Programming Language Semantics: Chapter 01 - Lambda Calcul

Tarek Boutefara – t_boutefara@univ-jijel.dz – Version 0.1, 12-10-2024

Table of Contents

1. Introduction

The λ-Calculus is a mathematical model that was proposed by by Alonzo Church as part of an attempt to propose a solution to the second proposition of Leibniz's ideal:

- To create a universal language in which all possible problems can be stated.
- Find a decision method to solve all the problems stated in the universal language.

λ-Calculus is a Formal System that makes it possible to define the notion of calculable functions. So, in order to understand and manipulate the λ-Calculus, we will begin by introducing the notion of a Formal System. This notion is not only important for understanding the the λ-Calculus; it is also important for modelling and understanding programming paradigms other than procedural programming. Indeed :

- Functional programming: based on the λ-Calculus,
- Logic programming: based on first-order predicate logic.

The λ-Calculus has evolved considerably since it was first proposed. We are going to focus on the notion of typed λ-Calculus. This is a version of the λ-Calculus that supports variable typing.

It is important to point out that we are going to limit ourselves to the computer vision of this very complex notion. of this very complex notion. In other words, we will study the λ-Calculus with the following objectives in mind:

- Understand the origin of functional programming,
- Write a proof for a functional program.

It is also important to note that other concepts related to mathematical without a detailed presentation. We are referring in particular to the theory of the Fixed Point.

2. Formal Systems

2.1. Definition

A Formal System (FS) is an abstract set of rules that manipulates a set of symbols solely syntactically (without considering meaning).

2.2. Formal Systems and modern logic

Formal Systems are proposed in the context of modern or non-standard logic. In our context, the essential difference between standard and non-standard logic lies in the change in the way of reasoning.

In standard (classical) logic, reasoning is performed on knowledge representations by making deductions that take into account the semantics of these representations. In non-standard (modern) logic, reasoning is considered to be independent of semantic content, but only of form.

Thus, reasoning is no longer based on the semantics of propositions or on a semantic deduction, as in the following

example:

- Every man is mortal,
- Socrates is a man,
- Then Socrates is mortal.

Following this vision, proving a theorem involves demonstrating that a particular proposition is always true, regardless of the truth values of its constituent propositions. This is typically done using a proof method.

A method that we have seen during your previous curriculum, is the Truth Table :

- Construct a truth table: List all possible combinations of truth values for the propositions involved.
- Evaluate the compound proposition: Determine the truth value of the compound proposition for each combination of truth values.
- Check for tautology: If the compound proposition is always true, regardless of the truth values of its constituent propositions, then it is a tautology and the theorem is proven.

However, it is transformed into a process of applying rules to obtain new theorems from theorems that have already been proved. Generally, the model proposes axioms (propositions assumed to be true without proof) as starting points for constructing a complete set of theorems.

2.3. Elements of a Formal System

A Formal System is a set of rules for specifying :

- A set of objects
- Elementary statements
- Theorems (true elementary statements)

We are talking about the morphology (form) of the formal system, which is made up of :

- The set of objects,
- The set of elementary statements.

We are talking about the proper theory of the system, which is :

- The set of theorems.

The proper theory can also be defined by the definition of a set of axioms and deduction rules (deductive rules):

- Axioms: elementary statements assumed to be true (without demonstration).
- Deductive rules: for proving theorems from axioms.

The definition of the proper theory of a formal system can be formalised as follows:

$S = (A, R, W)$

Where :

- A: set of axioms,
- R: set of rules,
- W: set of well-formed formulas (deduced by applying R and starting from A).

A rule can be defined as an application of W^n in W where n is the arity of the rule.

Starting from this definition, we can define the set of theorems by:

- $S = A \cup W$

Or by:

- $A^+ = \bigcup A_i, i \geq 0$
- $A_0 = A$
- $A_i = A_{i-1} \cup \left\{ R_j(a_1, a_2, ..., a_n), a_k \in A_{i-1}, R_j \in R \right\}$ (images of A_{i-1} elements generated by applying inference rules).

2.4. Formal System expression classes

Formal system expression classes are categories that group together different types of expressions within a formal system. These classes are often defined based on the syntactic structure and semantic interpretation of the expressions.

- Atomic Expressions :
  - Propositional variables: In propositional logic, these are basic statements that can be true or false.
  - Constants: In mathematical systems, these are fixed values or symbols.
  - Variables: In various systems, these are placeholders for values that can be assigned different values.
- Compound Expressions
  - Formulas: These are expressions that are formed by combining atomic expressions using logical connectives or operators. For example, in propositional logic, formulas can be formed using AND, OR, NOT, IMPLIES, and EQUIVALENT.
  - Terms: In mathematical systems, these are expressions that represent values or objects. For example, in arithmetic, terms can be formed using numbers, variables, and operations like addition, subtraction, multiplication, and division.

Expressions (or statements) can also be divided into :

- Given classes: atoms, operations, axioms, and rules.
- Constructed classes.

This classification allows us to propose an inductive definition of a Formal System. A Formal System can be defined by :

- Certain initial elements: initial specification.
- Certain construction procedures (generation methods): final specification.

2.5. Décidabilité d'un Système Formel

Decidability in the context of formal systems refers to the omputability of a problem within that system. More specifically, it's about whether there exists an algorithm that can determine the truth or falsity of any well-formed formula in the system.

A formal system is decidable if and only if there exists an algorithm t hat can take any formula in the system as input and:

- Terminate: The algorithm always halts after a finite number of steps.
- Give a correct answer: The algorithm correctly determines whether the formula is true or false.

If such an algorithm exists, the system is said to be decidable. Otherwise, it is said to be undecidable.

2.6. Exemples de Systèmes Formels

2.6.1. MUI

We define the following Formal System:

- Alphabet = $\{M, U, I\}$, if an exepression contains other elements, that means this expression is not a formula of this Formal System.
- Well-formed formulas: = $\{M, U, I\}^*$, ie., any string that contains only the elements M, U, and I.
- Proper theory of the Formal System :
  - Axioms = $\{MI\}$, ie. this Formal System defines only one axiom as a starting point to build all other theories.
  - Inference rules : let x and y be two statements
    - $xI \rightarrow xIU$ (Rule 1)
    - $Mx \rightarrow Mxx$ (Rule 2)
    - $xIIIy \rightarrow xUy$ (Rule 3)
    - $xUUy \rightarrow xy$ (Rule 4)

Examples:

(1) MI

(2) MII (by applying Rule 2)

(3) MIIII (by applying Rule 2)

(4) MIIIIU (by applying Rule 1)

(5) MUIU (by applying Rule 3)

Exercise:

Proof the following theorems :

- MIU

- MIUIUIUIU
- MUIIIIU

2.6.2. Formalised predicate calculus

It is possible to design a formal system to handle the formal aspect of the first-order predicate calculus without taking into account the semantic interpretation of these symbols.

The formal system in question can be defined by:

- Alphabet consists of the following elements:
  - a, b, ...,z: propositional variables,
  - $\neg \ \rightarrow$: primitif symbols.
  - (,): parentheses
- The set of well-formed formulas:
  - Every propositional variable is a well-formed formula (Rule 1),
  - If A and B are two WFF then $A \rightarrow B$, $\neg A$ are also WFF (Rule 2),
  - The set of WFF is built by applying rules 1 and 2.
- Axioms :
  - Ax1 : $(A \rightarrow (B \rightarrow A))$
  - Ax2 : $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$
  - Ax3 : $(\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B)$
- Inference rules (deduction):
  - Modus Ponens : $A, A \rightarrow B \vdash B$
  - Substitution : $B \vdash [A/p]\, B$

3. Lambda Calcul

Lambda Calculus or (λ-Calculus) was introduced in 1930 by Alonzo Church in an attempt to propose a solution to the Leibniz ideal. It has been proven to be equivalent to the Turing Machine and recursive functions. The functional part of this model is the backbone of the functional programming languages we know today.

This model, which originated in theoretical computer science, has applications in :

- Artificial intelligence: lambda calculus offers the possibility of modelling symbolic processing, which can be applied to knowledge bases that use symbolic representation,
- Proof systems: stemming from Formal Systems, λ-Calculus is equipped with theorem construction (or proof) mechanisms. In the case of a programme, it is possible to prove it formally using these mechanisms,

4. Functional programming

Functional programming is a programming paradigm in which processing is described by the application of functions.

Example:

$f(g(x, h(y)))$

$f(x - 1, g(h(x - 1), 2))$

The main characteristics of this programming paradigm are :

- Ease of expressing and proving programmes: this ease stems from the formal nature (symbolic manipulation) of languages based on the functional programming paradigm.
- Absence of assignment: in fact, the application is used to pass parameters (apply a function to a parameter), which makes it possible to give a value to a variable. The main result of this approach is the immutability of variables.
- Absence of sequentiability: in the absence of algorithmic modelling, the idea of sequentiability, which defines processing as a sequence of instructions, is also absent.
- Absence of explicit controls: notably Goto and Exit. Calls to the various functions cannot be assigned with control instructions.
- Calculation based on functions.

However, in order to construct recursive functions, it is necessary to define the only control structure, which is the conditional. In fact, a recursive function must have a 'stop condition' which allows recursive calls to be stopped.

This structure is defined by :

If condition Then instr1 Otherwise inst2 Fsi

In a functional context, we can write the following example:

Fact(n) = If n = 1 then 1 Else Multi(n, Fact(n - 1)) Fsi

5. λ-Notation

The λ-Notation is an essential element of the λ-Calculus. The main motivation for its proposal is to remove a common ambiguity in t raditional notation.

Indeed, the expression $f(x)$ can have two meanings:

- The defintion of the function itself,
- The value of the function for the parameter x: x.

The traditional notation becomes even more ambiguous when we define function composition, per example:

$f(g(x))$

$f(g(x + 1))$

λ-Notation can be defined by:

- $\lambda x.\ M$: The argument x is associated with the body of the function M. The function itself has no name (it is said to be

an anonymous function).

- $\lambda x.\ Mv$: Value of the function evaluated for the value v.

Example:

$\lambda x.\ \left(x^2\right)$: is equivalent to the function definition $f(x) = x^2$

$\lambda x.\ \left(x^2\right)2$: is 4, which refers to the evaluation of the value of f when the parameter *2* is passed to it $f(2) = 2^2 = 4$

Several parameters can be defined as follows:

$\lambda x_1.\ \lambda x_2.\ \lambda x_3.\ ...\lambda x_n.\ M$

Or with the abbreviation:

$lamndax_1x_2x_3...x_n.\ M$

Example:

$lamndax.\ \lambda y.\ (+\ xy) = \lambda xy.\ (+\ xy)$

Note:

Let the definition be as follows in λ-Notation:

$\lambda y.\ (+\ xy)$

In this case x in a constant. This is equivalent to the definition in classical notation:

$f(y) = y + x$

6. λ-Applicatives

Le λ-Calcul defines only two operations :

- Abstraction : it refers to the definition of a statement $\lambda x.\ M$
- Application : which refers to the evaluation of an aabstraction and is noted by juxtaposition $\lambda x.\ Mv$

The application is a left associative operation. Thus, the expression $fabc$ is equivalent to the expression $(((fa)b)c)$

Thus, the application is performed as follows:

- $(\lambda x_1x_2x_3...x_n.\ M)a_1a_2a_3...a_n\ =$
- $(\lambda x_1(\lambda x_2(\lambda x_3...(\lambda x_n.\ M))))a_1a_2a_3...a_n\ =$
- $(\lambda x_2(\lambda x_3...(\lambda x_n.\ M_2)))a_2a_3...a_n$ and M_2 is obtained from M by replacing all instance of x_1 by a_1.

7. Curryfication

Curryfication is a process where a function that takes multiple arguments is transformed into a series of nested functions,

1 each taking a single argument. This transformation is done in a way that preserves the original function's behavior.

We can think of it as a *partial evaluation* operation. If only one argument is given, we can partially evaluate the function with several arguments, giving rise to a new function with one less argument. Clearly, all the arguments need to be supplied in order to carry out a complete evaluation of the function.

Example:

Let be the function $f(x, y, z) = x + y + z$, we want to evaluate its value for the three parameters 3, 5 et 7.

$f(x, y, z) = x + y + z = f'(x)(y)(z) = f'(3)(y)(z)$

ie. the partial evaluation is possible and gives as result $f(3, y, z)$ which can be seen as the new function : $f'(3) = g$

Thus : $f(3, y, z) = g(y, z) = 3 + y + z$

In the same way : $g(3 + y + z) = g(y)(z) = g(5)(z)$

The partial evaluation $g(5, z) = 3 + 5 + z = 8 + z$, can be seen as the definition of the new function $g'(5) = h$ such as $h(z) = 8 + z$

Finally, we evaluate $h(7) = 8 + 7 = 15$.

8. Formal definition of λ-Calculus

8.1. λ-Calculus syntax

The basic definition of the λ-Calculus syntax is given by the following elements:

- An alphabet consisting of :
  - a set of variables (x, y, z, ...)
  - parentheses ()
  - the symbol λ.
- The set of elementary statements defined by the rules :
  - Each variable is a term,
  - If X and Y are λ-terms, then (XY) is a λ-term,
  - If X is a term and x is a variable, then λx.X is a term.

This definition given above is the definition given for the pure λ-Calculus. Another definition can be given for the so-called applied λ-Calculus as follows:

- An alphabet consisting of :
  - a set of variables (x, y, z, ...)
  - a set of constants (a, b, c, d ..)
  - a set of operations (+, -, and, or, ...)

- parentheses ()
- the λ symbol.

- The set of elementary statements defined by the rules :
  - Each variable is a term,
  - Each constant is a term,
  - Each operation is a term,
  - If X and Y are λ-terms, then (XY) is a λ-term,
  - If X is a term and x is a variable, then λx.X is a term.

8.2. Free variables and bound variables

Let be the term $\lambda x.\ e$, all instances of x in e are said bound.

8.2.1. Free variables

The set of free variables is called FV. This set is defined by :

- FV(a) = {}
- FV(x) = {x}
- FV(XY) = FV(X) ∪ FV(Y)
- FV(λx.X) = FV(X) - {x}

8.2.2. Bound variables

The set of bound variables is called BV and is defined as follows:

- BV(a) = {}
- BV(x) = {}
- BV(XY) = BV(X) ∪ BV(Y)
- BV(λx.X) = BV(X) ∪ {x}

Example:

Let be the λ-term : $X = (\lambda x.\ (\lambda y.\ y)\lambda z.\ x)$

This term takes the form of $X = MN$ where

- $M = \lambda x.\ (\lambda y.\ y)$
- $N = lamndaz.\ x$

FV(X) = FV(M) ∪ FV(N)
= FV($\lambda x.\ (\lambda y.\ y)$) ∪ FV($\lambda z.\ x$)
= FV($\lambda y.\ y$) - {x} ∪ FV(x) - {z}
= FV(y) - {y} - {x} ∪ {x} - {z}
= {} ∪ {x} = {x}

8.3. Substitution

Let be N and M in L, and x a free variable in M.

We note [N/x]M the substitution of all free instances of x in M by N.

Substitution can be defined by the following set of rules:

- [N/x]a = a
- [N/x]x = N
- [N/x]y = y if y ≠ x
- [N/x](XY) = [N/x]X [N/x]Y
- [N/x](λx.M) = (λx.M) (x is not free)
- [N/x](λy.M) =
  - = λy.[N/x]M, if y is not free in N,
  - = λz.[N/x]([z/y]M), if y is free in N.

Example:

M = λy.xy

- If N = t : [t/x]λy.xy = λy.ty
- If N = y : [y/x]λy.xy = λy.yy (that can lead to confusing the free instance with the free instance that replaced x).

Thus, a correct substitution can be:

[y/x]λy.xy = λz.[y/x]([z/y]xy)
= λz.[y/x]xz
=λz.yz

9. Conversions

9.1. α-Conversion

α-Conversion is a transformation that renames bound variables within a lambda expression without changing the meaning of the expression. It is a fundamental concept in lambda calculus and is used to ensure that the choice of bound variables does not affect the behavior of a lambda expression.

If y is not free in M then λx.M = λy.[y/x]M.

α-Conversion allow the definition of the congruency between λ-terms. Congruency in lambda calculus is a fundamental concept that defines when two lambda expressions are considered equivalent. It is based on the idea that two expressions are congruent if they can be transformed into each other using a set of predefined rules.

X ≡ Y iff X → Y1 → Y2 → Y3 → Y4 ... Yn → Y, n ≥ 1.

Congruency is an equivalence relation (reflexive, symmetrical and transitive).

9.2. β-Conversion

Beta conversion is a fundamental transformation rule in lambda calculus that defines how functions are applied to arguments. It is the core mechanism for evaluating lambda expressions and computing results.

The application (λx.M)N is called β-Radical (or β-redex or simpely redex) is a reduction and is evaluated as follow:

(λx.M) = [N/x]M

If λx.M is a function, its application on any N can be seen as the substitution of N for x in the expression M. N can be itself a function.

A reduction relation (⇒) can be defined between two λ-terms X and Y, and we say that X is reduced to Y and we write X ⇒ Y iff Y is the result of a sequence of α and β-Conversions.

(⇒) is reflexive and transitive.

Examples :

- (λx.xy)F ⇒ Fy
- (λx.y)F ⇒ y (no free instances of x in y)
- ((λx.(λy.yx)z)v) ⇒ ((λy.yv)z) (⇒ zv)
- (λx.(λy.yx))z v ⇒ (λy.yz)v ⇒ vz
- (λx.xxy)(λx.xxy) ⇒ (λx.xxy)(λx.xxy)y

In the λ-Calculus applied, we can define operators as applications (which give rise to reductions). For example, we write the sum () as : (1 x) which is seen as ((+ 1) x), i.e. we apply an increment (+1) to x.

Examples :

- (λx. + x 1) 4 ⇒ + 4 1 ⇒ 5
- (λx. + x x) 4 ⇒ + 4 4 ⇒ 8

9.3. Proper theory of λ-Calcul

The set of statements of the form X ⇒ Y that are true can be defined by applying :

- The axiom X ⇒ X
- The rules :
  - α et β Conversions
  - The deduction rules:
    - X ⇒ Y → ZX ⇒ ZY
    - X ⇒ Y → XZ ⇒ YZ
    - X ⇒ Y → λx.X ⇒ λx.Y
    - X ⇒ Y et Y ⇒ Z → X ⇒ Z

As with Formal Systems, X ⇒ Y is theorem if and only if there exists a proof using only the axioms and rules above.

9.4. η-Conversion (Eta-Conversion)

An η-conversion is applied before the β-conversion. An η-Conversion is defined by:

If x is not free in M then λx.(Mx) = M

Important Note: we are not talking here about an application, but about a simplification of the term before any β-conversion.

This conversion can be proved as follows:

λx.(Mx)Y = MY thus λx.(Mx) = M

9.5. Normal form

A λ-expression is said to be in normal form if it cannot be further reduced using the reduction rules of the calculus. This means that no beta reduction or other applicable reduction rules can be applied to the term to produce a simpler equivalent term. ie. the term does not contain any redex [(λx.M)N].

We can write : ∄ e' tq e ⇒ e'

In the case of recursion, we say that we don't have a normal form.

The notion of normal form is crucial in lambda calculus for several reasons:

- **Uniqueness:** If a term has a normal form, it is unique. This means that regardless of the reduction strategy used, the final result will be the same.
- **Evaluation:** Normal forms are often used to evaluate expressions in λ-calculus. By reducing a term to its normal form, we can obtain its value or meaning.
- **Equivalence:** Two terms are considered equivalent if they have the same normal form. This provides a way to determine whether two expressions are semantically identical.

9.6. Ordre de réduction

9.6.1. Normal Order (Call-by-Name)

The substitution is applied to the outermost, leftmost variable in the same level. This order is also known as normal order. Semantically, the variable is replaced by the argument, whatever that argument may be (without evaluating it).

This strategy tends to delay evaluation of arguments until they are needed, which can be beneficial for avoiding unnecessary computations in some cases.

9.6.2. Call-by-Value

The substitution is applied to the innermost, leftmost variable in the same level. Semantically, you don't replace a variable with the argument until after the argument has been evaluated (the argument is a value).

This strategy evaluates arguments before applying functions to them, which can be more efficient in some cases but may lead to infinite loops if arguments are non-terminating.

Example :

Le be the λ-expression : (λx.(λy. + x λx.- x 3)y) 5 6

Call-by-Name:

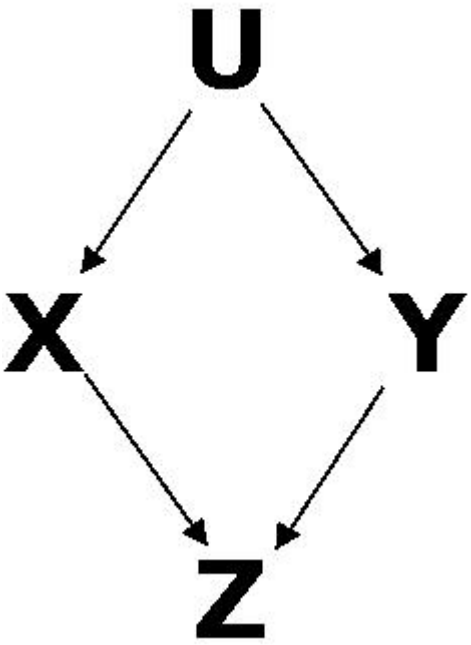
(λx.(λy. + x λx.- x 3)y) 5 6 = (λy. + 5 λx.- x 3)y 6
= + 5 ((λx.- x 3)6)
= + 5 (- 6 3)
= + 5 3
= 8

Call-by-Value:

(λx.(λy. + x λx.- x 3)y) 5 6 = (λx.(λy. + x (- y 3))) 5 6
= (λy. + 5 (- y 3)) 6
= + 5 (- 6 3)
= + 5 3
= 8

9.6.3. Church-Rosser Theorem

If U ⇒ X and U ⇒ Y, it exists Z such as X ⇒ Z and Y ⇒ Z.



If M ⇒ N and N is a normal form, then there is a sequence of reductions from M to N in normal order.

In other words, the normal order guarantees that the normal form will be found if it exists for the term M. In fact, the order by value may not lead to the normal form in cases where the arguments are recursive.

Example:

Let be the expression : (λx.y)λx.xx)(λx.xx

Call-by-Value:

(λx.y)λx.xx)(λx.xx ⇒ (λx.y)λx.xx)(λx.xx
⇒ (λx.y)λx.xx)(λx.xx
⇒ (λx.y)λx.xx)(λx.xx
⇒ ...

Call-by-Name :

(λx.y)λx.xx)(λx.xx ⇒ y

10. Calculability and λ-Calcul

The equivalence between the Turing Machine and the λ-Calculus means that any computable function can be represented as an expression in the λ-Calculus just as it can be represented as a program on the Turing Machine. How is this done?

10.1. First example : True, False and Not

As with binary encoding, we need to define a convention for representing the different values. The True value is represented by the value '1' and the False value is represented by the value '0'. For example, the f ollowing &ambda;-expressions:

- True = λxy.x
- False = λxy.y
- Not = λzxy.zyx

Let's check that this representation is correct:

Not True = (λzxy.zyx)(λxy.x)
= (λxy.(λxy.x)yx)
= (λxy.y)
= False

Not False = (λzxy.zyx)(λxy.y)
= (λxy.(λxy.y)yx)
= (λxy.x)
= True

10.2. And and Or logical operators

In the same way, we can define the And operator as follows:

- And = λxy.xyx

We can check that this definition is correct as follows:

And True False = (λxy.xyx) True False
= True False True
= (λxy.x) False True

= False

And True True = (λxy.xyx) True True
= True True True
= (λxy.x) True True
= True

And False True = (λxy.xyx) False True
= False True False
= (λxy.y) True False
= False

And False False = (λxy.xyx) False False
= False False False
= (λxy.y) False False
= False

In the same way, the Or operator can be defined as follows:

- Or = λxy.xyx

10.3. Church Numerals

As with Boolean values, Church has defined representations for numeric values as follows:

- 0 = λxy.y
- 1 = λxy.xy
- 2 = λxy.xxy
- 3 = λxy.xxxy
- ...

Operations are defined by the following expressions:

- Addition = λfxyz.fy(xyz)
- Multiplication = λxyz.x(yz)

Exercise:

Do the following calculations:

- 1 + 2
- 2 + 3

11. Combinators and Combinatorial Logic

11.1. Combinators

Combinators are a class of lambda expressions that do not contain any free variables. In other words, they are functions

that take other functions as arguments and return functions. Combinatorial logic is a system of formal logic that uses combinators to represent and manipulate logical expressions.

Combinators are used to evacuate the notion of variable and simplify the reduction of λ-expressions. They were proposed by Haskell Curry in an attempt to define a reduced set of combinators that allows computable functions to be defined.

11.2. Basic Combinators

The purpose of this reduced set of combinators is to generate all the other functions. These combinators are:

- I = λx.x (identity)
- K = λxy.x (constant function)
- S = λxyz.xz(yz) (application of two sub-expressions to the same argument)
- C = λfab.aba (inversion of parameters)
- Z = λfgx.f(gx) (composition of functions - changing the order of application)
- M = λf.ff (apply the function to itself)

The names may change from one reference to another, given that the original version was in German and the first use of these combinators in an English research article already included name changes.

Proving that these combinators can generate all computable functions is not our objective in this course. We will limit ourselves to the following examples:

- SKKx = Kx(Kx) = x

We can see that we can obtain the I combinator from the S and K combinators.

- KIxy = Iy = y

We can thus obtain the combinator which allows us to select the second element by combining the two combinators K and I. If we go back to the definitions of True and False values used to perform logical calculations, we can see that:

- True = K
- False = KI

11.3. Combinators for list manipulation

Several non-standard combinators have been proposed to model different different functionalities in different contexts. For list list manipulation, we have three well-known combinators:

- Cons : concatenation of two lists
  - Cons = (λa.λb.λf.fab)
- Car : first element in the list
  - Car = (λc.c(λa.λb.a))
- Cdr : rest of the list
  - Car = (λc.c(λa.λb.b))

Example :

Let's check whether Car(Cons pq) really equals p:

Car (Cons p q) = (λc.c(λa.λb.a))((λa.λb.λ.fab) p q)
= ((λa.λb.λf.fab) p q)(λa.λb.a)
= (λa.λb.a) p q
= p

11.4. Y Combinator

The Y combinator is a fixed-point combinator in lambda calculus, a mathematical formalism for expressing functions and computation with lambda expressions. It is a powerful tool that enables the definition of recursive functions without explicitly naming them.

Fixed-point combinators are expressions that satisfy the equation f(x) = x. In other words, when applied to a function, they return a value that, when the function is applied to it, yields the same value.

Let be the function :

- Fact = (λn. If (eq n 0) 1 (* n (Fact(- n 1))))

This function has the form:

- Fact = (λn.(…Fact…))

By performing an abstraction (the opposite of β-abstraction), we can rewrite this function as :

- Fact = (λf.(λn.(…f…))) Fact
- Fact = (λf.(λn. If (eq n 0) 1 (* n (f(- n 1))))) Fact

ie. :

- Fact = H Fact

Fact is said to be the fixed point of H of the form:

- YH = H(YH) where Fact = YH

Let be the Y Combinator:

- Y = (λh.(λx.h(xx))(λx.h(xx)))

It is sufficient to show that YH = H(YH) for the above condition to be satisfied.

YH = (λh.(λx.h(xx))(λx.h(xx))) H
= (λx.H(xx))(λx.H(xx))
= H (λx.H(xx))(λx.H(xx))
= H (YH)

Example:

From what we have shown, we can confirm that:

- Fact N = (YH) N = H(YH) N

Let evaluate Fact(1)

Fact 1 = YH 1
= H(YH) 1
= (λf.(λn. If (eq n 0) 1 (* n (f(- n 1))))) YH 1
= (λn. If (eq n 0) 1 (* n (YH(- n 1)))) 1
= (Si (eq 1 0) 1 (* 1 (YH(- 1 1))))
= (* 1 YH(0))
= (* 1 H(YH)0)
= (* 1 (λf.(λn. If (eq n 0) 1 (* n (f(- n 1))))) YH 0)
= (* 1 (λn. If (eq n 0) 1 (* n (YH(- n 1)))) 0)
= (* 1 (If (eq 0 0) 1 (* 0 (YH(- 0 1)))))
= (* 1 1)
= 1

12. Lambda Calculation Simply Typed

12.1. Introduction

The theory of the λ-Calculus can be extended by adding restrictions on t he application of one term to another by introducing the notion of type. Indeed, some combinators depend in their definition on the type of its parameters. For example, the combinator :

- Si = λfxy.fxy

This combinator makes sense if the first argument passed is a Boolean (True or False) as follows

- λfxy.fxy (True) = λxy.True xy = λxy.x
- λfxy.fxy (False) = λxy.False xy = λxy.y

This is equivalent to the known conditional strcuture:

- z = f ? x : y

However, in the basic definition of λ-Calculus, any function can be passed as the first parameter. Hence the need to apply restrictions depending on the type of arguments.

12.2. Definition

Let the set $\mathscr{V}$ = {α, β, γ, ...} be the set of type variables. The set $\mathscr{T}$ of simple types is defined by :

- (variable type) If $\alpha \in \mathscr{V}$ then $\alpha \in \mathscr{T}$
- (compound type) If $\alpha, \beta \in \mathscr{T}$ then $\alpha \rightarrow \beta \in \mathscr{T}$

This definition can be expressed inductively by :

- $\mathscr{T} = \mathscr{V} \mid \mathscr{T} \in \mathscr{T}$

The operator → is right associative. That is, the type α → β → γ → δ is equivalent to the type α → (β → (γ → δ)).

12.3. Type notation

The type is denoted by ':' (the colon).

If x is of type τ, then α:τ is written according to the following two conditions:

- No untyped variable can have more than one type. We cannot have α:τ, α:σ with τ ≠ σ.
- Each type τ can be assigned to an infinite number of variables.

12.4. Curry system (Simple typed system)

The Curry system can be defined by the following elements:

- Alphabet :
  - propvars = the set of type variables,
  - termvars = the set of term variables.
- Formal syntax:
  - proposition ::= propvars | (proposition → proposition)
  - term ::= termvars | (<term><term>) | (λtermvars:<proposition>.term)
  - formula ::= term:proposition
- Axioms : None
- Type assignment rules (inference rules)

12.4.1. Definition of typed variables

$$\frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha}(var)$$

12.4.2. Application type

$$\frac{\Gamma \vdash N : A, M : A \rightarrow B}{\Gamma \vdash MN : B}$$

In other words, if N is of type A and M is of type A &raar; B then the application of M to N (MN) will be of type B. We can explain this by saying: in order to apply the term M to N whose type is A then M must be of a type of the form A → B.

12.4.3. Abstraction type

$$\frac{\Gamma, (x : A) \vdash M : B}{\Gamma \vdash \lambda x : A.\ M : A \rightarrow B}$$

In other words, by abstracting x which is of type A into a term M of type B, we create a term (a function) of type A → B.

An important condition: the x must not be free in M.

12.5. Examples

- $(\lambda x{:}\sigma.x) : \sigma \rightarrow \sigma$
- $(\lambda x{:}\sigma.\lambda y{:}\tau.x) : \sigma \rightarrow \tau \rightarrow \sigma$

13. Summary of this first chapter

This chapter had two essential aims:

- To introduce the notion of Formal Systems,
- To introduce the λ-Calculus.

Formal systems provide a powerful mathematical tool for modelling objects. The importance of these systems is their ability to generate new objects by starting from a minimal set (axioms) and applying well-defined rules. In this way, it is possible to generate behaviour without essentially understanding the semantics of the rules applied during the generation of new objects (or new theorems).

Understanding Formal Systems allows us to understand λ-Calculus. The latter is a powerful tool for representing functions while distinguishing between the definition of the function and the application of the function to a parameter.

The λ-Calculus became even more important when Church and Turing succeeded in proving that it is equivalent to the Turing Machine, in other words, this Formal System can represent any calculable function. This power enabled it to become the basis of functional programming languages.

We have also seen the notion of data type. The aim of this extension is to limit the application of λ-terms in order to preserve their meaning, as we have seen with the conditional combinator.

The extension still respects the principles of definition of Formal Systems, which makes it possible to keep the computability characteristics obtained with the untyped λ-Calculus.

Version 0.1
Last updated 2024-10-13 00:39:27 +0100