An operative definition of α-equivalence in λ-calculus

Functional Programming Project

Fabio Brau

February 1, 2021

Contents

1 Introduction

The aim of this project is to provide a formal and operative definition of α-equivalence on the λ-terms without seeing it as a transitive closure of a relation and without assuming conventions on the name of the variables.

In Section 2 we will introduce the definition of λ-terms, the definition of *bound* and *free* variables, and the definition of *substitution*.

In the Section 3 we will provide the definition of the α-equivalence by using the definition of substitution.

In the Section 4 we will implement the definitions of the previous sections in the functional programming language `Haskell` that we can test with some examples in the last section.

2 Lambda Terms

From an intuitive point of view, a λ-term is a representation of a mathematical function written as a combination of *variables*. It is quite surprising to notice that "a systematic notation for functions is lacking in ordinary mathematics" [2]. In fact, the meaning of $f(x)$, that is the usual accepted notation to indicate a function (*Euler Notation*), is not uniquely determined and has to be deduced from the context. Sometimes, with this notation, we refer to a function depending on the variable x; sometimes we refers to the evaluation of the function in a value (or in a point, a vector, a matrix, a set and so on) equal to x.

An unambiguous way to indicate that the function f depends on the variable x could be the notation: $x \mapsto f(x)$. And, we can use the notation $f(x)$ when we want to evaluate f in the object x. The introduction of the symbol λ could be helpful, in a typographic sense, by shortening the notation $x \mapsto f(x)$ in $\lambda x.f(x)$.

Definition 1 (λ-terms)**.** Let be $V = \{v_1, v_2, \dots\}$ an infinite set of *variables*. The set of the λ-terms, indicated with Λ, is recursively defined as follow

- $V \subseteq \Lambda$, i.e., each variable is a λ-term;
- If M is a λ-term, then $\lambda x.M \in \Lambda$, for any variable x, is a λ-term called *abstraction*;
- If M, N are λ-terms then (MN) is a new λ-term called *application*.

It's compulsory to observe that the symbols $M, N, \dots, v_0, v_1, \dots$ have to be intended as names of λ-terms. The *assignment* of a name to a λ-term, indicated with $=$, only holds and has sense in the metalanguage. For example, in the formula $M = \lambda x.x$, the symbol M is just the label of the λ-term $\lambda x.x$.

By assuming that the abstraction and the application are associative on the left (as operators of λ-terms in the metalanguage) we can introduce the following notation.

Notation 1. Let x, y be two variables and M a λ-term, then the formula $\lambda xy.M$ is a shorter notation for $\lambda x.\lambda y.M$ that, by the left associative convention, represents uniquely $\lambda x.(\lambda y.M)$. Similarly, if X, Y, Z are λ-terms, then the formula XYZ uniquely represents $((XY)Z)$ by the left associative rule.

Definition 2. We will say that two λ-terms M, N are *syntactically equivalents*, and we will write $M \equiv N$, if each one can be mutually translated into the other trough the notation Notation 1.

2.1 Bound and Free Variables

Let us consider the λ-expression $M = y(\lambda x.x)$. It is evident that the two variable x, y has a different meaning in the λ-term. The variable x, that appears in M under the scope λ, is *bound* and is not a constant of the λ-term. In a certain sense, that it will be more clear later, the variable x can be substituted with another variable without that M loses its meaning. Differently the variable y is *free* and stores a different information that characterizes the term M. Observe, for example, that the term M could represent the formula $\int_0^y x\,dx$, in which the variable x is usually called "silent".

Definition 3. Given a λ-term M, the set $\mathcal{F}(M)$ of the *free variables* of M is recursively defined as follow:

- If $M = x$, where $x \in V$, then $\mathcal{F}(M) = \{x\}$;
- If $M = \lambda x.M_1$, then $\mathcal{F}(M) = \mathcal{F}(M_1) \setminus \{x\}$;
- If $M = M_1M_2$, then $\mathcal{F}(M) = \mathcal{F}(M_1) \cup \mathcal{F}(M_2)$.

Analogously, the set $\mathcal{B}(M)$ of the *bound variables* of M is recursively defined as follow:

- If $M = x$, where $x \in V$, then $\mathcal{B}(M) = \emptyset$;
- If $M = \lambda x.M_1$, then $\mathcal{B}(M) = \mathcal{B}(M_1) \cup \{x\}$;
- If $M = M_1M_2$, then $\mathcal{B}(M) = \mathcal{B}(M_1) \cup \mathcal{B}(M_2)$;

By induction on the construction of Λ, it is easy to prove that the definition above is well-placed.

Observation 1. Observe that, by the definition above, a variable x can be either a free and a bound variable of a λ-term M. For example, the λ-term $M = x(\lambda x.xx)$ is such that $\mathcal{F}(M) = \mathcal{B}(M) = \{x\}$.

2.2 Substitution without Capture

Let us consider a λ-term M and x a free variable of M. In this section we want to define the *substitution operation* in order to substitute the variable x with another λ-term N in M.

The following definition was first proposed by [2] in order to avoid the issue known as *binding of a free variable.*

Definition 4 (Substitution)**.** Let $M, N \in \Lambda$, let us define the operation of *substitution* that acts by substituting a variable x in M with N, returning a new λ-term $M[x := N]$. The definition is recursive on the construction of Λ.

Case 1 If M is a variable:

- If $M = x$, then $M[x := N] \equiv N$;
- If $M = y$ and $y \neq x$, then $M[x := N] \equiv y$;

Case 2 If $M = M_1 M_2$ is an application:

- $M[x := N] \equiv (M_1[x := N])(M_2[x := N])$;

Case 3 If M is an abstraction:

- If $M = \lambda x.M_1$, then $M[x := N] \equiv M$;
- If $M = \lambda y.M_1$ and $x \neq y$, then

$$M[x := N] \equiv \lambda z.M_1[y := z][x := N]$$

where: $z = y$ if $x \notin \mathcal{F}(M_1)$ or $y \notin \mathcal{F}(N)$; z is choose to be not in M and not in N otherwise;

The first and the second case are intuitive. Regarding the third case few more words are required. If M is an abstraction of the form $\lambda x.M_1$, substituting x with N has no meaning because x is bound. Differently, if we want to substitute x in a λ-term of the form $\lambda y.M_1$ (and $x \neq y$), then consider the following example. Let $M = \lambda y.x$ and $N = y$. Observe that y is free in N and is bound in M, and if we change x with N without introducing a new variable z we will obtain $\lambda y.y$ that is not equivalent, in a sense that it will be more clear later, to the desired result. This phenomena is called *binding* of a free variable.

Lemma 1. *For any $M, N \in \Lambda$, the following statements hold.*

1. *If $x \notin \mathcal{F}(M)$ and $x \notin \mathcal{B}(M)$, then $M[x := N] \equiv M$;*
2. *If $y \notin \mathcal{F}(M)$ and $y \notin \mathcal{B}(M)$, then $M \equiv M[x := y][y := x]$;*
3. *If $x \in \mathcal{F}(M)$ and $y \notin \mathcal{F}(M)$, then $x \notin \mathcal{F}(M[x := y])$ e $y \in \mathcal{F}(M[x := y])$;*

Proof. Each statement can be proved by (strong) induction on the construction of the λ-terms. The key idea is to observe that $\Lambda = \cup_{n=0}^{\infty} \Lambda_n$ where: $\Lambda_0 = V$ and Λ_n contains all the λ-terms that can be generated by application or abstraction of λ-terms in Λ_k with $k < n$. After that, by supposing that independently $1, 2, 3$ hold for Λ_k with $k < n$, it is easy by applying the Definition 4 to prove that the statement hold also for Λ_n. □

3 α-equivalence

From an intuitive point of view two λ-terms are α-equivalents if each one can be transformed in the other by changing one or more bound variables. This definition hides a number of complications, and it is not easy to find a definition that is either operative and easy to state.

Church, [1], solves the problem by including the α-equivalence in a semantic level by adding the statement $\lambda x.M = \lambda z.M[x := z]$ (called α) in the list of the *conversion rules*. Moreover, he required different assumptions on the construction of the λ-terms, for example in his formulation only abstraction trough free variables was accepted ($\lambda y.x$ is not considered a λ-term).

In Curry's formulation [2] a different idea is used:

1. The definition of α-equivalence is provided for abstractions by stating $\lambda x.M \equiv_\alpha \lambda z.M[x := z]$.

2. The relation $\equiv_\alpha$ is then extended to the whole λ-terms by transitive closure.

In [3], the definition of α-equivalence is included in the syntactic equivalence and it is defined informally as: "M is α-congruent to N, if N results from M by a series of changes of the bound variables". Moreover, the author needs a variable convention: When two terms are in the same statement (or definition, theorem, etc.) then different names for the variables have to be used.

Even if the definition above are totally valid by a logical and mathematical point of view, by a practical point of view they contains two uncomfortable things, so that can be not easy to code them in a functional programming language. For example, they don't provide an operative definition [2], and they use conventions on the terms ([1, 3]). The following lines are an attempt of an operative definition of α-equivalence that doesn't requires further conventions or assumption and that can be easily implemented trough a functional paradigm.

Definition 5 (α-equivalence)**.** Let us define $\equiv_\alpha$ recursively and in parallel with the construction of Λ. Let Λ_0 be the set of the terms of rank 0 (i.e, of the form x for each variable in V), we define

$$\forall x, y \in \Lambda_0, \quad x \equiv_\alpha y \iff x = y.$$

Let us consider the set Λ_k of the λ-terms of rank k defined as $\Lambda_k = \hat{\Lambda}_k \cup \bar{\Lambda}_k$ where

$$\hat{\Lambda}_k = \{\lambda x.M \, : M \in \Lambda_{k-1}, \, x \in V\}$$
$$\bar{\Lambda}_k = \{(MN), (NM) \, : \, M \in \Lambda_{k-1}, \, N \in \Lambda_i, \, i < k\}$$

Let us define over this set the relation $\equiv_\alpha$ as follow

- If $M, N \in \hat{\Lambda}_k$, with $M = \lambda x.M_1$ and $N = \lambda y.N_1$, then
  - If $x = y$, then
  $$M \equiv_\alpha N \iff M_1 \equiv_\alpha N_1$$
  - If $x \neq y$, then
  $$M \equiv_\alpha N \iff M_1 \equiv_\alpha N_1[y := x] \quad \wedge \quad (y \notin \mathcal{F}(M_1) \wedge x \notin \mathcal{F}(N_1))$$
- If $M, N \in \bar{\Lambda}_k$, with $M = M_1 M_2$ and $N = N_1 N_2$, then
$$M \equiv_\alpha N \iff M_1 \equiv_\alpha N_1 \wedge M_2 \equiv_\alpha N_2$$

Observing that $\Lambda = \cup_{k=0}^{\infty} \Lambda_k$, and observing that $\Lambda_i \cap \Lambda_j = \emptyset$, we can extend $\equiv_\alpha$ to the whole set of λ-terms.

Proposition 1. *The relation* $\equiv_\alpha \subseteq \Lambda \times \Lambda$ *is an equivalence.*

Proof. We want to proceed by strong induction over the rank. By construction $\equiv_\alpha$ is an equivalence over Λ_0. Let us suppose that $\equiv_\alpha$ is an equivalence for Λ_i with $i < k$. We want to prove *reflexivity, symmetry and transitivity* property.

Reflex. If $M \in \hat{\Lambda}_k$ with $M = \lambda x.M_1$, then $M \equiv_\alpha M \iff M_1 \equiv_\alpha M_1$ that is true because $\equiv_\alpha$ is of equivalence on Λ_{k-1}. If $M \in \bar{\Lambda}$ with $M = M_1 M_2$, then $M \equiv_\alpha M$ if and only if $M_1 \equiv_\alpha M_1$ and $M_2 \equiv_\alpha M_2$, that is true because $\equiv_\alpha$ is of equivalence on Λ_i with $i < k$.

Symmet. Let $M \equiv_\alpha N$. If $M, N \in \hat{\Lambda}_k$ with $M = \lambda x.M_1$ and $N = \lambda y.N_1$, then by construction: if $x = y$, $M_1 \equiv_\alpha N_1$ implies that $N_1 \equiv_\alpha M_1$; otherwise, if $x \neq y$, then $M_1 \equiv_\alpha N_1[y := x]$ (with $x \notin \mathcal{F}(N_1)$ and $y \notin \mathcal{F}(M_1)$) and so by Lemma 1 $M_1[x := y] \equiv_\alpha N_1[y := x][x := y] \equiv N_1$. The case $M, N \in \bar{\Lambda}_k$ follows from Inductive Hypothesis.

Trans. Suppose that $X, Y, Z \in \Lambda_k$, $X \equiv_\alpha Y$, and $Y \equiv_\alpha Z$. If $X, Y, Z \in \hat{\Lambda}_k$ with $X = \lambda x.X_1$, $Y = \lambda y.Y_1$, $Z = \lambda z.Z_1$ then for point 3 of Lemma 1 $x \notin \mathcal{F}(Z_1)$ and so $X_1 \equiv_\alpha Y_1[y := x] \equiv_\alpha Z_1[z := y][y := x] \equiv_\alpha Z_1[z := x]$. Therefore, if $X, Y, Z \in \bar{\Lambda}_k$, then, because the transitive property is valid for the single terms of the applications in $\bar{\Lambda}_i$ with $i < k$, then by inductive hypothesis the property also hold for Λ_k.

□

4 Implementation

All the definitions in the previous sections can be implemented in Haskell language.

4.1 λ-terms

The λ-terms are formally defined as a data type as follow

```
-- Definition of lambda terms
data LamTerm = Var String | Abs String LamTerm | App LamTerm LamTerm
```

The following function checks whether a variable x is free in a term M. The definition is made by cases on the construction of the type `LamTerm`.

```
-- Function that check if a variable is free
is_free x (Var y)     = (x == y)
is_free x (Abs y e1)  = if (x==y) then False else is_free x e1
is_free x (App e1 e2) = is_free x e1 || is_free x e2
```

In a similar manner we can define the function `is_bound` that check whether a variable is bound in a λ-term

```
-- Function that check if a variable is bound
is_bound x (Var y)     = False
is_bound x (Abs y e1)  = if (x==y) then True else is_bound x e1
is_bound x (App e1 e2) = is_bound x e1 || is_bound x e2
```

4.2 Substitution

Before presenting the `subs` function, we have to notice that when a λ-term is an abstraction (Case 3 of Definition 4) we carefully need to introduce a new fresh variable. The following functions allows to provide a new variable that is not present in a term neither as free or bound variable.

```
-- Function that generate fresh Var
-- Auxiliary function that generate a new variable from an old name
fresh_from_str m y =
  if not ((is_free y m) || (is_bound y m)) then
    y
  else
    fresh_from_str m (y ++ "+")

-- fresh m --> String, return a string the is not free and not bound in m
fresh m = fresh_from_str m "a"
```

Finally, the function `subs` take respectively a variable x, a term M and a term N and provides the λ-term $M[x := N]$ by using Definition 4.

```
-- Substitution function:
-- (subs x m n) substitutes the variable x in m with the lambda term n
--
```

```
-- Case 1
subs x (Var y) n = if (x == y) then n else (Var y)

-- Case 2
subs x (App m1 m2) n = App (subs x m1 n) (subs x m2 n)

-- Case 3
subs x (Abs y m) n =
  if (x == y) then
    (Abs y m)
  else
    if not ((is_free x m) && (is_free y n)) then
      Abs y (subs x m n)
    else
      Abs (fresh (App m n)) (subs x (subs y m (Var (fresh (App m n)))) n)
```

4.3 α-equivalence

In conclusion, the function `alpha_eq` checks whether two λ-terms are α-equivalents by following the Definition 5.

```
-- Check for Alpha equivalence
alpha_eq (Var x) (Var y) = (x == y)
alpha_eq (App m1 m2) (App n1 n2) = (alpha_eq m1 n1) && (alpha_eq m2 n2)
alpha_eq (Abs x m) (Abs y n) =
  if (x==y) then
    alpha_eq m n
  else
    (not(is_free y m)) && (not(is_free x n)) && (alpha_eq m (subs y n (Var x)))
-- Spurious cases
alpha_eq _ _ = False
```

Observe that the last line is compulsory in order to define the function in all the possible cases and to avoid `Non-exhaustive patterns in function alpha_eq` error.

5 Examples

The following lines provide a list of examples of utilizations of the previous definitions.

It is helpful to introduce the function `l2s` that prints an object of type `LamTerm` as as string. The function is recursively defined on the definition of the type

```
-- Prinitng function
l2s (Var n)     = n
l2s (App e1 e2) = "(" ++ (l2s e1) ++ " " ++ (l2s e2) ++ ")"
l2s (Abs n e)   = "\\" ++ n ++ "." ++ (l2s e)
```

We can put all the definition above in the module `LambdaCalcolo.hs` and load it in `ghci`

```
GHCi, version 8.4.4: http://www.haskell.org/ghc/  :? for help
Prelude> :load LambdaCalcolo.hs
[1 of 1] Compiling Main             ( LambdaCalcolo.hs, interpreted )
Ok, one module loaded.
```

Initially we can define and print two terms

```
*Main> m = Abs "x" (App (Var "x") (Var "y"))
*Main>
*Main> l2s m
"\\x.(x y)"
*Main>
*Main>
*Main> n = Abs "z" (App (Var "z") (Var "x"))
*Main>
*Main> l2s n
"\\z.(z x)"
```

We can check the presence of bound and free variables by running

```
*Main> is_free "x" m
False
*Main> is_bound "x" m
True
```

We can now check whether the substitution is made without capture by running

```
*Main> l2s (subs "y" m (Var "x"))
"\\a.(a x)"
*Main>
*Main> l2s (subs "y" m n)
"\\a.(a \\z.(z x))"
*Main>
```

And finally we can observe that the two terms m and n are not equivalents.

```
*Main> alpha_eq m n
False
*Main>
*Main> alpha_eq m (subs "x" n (Var "y"))
True
*Main>
```

References

[1] Alonzo Church. *The calculi of lambda-conversion.* Princeton University Press, 1941.

[2] Haskell B. Curry, Robert Fyes, and William Craig. *Combinatory Logic. Volume I.* North-Holland, Amsterdam, 1958.

[3] H.P.Barendregt. *Studies in Logic and the Foundation of Mathematics - The Lambda calculus. Its Syntax and Semantics.* Ed. by J.Barwise et al. Elsevier, 1984.