Principles of Programming Languages

Module M04: λ-Calculus: Semantics

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

January 24 & 31 February 02, 2022

Table of Contents

Module M04

Partha Pratim Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction
α-Reduction
β-Reduction
η-Reduction
δ-Reduction

Order of
Evaluation
Normal and
Applicative Order

Module M04

Partha Pratim
Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction

α-Reduction
β-Reduction
η-Reduction
δ-Reduction

Order of
Evaluation

Normal and
Applicative Order

Semantics of λ-Expressions

Source:
- λ- Calculus Overview
- Operational Semantics of Pure Functional Languages

- A lambda expression has as its meaning the lambda expression that results after all its function applications (combinations) are carried out
- Evaluating a lambda expression is called **reduction**
- The basic reduction rule involves substituting expressions for free variables in a manner similar to the way that the parameters in a function definition are passed as arguments in a function call
- We start by defining the concepts of free occurrences of variables and the substitution of expressions for variables

Free and Bound Variables

- An occurrence of a variable x is said to be *bound* when it occurs in the body M of an abstraction $\lambda x.\ M$
- We say that λx is a *binder* whose scope is M
- An occurrence of x is *free* if it appears in a position where it is *not bound* by an enclosing abstraction on x
- For example,
  - Occurrences of x in xy and $\lambda y.xy$ are *free*
  - Occurrences of x in $\lambda x.x$ and $\lambda z.\lambda x.\lambda y.x(yz)$ are *bound*
  - In $(\lambda x.x)x$ the first occurrence of x is *bound* and the second is *free*
- In a loose parallel to C functions, consider the *bound* variables as *local* (including *parameters*) and *free* variables as *global* or *non-local*

- In an abstraction, the variable named is referred to as the *bound* variable and the associated λ-expression is the *body* of the abstraction

- In an expression of the form:

$$\lambda v.\ e$$

  occurrences of variable v in expression e are *bound*

- All occurrences of other variables are *free*

- Example:

$$((\lambda x.\ \lambda y.\ (xy))(yw))$$

  ○ x, and y are *bound* in first part
  ○ y, and w are *free* in second part

- $\int_0^1 x^2 dx$; $\int_0^1 A * x^2 dx$
- $\sum_{x=1}^{10} \frac{1}{x}$; $\sum_{x=1}^{10} K * \frac{1}{x}$
- $\lim_{x \to \infty} e^{-x}$; $\lim_{x \to \infty} (M + e^{-x})$
- `int succ(int x) { return x + 1; }`
- $\forall x \in \mathbb{R}, x > 1 \Rightarrow \frac{1}{x} < 1$

- **Definition**: An occurrence of a variable v in a λ-expression is called *bound* if it is within the scope of a λv; otherwise it is called *free*
  - A variable may occur both bound and free in the same λ-expression – for example, in $\lambda x.\ y\ \lambda y.\ y\ x$ the first occurrence of y is free and the other two are bound

- **Definition**: The **set of free variables** in an expression E, denoted by $FV(E)$, is defined as follows:

  [1] $FV(c) = \Phi$ for any constant c
  [2] $FV(x) = \{x\}$ for any variable x
  [3] $FV(E1\ E2) = FV(E1) \cup FV(E2)$
  [4] $FV(\lambda x.\ E) = FV(E) - \{x\}$

- A λ-expression E with no free variables ($FV(E) = \Phi$) is called **closed**

Substitution

Module M04

Partha Pratim
Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction
α-Reduction
β-Reduction
η-Reduction
δ-Reduction

Order of
Evaluation

Normal and
Applicative Order

Substitution

- The notation $E[v \rightarrow E1]$ refers to the λ-expression obtained by replacing each free occurrence of the variable v in E by the λ-expression $E1$

- **Naive Rules of Substitution**

  [1] $v[v \rightarrow E_1] = E_1$ for any variable v
  [2] $x[v \rightarrow E_1] = x$ for any variable $x \neq v$
  [3] $(\lambda v.\ E)[v \rightarrow E_1] = \lambda v.\ (E[v \rightarrow E_1])$
  [4] $(E_{rator}\ E_{rand})[v \rightarrow E_1] = ((E_{rator}[v \rightarrow E_1])(E_{rand}[v \rightarrow E_1]))$

- Does it work?

$$(\lambda y.x)[x \rightarrow (\lambda z.zw)] = \lambda y.\lambda z.zw$$

  YES!

- Consider:

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

  conflicts with a basic understanding that the names of bound variables (that is, parameters) do not matter.

- The identity function is the same whether we write it as $\lambda x.x$ or $\lambda z.z$ or $\lambda fred.fred$.

- If these do not behave the same way under substitution they would not behave the same way under evaluation and that seems wrong

- The mistake is that the substitution should only apply to **free** variables and **not bound** ones

- Here x is bound in the term so we should not substitute it

- That seems to give us what we want:

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

Module M04

Partha Pratim
Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction

α-Reduction
β-Reduction
η-Reduction
δ-Reduction

Order of
Evaluation

Normal and
Applicative Order

Unsafe Substitution: Example

- Again, the naive substitution

$$(\lambda x.\ (mul\ y\ x))[y \to x] \Rightarrow (\lambda x.\ (mul\ x\ x))$$

  is unsafe since the result represents a squaring operation whereas the original lambda expression does not

  ○ A substitution is **valid** or **safe** if no free variable in $E1$ becomes bound as a result of the substitution $E[v \to E1]$
  ○ An invalid substitution involves a **variable capture** or **name clash**

- Correct way would be:

$$(\lambda x.\ (mul\ y\ x))[y \to x] \Rightarrow (\lambda z.\ (mul\ y\ z))[y \to x]$$

$$(\lambda z.\ (mul\ y\ z))[y \to x] \Rightarrow (\lambda z.\ (mul\ x\ z))$$

- Unsafe substitutions change in semantics!

- **Definition**: The **substitution** of an expression for a (*free*) variable in a λ-expression is denoted by $E[v \rightarrow E_1]$ and is defined as follows:

  [1] $v[v \rightarrow E_1] = E_1$ for any variable v

  [2] $x[v \rightarrow E_1] = x$ for any variable $x \neq v$

  [3] $c[v \rightarrow E_1] = c$ for any constant c

  [4] $(E_{rator}\ E_{rand})[v \rightarrow E_1] = ((E_{rator}[v \rightarrow E_1])(E_{rand}[v \rightarrow E_1]))$

  [5] $(\lambda v.\ E)[v \rightarrow E_1] = (\lambda v.\ E)$ // v is not free in E

  [6] $(\lambda x.\ E)[v \rightarrow E_1] = \lambda x.\ (E[v \rightarrow E_1])$ when $x \neq v$ and $x \notin FV(E_1)$

  [7] $(\lambda x.\ E)[v \rightarrow E_1] = \lambda z.\ (E[x \rightarrow z][v \rightarrow E_1])$ when $x \neq v$ and $x \in FV(E_1)$, where $z \neq v$ and $z \notin FV(E\ E_1)$

- In part ([7]), the first substitution $E[x \rightarrow z]$ replaces the bound variable x that will capture the free x's in E_1 by an entirely new bound variable z. Then the intended substitution can be performed safely.

Substitution Example

Module M04

Partha Pratim
Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction
  α-Reduction
  β-Reduction
  η-Reduction
  δ-Reduction

Order of
Evaluation
  Normal and
  Applicative Order

$$(\lambda y.\ (\lambda f.\ f\ x)\ y)\ [x \to f\ y] \qquad \Rightarrow_\alpha$$
$$\lambda z.\ ((\lambda f.\ f\ x)\ z)\ [x \to f\ y] \qquad \Rightarrow \qquad \text{by g) since } y \in FV(f\ y)$$
$$\lambda z.\ ((\lambda f.\ f\ x)\ [x \to f\ y]\ z[x \to f\ y]) \qquad \Rightarrow \qquad \text{by d)}$$
$$\lambda z.\ ((\lambda f.\ f\ x)\ [x \to f\ y]\ z) \qquad \Rightarrow \qquad \text{by b)}$$
$$\lambda z.\ (\lambda g.\ (g\ x)\ [x \to f\ y])\ z \qquad \Rightarrow \qquad \text{by g) since } f \in FV(f\ y)$$
$$\lambda z.\ (\lambda g.\ g\ (f\ y))\ z \qquad \Rightarrow \qquad \text{by d), b), and a)}$$

Rules

[1] $v[v \to E_1] = E_1$ for any variable v

[2] $x[v \to E_1] = x$ for any variable $x \neq v$

[3] $c[v \to E_1] = c$ for any constant c

[4] $(E_{rator}\ E_{rand})[v \to E_1] = ((E_{rator}[v \to E_1])(E_{rand}[v \to E_1]))$

[5] $(\lambda v.\ E)[v \to E_1] = (\lambda v.\ E)$

[6] $(\lambda x.\ E)[v \to E_1] = \lambda x.\ (E[v \to E_1])$ when $x \neq v$ and $x \notin FV(E_1)$

[7] $(\lambda x.\ E)[v \to E_1] = \lambda z.\ (E[x \to z][v \to E_1])$ when $x \neq v$ and $x \in FV(E_1)$, where $z \neq v$ and $z \notin FV(E\ E_1)$

Reduction

- A λ-expression has as its meaning the λ-expression that results after all its function applications (combinations) are carried out
- Evaluating a λ-expression is called **reduction**
- Four rules of reduction
  - α-**Reduction**: Renaming rule
  - β-**Reduction**: Substitution rule
  - η-**Reduction**: Function Equality rule
  - δ-**Reduction**: Pre-defined Constants' rule

Module M04

Partha Pratim Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction
α-Reduction
β-Reduction
η-Reduction
δ-Reduction

Order of
Evaluation
Normal and
Applicative Order

- **Definition**: If v and w are variables and E is a λ-expression,

$$\lambda v.\ E \Rightarrow_\alpha \lambda w.\ E[v \to w]$$

  provided that w does not occur at all in E, which makes the substitution $E[v \to w]]$ safe

- The equivalence of expressions under α-reduction is what makes part g) of the definition of substitution correct

- The α-reduction rule simply allows the changing of bound variables as long as there is no capture of a free variable occurrence

- The two sides of the rule can be thought of as variants of each other, both members of an equivalence class of *congruent* λ-expressions

- The last example contains two α-reductions:

$$\lambda y.\ (\lambda f.\ f\ x)\ y \Rightarrow_\alpha \lambda y.\ ((\lambda f.\ f\ x)\ y)[y \to z] \Rightarrow_\alpha \lambda z.\ (\lambda f.\ f\ x)\ z$$

$$\lambda z.\ (\lambda f.\ f\ x)\ z \Rightarrow_\alpha \lambda z.\ ((\lambda f.\ f\ x)\ z)[f \to g] \Rightarrow_\alpha \lambda z.\ (\lambda g.\ g\ x)\ z$$

- Now that we have a justification of the substitution mechanism, the main simplification rule can be formally defined

β-Reduction

Module M04

Partha Pratim Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction
  α-Reduction
  β-Reduction
  η-Reduction
  δ-Reduction

Order of
Evaluation
  Normal and
  Applicative Order

- **Definition**: If v is a variable and E and E_1 are λ-expressions,

$$(\lambda v.\ E)\ E_1 \Rightarrow_\beta E[v \rightarrow E_1]$$

  provided that the substitution $E[v \rightarrow E_1]$ is carried out according to the rules for a safe substitution

- This β-reduction rule describes the function application rule in which the actual parameter or argument E_1 is *passed to* the function $(\lambda v.\ E)$ by substituting the argument for the formal parameter v in the function

- The left side $(\lambda v.\ E)\ E_1$ of a β-reduction is called a β-**redex** – derived from *reduction expression* – and meaning an expression that can be β-reduced

- β-reduction is the main rule of evaluation in the λ-calculus

- α-reduction makes the substitutions for variables valid

β-Reduction

Module M04

Partha Pratim Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction

α-Reduction
β-Reduction
η-Reduction
δ-Reduction

Order of
Evaluation

Normal and
Applicative Order

- The evaluation of a λ-expression consists of a series of β-reductions, possibly interspersed with α-reductions to change bound variables to avoid confusion
- Take $E \Rightarrow F$ to mean $E \Rightarrow_\beta F$ or $E \Rightarrow_\alpha F$ and let $\Rightarrow^*$ be the *reflexive* and *transitive* closure of $\Rightarrow$
- Hence:
  - For any expression E, $E \Rightarrow^* E$ and
  - For any three expressions, $(E_1 \Rightarrow^* E_2$ and $E_2 \Rightarrow^* E_3)$ implies $E_1 \Rightarrow^* E_3$
- The goal of evaluation in the λ-calculus is to reduce a λ-expression via $\Rightarrow$ until it contains no more β-redexes
- To define an *equality* relation on λ-expressions, we also allow a β-reduction rule to work backward

β-Abstraction

Module M04

Partha Pratim Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction
α-Reduction
β-Reduction
η-Reduction
δ-Reduction

Order of
Evaluation
Normal and
Applicative Order

- **Definition**: Reversing β-reduction produces the β-**abstraction** rule,

$$E[v \to E_1] \Rightarrow_\beta (\lambda v.\ E)\ E_1$$

  and the two rules taken together give β-**conversion**, denoted by $\Leftrightarrow_\beta$

- Therefore $E \Leftrightarrow_\beta F$ if $E \Rightarrow_\beta F$ or $F \Rightarrow_\beta E$

- Take $E \Leftrightarrow F$ to mean $E \Leftrightarrow_\beta F$, $E \Rightarrow_\alpha F$ or $F \Rightarrow_\alpha E$ and let $\Leftrightarrow^*$ be the *reflexive* and *transitive* closure of $\Leftrightarrow$

- Two λ-expressions E and F are **equivalent** or **equal** if $E \Leftrightarrow^* F$

- Reductions (both α and β) are allowed to sub-expressions in a λ-expression by three rules:

  [1] $E_1 \Rightarrow E_2$ implies $E_1\ E \Rightarrow E_2\ E$
  [2] $E_1 \Rightarrow E_2$ implies $E\ E_1 \Rightarrow E\ E_2$
  [3] $E_1 \Rightarrow E_2$ implies $\lambda x.\ E_1 \Rightarrow \lambda x.\ E_2$

- **Definition**: If v is a variable and E is a λ-expression (denoting a function), and v has no free occurrence in E,

$$\lambda v.\ (E\ v) \Rightarrow_\eta E$$

- Example:

$$\lambda x.\ (sqr\ x) \Rightarrow_\eta sqr$$

$$\lambda x.\ (add\ 5\ x) \Rightarrow_\eta (add\ 5)$$

  Note: $(add\ 5\ x)$ abbreviates as $(add\ 5)$

- Take $E \Leftrightarrow_\eta F$ to mean $E \Rightarrow_\eta F$ or $F \Rightarrow_\eta E$

- The requirement that x should have no free occurrences in E is necessary to avoid an invalid reduction such as

$$\lambda x.\,(add\ x\ x) \Rightarrow (add\ x)$$

- This rule fails when E represents some constants; for example, if 5 is a predefined constant numeral, $\lambda x.\,(5\ x)$ and 5 are not equivalent or even related

- η-reduction, justifies an extensional view of functions; that is, *two functions are equal if they produce the same values when given the same arguments*

$$\forall x, f(x) = g(x) \Rightarrow f = g$$

Module M04

Partha Pratim
Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction
  α-Reduction
  β-Reduction
  η-Reduction
  δ-Reduction

Order of
Evaluation
  Normal and
  Applicative Order

- **Extensionality Theorem**: If $F_1\ x \Rightarrow^* E$ and $F_2\ x \Rightarrow^* E$ where $x \notin FV(F_1\ F_2)$, then $F_1 \Leftrightarrow^* F_2$ where $\Leftrightarrow^*$ includes η-reductions.
  Proof:

$$F_1 \Leftrightarrow_\eta \lambda x.\ (F_1\ x) \Leftrightarrow_\eta \lambda x.\ E \Leftrightarrow_\eta \lambda x.\ (F_2\ x) \Leftrightarrow_\eta F_2$$

- The rule is not strictly necessary for reducing λ-expressions and may cause problems in the presence of constants, but included for completeness

δ-Reduction

Module M04

Partha Pratim
Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction
  α-Reduction
  β-Reduction
  η-Reduction
  δ-Reduction

Order of
Evaluation
  Normal and
  Applicative Order

- **Definition**: If the λ-calculus has predefined constants (that is, if it is not pure), rules associated with those predefined values and functions are called *delta* rules:

- Example:

$$(add\ 3\ 5) \Rightarrow_\delta 8 \text{ and } (not\ true) \Rightarrow_\delta false$$

- Example:

$$twice = \lambda f.\ \lambda x.\ f\ (f\ x)$$

$twice\ (\lambda n.\ (add\ n\ 1))\ 5 \Rightarrow_\beta$
$(\lambda f.\ \lambda x.\ (f\ (f\ x)))(\lambda n.\ (add\ n\ 1))\ 5 \Rightarrow_\beta$
$(\lambda x.\ ((\lambda n.\ (add\ n\ 1))((\lambda n.\ (add\ n\ 1))\ x)))\ 5 \Rightarrow_\beta$
$(\lambda n.\ (add\ n\ 1))\ ((\lambda n.\ (add\ n\ 1))\ 5) \Rightarrow_\beta$
$(add\ ((\lambda n.\ (add\ n\ 1))\ 5)\ 1) \Rightarrow_\beta$
$(add\ (add\ 5\ 1)\ 1) \Rightarrow_\delta 7$

Order of Evaluation

- **Call-by-Value (CBV)**
  - C / C++: the argument expression is evaluated, and the resulting value is bound to the corresponding variable in the function (frequently by copying the value into a new memory region)
- **Call-by-Reference (CBR)**
  - C++: a function receives an implicit reference to a variable used as argument, rather than a copy of its value
  - CBR may be simulated in languages that use CBV by making use of references, such as pointers (**Call-by-Address or CBA**)
- **Call-by-Copy-Restore (CBCR) / Value-Result**
  - Fortran (old): a special case of call by reference where the provided reference is unique to the caller (**Copy-in-Copy-out**)
- **Call-by-Name (CBN)**
  - C / C++ Macro: the arguments to a function are not evaluated before the function is called – rather, they are substituted directly into the function body
  - Lazy Evaluation
  - **Call-by-Need**: a memorized variant of CBN where, if the function argument is evaluated, that value is stored for subsequent uses

Evaluation Strategies

```cpp
#include <iostream>
using namespace std;

void f(int a, int b) { a++; b--; return; }            // CBV
void g(int& a, int& b) { a++; b--; return; }          // CBR
void h(int* pa, int* pb) { (*pa)++; (*pb)--; return; } // CBA
#define m_f(a, b) ( a * b )                            // CBN

int main() {
    int x = 3, y = 4, z = 5;
    f(x, y);
    cout << x << " " << y << endl;        // CBV = 3 4
    g(x, y);
    cout << x << " " << y << endl;        // CBR = 4 3
    h(&x, &y);
    cout << x << " " << y << endl;        // CBA = 5 2
    g(z, z);
    cout << z << endl;                    // CBR = 5, CBCR = 6 or 4
    cout << m_f(x + 1, y + 1) << endl;    // CBN = x + y + 1 = 8
}
```

Definition: A λ-expression is in **normal form** if it contains no β-redexes (and no δ-rules in an applied λ calculus), so that it cannot be further reduced using the β-rule or the δ-rule.

An expression in normal form has no more function applications to evaluate

Questions:

[1] Can every λ-expression be reduced to a normal form?

[2] Is there more than one way to reduce a particular λ-expression?

[3] If there is more than one reduction strategy, does each one lead to the same normal form expression?

[4] Is there a reduction strategy that will guarantee that a normal form expression will be produced?

1. Can every λ-expression be reduced to a normal form?

No. Consider:

$(\lambda x.\ x\ x)(\lambda x.\ x\ x) \Rightarrow$
$(\lambda x.\ x\ x)(\lambda x.\ x\ x) \Rightarrow$
$(\lambda x.\ x\ x)(\lambda x.\ x\ x) \Rightarrow$
...

Module M04

Partha Pratim Das

Semantics of λ-Expressions

Free and Bound Variables

Substitution

Reduction

α-Reduction
β-Reduction
η-Reduction
δ-Reduction

Order of Evaluation

Normal and Applicative Order

2. Is there more than one way to reduce a particular λ-expression?

Yes. Consider:

$$(\lambda x.\ \lambda y.\ (add\ y\ ((\lambda z.\ (mul\ x\ z))\ 3)))\ 7\ 5$$

Path 1: OUTERMOST
$(\lambda x.\ \lambda y.\ (add\ y\ ((\lambda z.\ (mul\ x\ z))\ 3)))\ 7\ 5 \Rightarrow_\beta$
$(\lambda y.\ (add\ y\ ((\lambda z.\ (mul\ 7\ z))\ 3)))\ 5 \Rightarrow_\beta$
$(add\ 5\ ((\lambda z.\ (mul\ 7\ z))\ 3)) \Rightarrow_\beta (add\ 5\ (mul\ 7\ 3)) \Rightarrow_\delta (add\ 5\ 21) \Rightarrow_\delta 26$

Path 2: INNERMOST
$(\lambda x.\ \lambda y.\ (add\ y((\lambda z.\ (mul\ x\ z))\ 3)))\ 7\ 5 \Rightarrow_\beta$
$(\lambda x.\ \lambda y.\ (add\ y\ (mul\ x\ 3)))\ 7\ 5 \Rightarrow_\beta (\lambda x.\ (add\ 5\ (mul\ x\ 3)))\ 7 \Rightarrow_\beta$
$(add\ 5\ (mul\ 7\ 3)) \Rightarrow_\delta (add\ 5\ 21) \Rightarrow_\delta 26$

Path 3: MIXED
$(\lambda x.\ \lambda y.\ (add\ y((\lambda z.\ (mul\ x\ z))\ 3)))\ 7\ 5 \Rightarrow_\beta$
$(\lambda x.\ \lambda y.\ (add\ y\ (mul\ x\ 3)))\ 7\ 5 \Rightarrow_\beta (\lambda y.\ (add\ y\ (mul\ 7\ 3)))\ 5 \Rightarrow_\delta$
$(\lambda y.\ (add\ y\ 21))\ 5 \Rightarrow_\beta (add\ 5\ 21) \Rightarrow_\delta 26$

3. If there is more than one reduction strategy, does each one lead to the same normal form expression?

No. Consider:

$$(\lambda y.\ 5)((\lambda x.\ x\ x)(\lambda x.\ x\ x))$$

Path 1:
$(\lambda y.\ 5)((\lambda x.\ x\ x)(\lambda x.\ x\ x)) \Rightarrow 5$

Path 2:
$(\lambda y.\ 5)((\lambda x.\ x\ x)(\lambda x.\ x\ x)) \Rightarrow$
$(\lambda y.\ 5)((\lambda x.\ x\ x)(\lambda x.\ x\ x)) \Rightarrow$
$(\lambda y.\ 5)((\lambda x.\ x\ x)(\lambda x.\ x\ x)) \Rightarrow$
...

4. Is there a reduction strategy that will guarantee that a normal form expression will be produced?

Mathematician Curry proved that *if an expression has a normal form, then it can be found by leftmost reduction*.

A *normal order reduction* can have either of the following outcomes:

[1] It reaches a unique (up to α-conversion) normal form λ-expression

[2] It never terminates

Unfortunately, there is no algorithmic way to determine for an arbitrary λ-expression which of these two outcomes will occur

Module M04

Partha Pratim Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction
α-Reduction
β-Reduction
η-Reduction
δ-Reduction

Order of
Evaluation

Normal and
Applicative Order

Reduction Strategies: Normal and Applicative Order

Two important orders of rewriting:

- **Normal Order** – rewrite the *outermost (leftmost)* occurrence of a function application.
  - ○ *This is equivalent to call by name*
- **Applicative Order** – rewrite the *innermost (leftmost)* occurrence of a function application first
  - ○ *This is equivalent to call by value*

Normal order evaluation always gives the same results as lazy evaluation, but may end up evaluating an expression more times

Module M04

Partha Pratim
Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction
α-Reduction
β-Reduction
η-Reduction
δ-Reduction

Order of
Evaluation
Normal and
Applicative Order

Order of Evaluation

- Example:

$$double\ x\ =\ x + x$$
$$average\ x\ y\ =\ (x + y)/2$$

- Using prefix notation:

$$double\ x\ =\ plus\ x\ x$$
$$average\ x\ y\ =\ divide\ (plus\ x\ y)\ 2$$

- Evaluate:

$$double\ (average\ 2\ 4)$$

- Evaluate:

$$double \ (average \ 2 \ 4)$$

- **Using normal order of evaluation**:

  $double \ (average \ 2 \ 4) \Rightarrow plus \ (average \ 2 \ 4) \ (average \ 2 \ 4) \Rightarrow$
  $plus \ (divide \ (plus \ 2 \ 4) \ 2) \ (average \ 2 \ 4) \Rightarrow plus \ (divide \ 6 \ 2) \ (average \ 2 \ 4) \Rightarrow$
  $plus \ 3 \ (average \ 2 \ 4) \Rightarrow plus \ 3 \ (divide \ (plus \ 2 \ 4) \ 2) \Rightarrow plus \ 3 \ (divide \ 6 \ 2) \Rightarrow$
  $plus \ 3 \ 3 \Rightarrow 6$

  ○ Notice that (*average* 2 4) was evaluated twice ... lazy evaluation would cache the results of the first evaluation

- **Using applicative order of evaluation**:

  $double \ (average \ 2 \ 4) \Rightarrow double \ (divide \ (plus \ 2 \ 4) \ 2) \Rightarrow double \ (divide \ 6 \ 2) \Rightarrow$
  $double \ 3 \Rightarrow plus \ 3 \ 3 \Rightarrow 6$

Module M04

Partha Pratim Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction
  α-Reduction
  β-Reduction
  η-Reduction
  δ-Reduction

Order of
Evaluation

Normal and
Applicative Order

Order of Evaluation

- **Consider**:
  my_if True x y = x
  my_if False x y = y

- **Evaluate**: *my_if (less 3 4) (plus 5 5) (divide 1 0)*

- **Using normal order of evaluation**:

  my_if (less 3 4) (plus 5 5) (divide 1 0) ⇒
  my_if True (plus 5 5) (divide 1 0) ⇒ *(plus 5 5)* ⇒
  10

- **Using applicative order of evaluation**:

  my_if (less 3 4) (plus 5 5) (divide 1 0) ⇒
  my_if True (plus 5 5) (divide 1 0) ⇒
  my_if True 10 (divide 1 0) ⇒
  DIVIDE BY ZERO ERROR

Two important properties of evaluation order:

- *If there is any evaluation order that will terminate and that will not generate an error, normal order evaluation will terminate and will not generate an error*
- *ANY evaluation order that terminates without error will give the same result as any other evaluation order that terminates without error*

Definition: A function f is *strict* in an argument if that argument is *always evaluated* whenever an application of f is evaluated.

If a function is strict in an argument, we can safely evaluate the argument first if we need the value of applying the function.

Lazy Evaluation and Strictness Analysis

Module M04

Partha Pratim
Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction

α-Reduction
β-Reduction
η-Reduction
δ-Reduction

Order of
Evaluation

Normal and
Applicative Order

- We can use lazy evaluation on an ad-hoc basis (for example, for *if*), for all arguments
- For all arguments, for some implementations of functional languages we can improve efficiency using strictness analysis
  - *plus a b* is strict in both arguments
  - *if x y z* is strict in *x*, but not in *y* and *z*
- We can do some analysis and sometimes decide if a user-defined function is strict in some of its arguments:
- Examples:
  - *double x* is strict in *x*
  - *squid n x = if n = 0 then x + 1 else x − n* is strict in *n* and *x*
  - *crab n x = if n = 0 then x + 1 else n* is strict in *n* but not *x*
- If a function is strict in an argument *x*, it is correct to pass *x* by value, even with normal order evaluation semantics
- It is not always decidable whether a function is strict in an argument – if we do not know, pass using lazy evaluation

- **Definition**: A **normal order reduction** always reduces the *leftmost outermost* β-redex (or δ-redex) first
- **Definition**: An **applicative order reduction** always reduces the *leftmost innermost* β-redex (or δ-redex) first
- **Definition**: For any λ-expression of the form $E = ((\lambda x. \ B) \ A)$, we say that β-redex E is outside any β-redex that occurs in B or A and that these are inside E
- A β-redex in a λ-expression is
  - *outermost* if there is no β-redex outside of it
  - *innermost* if there is no β-redex inside of it
- Use AST for detection

β-redexes in $(((\lambda x. \ \lambda y. \ (add \ x \ y)) \ ((\lambda z. \ (succ \ z)) \ 5)) \ ((\lambda w. \ (sqr \ w)) \ 7))$

Module M04

Partha Pratim
Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction
 α-Reduction
 β-Reduction
 η-Reduction
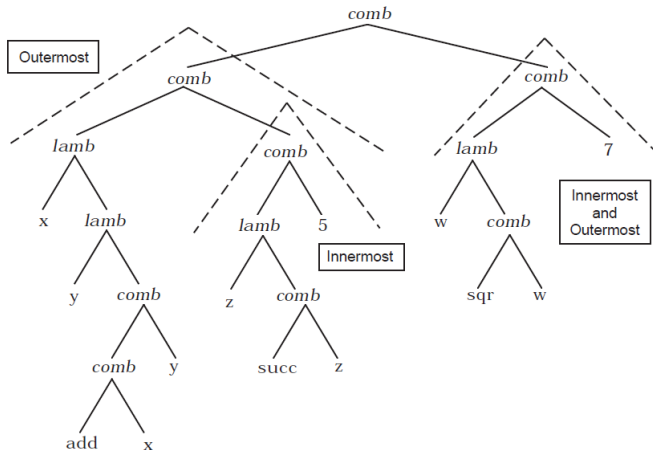 δ-Reduction

Order of
Evaluation
 Normal and
 Applicative Order

Order of Evaluation

- **Applicative Order** (*leftmost innermost*)

  $((\lambda n. (add\ 5\ n))\ 8) \Rightarrow$
  $((\lambda n. (add5\ n))\ 8) \Rightarrow - add5 : N \to N$ is curried $(add5\ 8) \Rightarrow 13$

  - Eager Evaluation
  - Call-by-Value (CBV)
  - Curried functions ($f\ x\ y\ z$) use eager reduction

- **Normal Order** (*leftmost outermost*)

  $((\lambda n. (add\ 5\ n))\ 8) \Rightarrow (add\ 5\ 8) \Rightarrow 13$

  - Lazy Evaluation
  - Call-by-Name (CBN)
  - Function $f(x, y, z)$ use lazy reduction

Module M04

Partha Pratim
Das

Semantics of
λ-Expressions

Free and Bound
Variables

Substitution

Reduction
  α-Reduction
  β-Reduction
  η-Reduction
  δ-Reduction

Order of
Evaluation

Normal and
Applicative Order

```
     ((lambda (x) (+ x x)) (* 2 3))
                /        \
          lazy/          \eager
              /            \
             /              \
  (+ (* 2 3) (* 2 3))   ((lambda (x) (+ x x)) 6)
              \            /
               \          /
                \        /
                 \      /
                  (+ 6 6)
```