, call-by-name and call-by-value). Why?

Normalization: Examples

- $\Omega = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$ is neither weakly or strongly normalising under full-beta, normal order, call-by-name and call-by-value reduction strategies.
- $(\lambda x.\, y)\ \Omega$ is
  - Weakly normalising but not strongly normalising under full beta reduction.
  - Normalises under normal order and call-by-name.
  - No normal form under call-by-value.

In [5]:

```
eval_normal ~log:true "(\\x.y) ((\\x.x x) (\\x.x x))"
```

```
= (λx.x) (λz.(λx.x) z)
= λz.(λx.x) z
```

Out[5]:

```
- : string = "y"
```

In [6]:

```
eval_cbn ~log:true "(\\x.y) ((\\x.x x) (\\x.x x))"
```

```
= y
= y
```

Out[6]:

```
- : string = "y"
```

In [7]:

```
eval_cbv ~log:true "(\\x.y) ((\\x.x x) (\\x.x x))"
```

```
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
= (λx.y) ((λx.x x) (λx.x x))
```

Normalization: Examples

- $\lambda x. x$ is strongly normalising
  - Every beta-normal form is strongly normalising.
- $(\lambda x. y)\ ((\lambda x. x)\ (\lambda x. x))$ is
  - Strongly normalising under full-beta, normal order, call-by-name and call-by-value.

In [8]:

```
eval_cbv ~log:true "(\\x.y) ((\\x.x) (\\x.x))"
```

```
= (λx.y) (λx.x)
= y
```

Out[8]:

```
- : string = "y"
```

In [9]:

```
eval_cbn ~log:true "(\\x.y) ((\\x.x) (\\x.x))"
```

Out[9]:

```
- : string = "y"
```

In [10]:

```
eval_normal ~log:true "(\\x.y) ((\\x.x) (\\x.x))"
```

```
= y
= y
```

Out[10]:

```
- : string = "y"
```

Extensionality

- Is β-equivalence the best notion of "equality" between λ-terms?
  - We do not have $(\lambda x.\, sin\ x) =_\beta sin$.
  - But, $(\lambda x.\, sin\ x)\ M =_\beta sin\ M$, for any M.

Add η-equivalence

$$\frac{x \neq FV(M)}{\lambda x.\, M\ x =_\eta M}$$

$\beta\eta$-equivalence captures equality of lambda terms nicely.

η-reduction

$$\frac{x \neq FV(M)}{\lambda x.\, M\ x \rightarrow_\eta M}$$

We have applied this rule informally throughout the class in our OCaml examples.

```
List.map (fun x -> shirt_color x) l
```

equivalent to

```
List.map shirt_color l
```

Fin.