A Simple Introduc

Contents

- Low-level target la
- Very simple syntax
- Simple rules for eval
- Order of applying the rules
- Terminology : "bound" and "free"

- High-level functional languages may be translated by a compiler into the lambda calculus (though there are other implementation routes) ; the λ-calculus might then then translated to an even simpler run-time re
- The λ-calculus is v
- The λ-calculus vi
- Although the λ-calculus was initially conceived as being seque ial implementations (e.g. much work was done in the 1980s to use functi λ-calculus - for parallel processing).

- A program is an expression (like an arithmetic expression) rather than a sequence of instructions
- All a program does is t

  - There are no "side effects" — the only purpose of the program is to retur
    purpose of each part of the program is to return a value

  - In a programming language based on the λ-calculus ht be an
    instruction to the operating system (e.g. to write to a file, or to print to the screen)

Untyped (or "Type-Free") Lambda Calculus Syntax

program : : expression

expression : : x
| expressi e
|

- **Variable :** the na
- **Application :** when one expression follows another, the forme
and the latter is taken to be the argument, thus *exp* s the function *expression*$_1$ applied to the argument *expression*$_2$.
- **Abstraction :** the lambda abstraction $\lambda x.expression$ nt x and with function body *expression*. The name x can be used inside *expression* and represents the value to which the function is applied. We will assume that it is permissible for x to **not** appear inside *expression* (there are different versions of the λ-calculus : some permit this, and some do not).

The type-free λ-calculus can compute anything that is computable. However, the minimal syntax is cumbersome. For example, the numbers 0 and 1 are represented as functions : $\lambda f.\lambda x.x$ $\lambda f.\lambda x.fx$

The syntax is therefore often extended with :

- Constant values su
- Operators such as
  - Initially, all op
- Extra brackets for g
- Types (such as char, bool)
  - But we will not cover the typed lambda calculus[1]
- Lambda abstractions with more than one argument : these can alrea nested abstractions (e.g. $\lambda x.\lambda y.expression$ or $\lambda x.(\lambda y.$) xtended to permit the equivalent $\lambda x_1 x_2.expression$ or in general $\lambda x_1 \ldots x_n.expression$

1. Note that whilst the untyped λ-calculus is Turing-equivalent, the typed λ-calculus typically is not (it depends on the properties of the type system)

Untyped Lambda Calculus — extended syntax

program : : expression

expression : : x
| constant
| operator
| expression expression
| λx . expression
| (*expression*)

Lambda calculus functions

- Functions do NOT have names!
  - Functions ca
- Function argume
  - that can only be u
- Functions can be arguments to other functions (they are
  - that way they can have names when they are passed as arguments to ot
  - and can be used zero or more times inside the other function
  - and it is also possible for a function to return a function as its result

Defining and applying a Lambda calculus function

- To define the (anonymous) function taking one argument (the argument is called x) which adds 1 to x and returns the sum as its result :

- Often simplified to ?

$\lambda x.(+ \quad)$ [*he left*]

$\lambda x.(x + 1)$ [*but we must extend the syntax*]

- To apply the previously defined function to the constant number 3 :

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

Untyped Lambda Calculus — extended syntax with infix operators

program : : expression

expression : : x
| constan
| operato
| expression expression
| expression operator expression
| λx . expression
| (*expression*)

Rules for evaluation

- α-reduction $\lambda x.E \rightarrow \lambda y.E[y/x]$
- β-reduction
- η-reduction

- δ-rules — there is a separate δ-rule for each operator (suc -rule for $+$ says that $3 + 4$ evaluates to 7

- NB : $E[y/x]$ means "for each *free* occurrence of x in his becomes important if E contains another function definition that re-uses the name x for its argument. The embedded function definition *binds* the name x to a new value, thus the enclosing expression E sees all occurrences of x inside the embedded function definition as being *bound* (i.e. not *free*).

Terminology

- Binding — a BINDING links a name to a value. This happens whenever a function is applied.
- Bound and Not Bou

we say that x is BOUND and y is NOT BOUND (alterna FREE). This is because we know what value x refers to - it is the argument to he value of y is unknown (presumably this expression occurs inside the function y, but we don't know how deeply nested we might be inside function defi definitions)

Order of evaluation ("reduction order")

- Normal Order Reduction
  - "leftmost-o
  - guaranteed t
- Other possible red
  - applicative or
  - parallel reduction

All evaluation strategies are guaranteed to give the same result for an expre
That unique result is called the **Normal Form** of the expression.

Lambda Calculus Examples

Example 1 :

$(5 + 3)$ $\rightarrow$ *by* δ *rule for* $+$
8

Example 2 :

$(\lambda x.(x + 3))\ 5$
$(5 + 3)$
8

Example 3 :

$(\lambda y.((\lambda x.(x + y))\ 5))\ 3$ $\rightarrow$ *by*β *reduction*
$(\lambda x.(x + 3))\ 5$ $\rightarrow$ *by*β *reduction*
$(5 + 3)$ $\rightarrow$ *by* δ *rule for* $+$
8

Lambda Calculus Examples

Example 4 :

$(\lambda x.((\lambda x.(x+3))\ x))\ 5 \quad \rightarrow$ *by* α *reduction*
$(\lambda y.((\lambda x.(x+3$
$(\lambda x.(x+3))\ 5$
$(5+3)$
8

Example 5 :

$(\lambda x.(x\ 5))\quad (\lambda x.(x+3)) \quad \rightarrow$ *by* β *reduction*
$((\lambda x.(x+3))\ 5) \quad \rightarrow$ *by* β *reduction*
$(5+3) \quad \rightarrow$ *by* δ *rule for* $+$
8

Lambda Calculus Examples

Example 6 :

$\lambda x.((x\ 5) + (x$
$((\lambda x.(x + 3))\ 5)$
$(5 + 3) + ((\lambda x.($
$(5 + 3) + (4 + 3)$
$8 + (4 + 3)$
$8 + 7$
15

$\rightarrow \quad \delta \qquad +$
$\rightarrow$ *by* δ *rule for*
$\rightarrow$ *by* δ *rule for*

Summary

- Low-level target lla
- Very simple syntax
- Only four rules for evaluation
- Apply the rules in any order (caveat termination)
- "Normal Order" guaranteed to terminate (if termination is possib
- Terminology : "bound" and "free"