Introduction to Functional Programming

Lecture 1 : Lambda Calculus

What is λ-calculus

- Programming language
  - Invented in 1930s, by Alonzo Church and Stephen Cole Kleene
- Model for computation
  - Alan Turing, 1937: Turing machines equal λ-calculus in expressiveness

Overview: λ-calculus as a language

- Syntax
  - How to write a program?
  - Keyword “λ” for defining functions
- Semantics
  - How to describe the executions of a program?
  - Calculation rules called reduction

Syntax

- λ terms or λ expressions:

  (Terms) $M, N ::= x \mid \lambda x.\, M \mid M\ N$

  - Lambda abstraction ($\lambda x.M$): "anonymous" functions

    int f (int x) { return x; } ➔ $\lambda x.\, x$

  - Lambda application ($M\ N$):

    = 3

    int f (int x) { return x; }
    f(3); ➔ $(\lambda x.\, x)\ 3$

Syntax of Lambda Calculus

- **The syntax is simple:**
-
```
<expr> ::=    <constant>
           |  <variable>
           |  (<expr> <expr>)
           |  (λ <variable>. <expr>)
```
  - **<constant> are numbers like** “0” and “1” or predefined functions like “+” , “*”.
  - **<variable>** are names like x,y.
  - 3rd rule means function application (f x)
  - 4th rule is **lambda abstraction**, for building new functions

Syntax

Derivation of (λ x. ((+ 1) x)):

<exp> => (λ <variable>. <expr>)

⇒ (λ x. <expr>)

⇒ (λ x. (<expr> <expr>))

⇒ (λ x. ((<expr> <expr>) <expr>))

⇒ (λ x. ((<constant> <constant>) <expr>))

⇒ (λ x. ((+ 1) <expr>))

⇒ (λ x. ((+ 1) <variable>))

⇒ (λ x. ((+ 1) x))

Calling the Function with Argument

- **Function application**
  - $((\lambda x.\ ((+\ 1)\ x))\ 2) = (\lambda x.\ +\ 1\ x)\ 2$

We call the above function with the argument 2

$(\lambda x.\ +\ 1\ x)\ 2 \Rightarrow (+\ 1\ 2) \Rightarrow 3$

Increase the Readability

- Parentheses are needed for function application to eliminate ambiguity
- According to the syntax description the following:

  $(\lambda x. + 1\ x)$ should be written as:

  $(\lambda x. ((+\ 1)\ x))$

  but to increase readability, we omit the parentheses if no ambiguity results from it

Expressions

- The basic operation of lambda calculus is the application of expressions:

  $((\lambda x. + x\ 1\ )\ 2)$ equals $(+\ 2\ 1)$

  by substituting x for 2 and **throwing** the lambda away

  This equals applying the function with the argument 2

  Historically, this is called a **beta-conversion**

Variables

- A variable **x** in an expression (λx. E) is said to be **bound** by lambda
- The scope of the binding is the expression E
- An unbound variable is said to be **free**
- Example:
  - (λx. + x y)
    - Here, x is bound and y is free

Variables

- $(\lambda x.\ +\ x\ y)\ 2 \Rightarrow (+\ 2\ y)$
  - y is like a non-local reference in this function
- In Scheme:

```
(define y 5)
((lambda (x) (+ x y)) 2)
;Value 7
```

Variables

- The same variable can appear many times in different contexts. Some instances may be bound, others free.
- Example:
  - $(\lambda x. + ((\lambda y. ((\lambda x. * x\ y)\ 2))\ x)\ y)$
  - The 1st x, after the * is bound by a different lambda than the outer x
  - The first instance of y is bound, the second is free

Let's Substitute

$(\lambda x.\ +\ ((\lambda y.\ ((\lambda x.\ *\ x\ y)\ 2))\ x)\ y)$

$\Rightarrow (\lambda x.\ +\ ((\lambda y.\ (*\ 2\ y))\ x)\ y)$

$\Rightarrow (\lambda x.\ +\ ((*\ 2\ x))\ y)$

$\Rightarrow (\lambda x.\ +\ (*\ 2\ x)\ y)$; y is free

- This function adds y to the product of 2 and the argument x, i.e. (2*x)+y.

Applying Beta Conversion

- We may use pass by value or pass by name
  - **Pass by value**:

    $((\lambda x.\ (*\ x\ x))\ (+\ 2\ 3)) \Rightarrow ((\lambda x.\ (*\ x\ x))\ 5) \Rightarrow$

    $(*\ 5\ 5) \Rightarrow 25$
  - **Pass by name, "delayed evaluation", "outermost evaluation", "normal order"**

    $((\lambda x.\ (*\ x\ x))\ (+\ 2\ 3)) \Rightarrow (*\ (+\ 2\ 3)\ (+\ 2\ 3)) \Rightarrow$

    $(*\ 5\ 5) \Rightarrow 25$

Applying Beta Conversion

- The order of **beta** conversions can affect the result. How can this happen?
- Consider this expression:
  - $((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))$
    - Substitution will give you the same expression! (Infinite loop)
  - If this expression is used as an argument for:
  - $((\lambda y.\ 2)\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x)))$ then the result is undefined according to call-by-value but if we use "pass-by-name" the value 2 is returned since the expression doesn't depend on the argument y.

Twice

$(((\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda y.\ (*\ y\ y)))\ 3)$

$= ((\ \lambda x.\ (\lambda y.\ (*\ y\ y))\ ((\lambda y.\ (*\ y\ y))\ x)\ )\ 3)$

$= (\ \lambda x.\ (\lambda y.\ (*\ y\ y))\ ((\lambda y.\ (*\ y\ y))\ x)\ \ 3)$

$= (\ (\lambda y.\ (*\ y\ y))\ ((\lambda y.\ (*\ y\ y))\ 3)\ )$

$= (\lambda y.\ (*\ y\ y))\ ((\lambda y.\ (*\ y\ y))\ 3)$

$= (\lambda y.\ (*\ y\ y))\ ((*\ 3\ 3))$

$= (\lambda y.\ (*\ y\ y))\ (*\ 3\ 3) = (\lambda y.\ (*\ y\ y))\ 9$

$= (*\ 9\ 9) = \underline{81}$