

---

# Lambda Calculus

by

**Antonino Salibra**

**Università Ca'Foscari di Venezia**

## **Chapter 1**

### **Index**

- Introduction to Lambda Calculus
- Lambda Calculus and Computability Theory
- Substitution and Beta Reduction

[www.dsi.unive.it/~salibra](http://www.dsi.unive.it/~salibra)

# Lambda calculus: introduction

---

- **Function versus Rule (extension versus intension):**
  - A function  $f: X \rightarrow Y$  is determined by a domain  $X$ , a codomain  $Y$  and a binary relation  $\{(x, f(x)) : x \text{ in } X\}$ .  $f: X \rightarrow Y = g: X' \rightarrow Y'$  iff  $X = X'$ ,  $Y = Y'$  and  $\{(x, f(x)) : x \text{ in } X\} = \{(x, g(x)) : x \text{ in } X'\}$ .
  - **Intension:** In computer science a function is defined by a rule. For example,  $f(x) = x^2 + 3x + 5$ . The domain and codomain is not specified (for example, is the domain of  $f$  equal to the set of real numbers or to the set of complex numbers?). The extension of a function as rule is its input-output behavior (as set of pairs). Two functions as rules may be not equal (for example,  $f(x) = x$  and  $g(x) = x+0$ ), but they may be extensional equal on a subset of arguments (for example,  $f$  and  $g$  are equal on real numbers).
- **The lambda calculus is a theory of rules. These ‘rules’ are called  $\lambda$ -terms.**
- **In the “world”  $\Lambda$  of lambda calculus is only possible the life of these lambda terms. No other kind of life is allowed! Then the extension of a  $\lambda$ -term (i.e., its input-output behavior) is a set of pairs in  $\Lambda \times \Lambda$ :**

**Argument (input) = Result (output) = Function.**

# Lambda calculus: introduction

---

- **Every  $\lambda$ -term can be thought as a function(al program). It may get another function(al program) as input and it may give another function(al program) as output.**
- **No needness of codify: in the usual machines the data, the programs and the states of the machine have different structure. Therefore, data in input must be codified in the initial state, data in output must be obtained from the final state, and programs are only a part of the state. In lambda calculus**  
$$\text{data} = \text{program} = \text{state}.$$
- **No partiality: every  $\lambda$ -term can be applied to any other  $\lambda$ -term.**
- **Untyped versus typed: every lambda term has no domain and codomain against the usual concept of function in mathematics. So, we have the possibility of applying a lambda term to itself.**
- **There exists a typed version of lambda calculus.**

# Lambda calculus: introduction

---

- The way as a new function is defined starting from an expression.
- Mathematics:
  - Given an expression dependent on  $x$ , for example  $2(x+x)$ , we introduce a new symbol  $f$  defined as follows:  $f(x) = 2(x+x)$ .
- Lambda calculus style:
  - The function  $f$  of  $x$  can be represented as a  $\lambda$ -expression:  $\lambda x.2(x+x)$ .

The calculus is by substitution

$$(\lambda x.2(x+x))3 \rightarrow 2(3+3) \rightarrow 12$$

The lambda expression has the advantage that it contains all the informations about the function, while the symbol  $f$  has no immediate meaning (without its definition).

- Given an expression dependent on  $x$  and  $y$ , for example  $2(x+y)$ , we define the corresponding function of two variables as follows:

$$\text{either } f(x,y) = 2(x+y) \text{ or } \lambda x.\lambda y.2(x+y).$$

# Lambda calculus: introduction

---

- **The set  $\Lambda$  of lambda terms**
  - **The building blocks are the variables  $x, y, z, \dots$** 
    - **A variable  $x$  represents a generic function.  $x$  does not have an internal structure (it is an atom!) and it may be or become any other lambda term during a computation.**
  - **If  $M$  and  $N$  are lambda terms, then  $(MN)$  is a lambda term.**
    - **$(MN)$  represents the result of applying the function (program)  $M$  to the argument (input)  $N$ .**
    - **Notation:  $MN \dots P$  stands for  $(\dots(MN)\dots P)$**
  - **If  $M$  is a lambda term then  $(\lambda x.M)$  is a lambda term.**
  - **$(\lambda x.M)$  can be viewed as a function of  $x$  or as a “subprogram” with formal parameter  $x$ . All the occurrences of  $x$  in  $(\lambda x.M)$  are occurrences of the formal parameter  $x$ . They are called bound occurrences of  $x$ .**
  - **Notation:  $(\lambda xyz.M)$  stands for  $(\lambda x.(\lambda y.(\lambda z.M)))$**

# Lambda calculus: introduction

- **Examples of closed lambda terms (where every occurrence of a variable is bound):**
  - $I \equiv \lambda x.x$ ;  $K \equiv \lambda xy.x$
  - $S \equiv \lambda xyz.xz(yz)$
  - $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$
  - $\Omega_3 \equiv (\lambda x.xxx)(\lambda x.xxx)$
- **The way we calculate  $(\lambda x.M)N$ . The argument  $N$  is the actual parameter; when we make a call  $(\lambda x.M)N$  of the subprogram  $(\lambda x.M)$  with the actual parameter  $N$ , then the result of the computation is the term  $M[x:=N]$  obtained by substituting the actual parameter  $N$  for  $x$  within  $M$ . Every subterm  $(\lambda x.M)N$  is called a redex.**
  - $(\lambda x.x)z \rightarrow_{\beta} z$  (we substitute  $z$  for  $x$  in  $x$ )
  - $(\lambda xy.x)zu \rightarrow_{\beta} (\lambda y.z)u$  [we substitute  $z$  for  $x$  in  $\lambda y.x$ ]  $\rightarrow_{\beta} z$  [we subst.  $u$  for  $y$  in  $z$ ]  
 $(\lambda xy.x)$  is a function of two variables ( the projection on the first argument).
  - $(\lambda x.xx)(\lambda x.xx) \rightarrow_{\beta} (\lambda x.xx)(\lambda x.xx)$  [we substitute  $(\lambda x.xx)$  for  $x$  in  $(xx)$ ]. The path of reduction is infinite.
  - The double arrow “ $\rightarrow_{\beta}$ ” represents zero, one or more steps of reduction “ $\rightarrow_{\beta}$ ”

# Lambda calculus and computability theory

- The lambda calculus a universal computational formalism
- True and False: The terms  $T \equiv \lambda xy.x$  and  $F \equiv \lambda xy.y$  define respectively the truth value “true” and the truth value “false”, because  $Txy \rightarrow_{\beta} x$  and  $Fxy \rightarrow_{\beta} y$
- Pairs:  $[M,N] \equiv \lambda z.zMN$ 
  - $[M,N]T \rightarrow_{\beta} M$
  - $[M,N]F \rightarrow_{\beta} N$
- if\_then\_else:
  - $TMN \equiv (\lambda yz.y)MN \rightarrow_{\beta} M$ ;  $FMN \equiv (\lambda yz.z)MN \rightarrow_{\beta} N$
- Numbers:
  - $\underline{0} \equiv I$ ;  $\underline{n+1} \equiv [F,\underline{n}]$
- Test:  $\underline{Zero?} \equiv \lambda z.zT$ 
  - $\underline{Zero?} \underline{0} \equiv (\lambda z.zT)\underline{0} \rightarrow_{\beta} \underline{0}T \rightarrow_{\beta} T$
  - $\underline{Zero?} \underline{n+1} \equiv (\lambda z.zT)\underline{n+1} \rightarrow_{\beta} \underline{n+1}T \equiv [F,\underline{n}]T \equiv (\lambda z.zFn)T \rightarrow_{\beta} TF\underline{n} \rightarrow_{\beta} F$
- Predecessor and successor:
  - $\underline{Pred} \equiv \lambda x.(\underline{Zero?} x)\underline{0}(xF)$ ;  $\underline{Succ} \equiv \lambda x.[F,x]$

# Lambda calculus and computability theory

---

- **The addition is recursive!**
  - **Mathematical Definition:**
    - $x+0 = x$
    - $x+(y+1) = (x+y) +1$
  - **Mathematical Definition in a lambda calculus style**
    - $+xy = (\underline{\text{Zero?}}\ y)x (\underline{\text{Succ}}(+x(\underline{\text{Pred}}\ y)))$
- **Recursive definition with unknown +**
  - $+ = \lambda xy.(\underline{\text{Zero?}}\ y)x (\underline{\text{Succ}}(+x(\underline{\text{Pred}}\ y)))$
  - $= (\lambda u.\lambda xy.(\underline{\text{Zero?}}\ y)x (\underline{\text{Succ}}(ux(\underline{\text{Pred}}\ y))))+$
  - **Consider the term:**
    - $H = \lambda u.\lambda xy.(\underline{\text{Zero?}}\ y)x (\underline{\text{Succ}}(ux(\underline{\text{Pred}}\ y)))$
    - **Then the addition + is a fixpoint of H (if such a fixpoint exists), i.e.,  $H+ = +$**
- **To solve recursive equations is equivalent to find fixpoints of abstraction terms.  
Which lambda terms admit fixpoints in lambda calculus?**

# A fixpoint theorem

- **Theorem.** Every lambda term  $M$  admits a fixpoint which can be obtained in a uniform way by a fixpoint “combinator”  $\Theta$  satisfying

$$\Theta M \rightarrow_{\beta} M(\Theta M).$$

**Proof.** For every lambda term  $C$ , it is simple to find a lambda term  $B$  such that

$$BB \rightarrow_{\beta} C(BB). \quad (*)$$

Define  $B \equiv \lambda x.C(xx)$ , where  $x$  is a fresh variable. Then we have

$$BB \equiv (\lambda x.C(xx))(\lambda x.C(xx)) \rightarrow_{\beta} C((\lambda x.C(xx))(\lambda x.C(xx))) \equiv C(BB).$$

For example, if  $C \equiv yz$  then  $B \equiv \lambda x.yz(xx)$ , so that

$$BB \equiv (\lambda x.yz(xx))(\lambda x.yz(xx)) \rightarrow_{\beta} yz((\lambda x.yz(xx))(\lambda x.yz(xx))) \equiv yz(BB).$$

If the term  $C \equiv \lambda z.P$  is an abstraction term, then

$$\lambda x.C(xx) \equiv \lambda x.(\lambda z.P)(xx) \rightarrow_{\beta} \lambda x.P[z:=(xx)]$$

Then, if  $C \equiv \lambda z.P$  is an abstraction term, we can directly define  $B \equiv \lambda x.P[z:=(xx)]$ .

For example, if  $C \equiv \lambda z.\lambda y.yz$ , then  $B \equiv \lambda x.(\lambda y.yz)[z:=(xx)] \equiv \lambda xy.y(xx)$ , so that

$$BB \rightarrow_{\beta} (\lambda xy.y(xx))(\lambda xy.y(xx)) \rightarrow_{\beta} \lambda y.y((\lambda xy.y(xx))(\lambda xy.y(xx))) \equiv \lambda y.y(BB).$$

# A fixpoint theorem

We now define two different fixpoint combinators.

**Curry Fixpoint Combinator Y:** We put  $C \equiv f$  in the above recursive equation (\*), where  $f$  is an arbitrary variable. Then  $B \equiv \lambda x.f(xx)$  and  $BB \rightarrow_{\beta} f(BB)$ . The fixpoint  $BB$  of  $f$  is not obtained in a uniform way! If we consider  $Y \equiv \lambda f.BB$ , we have:  $(\lambda f.BB)f \rightarrow_{\beta} BB \rightarrow_{\beta} f(BB)$ . We have the problem here that  $f(BB)$  does not reduce to  $f((\lambda f.BB)f)$ . It is true the opposite. In conclusion we have:

$$(\lambda f.BB)f \rightarrow_{\beta} BB \rightarrow_{\beta} f(BB) \leftarrow_{\beta} f((\lambda f.BB)f).$$

**Turing Fixpoint Combinator Θ:** We would like to define  $\Theta$  in such a way that

$$\Theta x \rightarrow_{\beta} x(\Theta x).$$

This means that the lambda term  $\Theta$  should satisfy the recursive equation

$$\Theta x \rightarrow_{\beta} x(\Theta x),$$

that it is equivalent to

$$\Theta = \lambda x.x(\Theta x),$$

that can be written as follows:

$$\Theta = (\lambda yx.x(yx))\Theta.$$

# A fixpoint theorem

We apply the technique of the above slide to the recursive equation

$$BB = (\lambda yx.x(yx))BB.$$

Since  $(\lambda yx.x(yx))$  is an abstraction term, we define  $B \equiv \lambda zx.x(zzx)$ .

$\Theta$  is  $(\lambda zx.x(zzx))(\lambda zx.x(zzx))$ .

- **Exercise:** Show that  $YI = \Omega: (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))I \rightarrow_{\beta} (\lambda x.I(xx))(\lambda x.I(xx)) \rightarrow_{\beta} (\lambda x.xx)(\lambda x.I(xx)) \rightarrow_{\beta} (\lambda x.xx)(\lambda x.xx) \equiv \Omega$ .
- **Example:**  
 $fxy = fyx f$   
 $f = \lambda xy.fyx f$   
 $f = (\lambda fxy.fyx f)f$   
 $f = \Theta(\lambda fxy.fyx f)$
- **Denotational Semantics of programming languages.**

# Lambda calculus and computability theory

- Recall that  $H = \lambda u. \lambda xy. (\text{Zero? } y)x (\text{Succ}(ux(\text{Pred } y)))$
- Define the addition  $+ = \Theta H$ , so that  $H(\Theta H) = \Theta H$ . We calculate!
- $+ \underline{2} \underline{0} = \Theta H \underline{2} \underline{0}$ 
  - $\rightarrow_{\beta} H(\Theta H) \underline{2} \underline{0}$
  - $\rightarrow_{\beta} [\lambda xy. (\text{Zero? } y)x (\text{Succ}(\Theta H x (\text{Pred } y)))] \underline{2} \underline{0}$
  - $\rightarrow_{\beta} [\lambda y. (\text{Zero? } y) \underline{2} (\text{Succ}(\Theta H \underline{2} (\text{Pred } y)))] \underline{0}$
  - $\rightarrow_{\beta} (\text{Zero? } \underline{0}) \underline{2} (\text{Succ}(\Theta H \underline{2} (\text{Pred } \underline{0})))$
  - $\rightarrow_{\beta} \text{T} \underline{2} (\text{Succ}(\Theta H \underline{2} (\text{Pred } \underline{0})))$
  - $\rightarrow_{\beta} \underline{2}$
- $+ \underline{2} \underline{1} = \Theta H \underline{2} \underline{1}$ 
  - $\rightarrow_{\beta} (\text{Zero? } \underline{1}) \underline{2} (\text{Succ}(\Theta H \underline{2} (\text{Pred } \underline{1})))$
  - $\rightarrow_{\beta} \text{F} \underline{2} (\text{Succ}(\Theta H \underline{2} (\text{Pred } \underline{1})))$
  - $\rightarrow_{\beta} \text{Succ}(\Theta H \underline{2} (\text{Pred } \underline{1}))$
  - $\rightarrow_{\beta} \text{Succ}(\Theta H \underline{2} \underline{0})$
  - $\rightarrow_{\beta} \text{Succ } \underline{2} \rightarrow_{\beta} \underline{3}$

# Lambda calculus and computability theory

---

- **Primitive Recursion**
  - **f is defined by primitive recursion from g and h:**
    - $f(x,0) = g(x)$
    - $f(x,y+1) = h(x,y, f(x,y))$
  - **Example: if  $g(x) = x$  and  $h(x,y,z) = z+1$ , then f is the addition.**
  - **Mathematical Definition in lambda calculus style:**
    - $\underline{f}xy = (\underline{\text{Zero?}}\ y)(gx)(hx(\underline{\text{Pred}}\ y)(\underline{f}x(\underline{\text{Pred}}\ y)))$
  - **Mathematical Definition as fixpoint:**
    - $\underline{f} = \lambda xy. (\underline{\text{Zero?}}\ y)(gx)(hx(\underline{\text{Pred}}\ y)(\underline{f}x(\underline{\text{Pred}}\ y)))$
  - **Let  $U \equiv \lambda uxy. (\underline{\text{Zero?}}\ y)(gx)(hx(\underline{\text{Pred}}\ y)(ux(\underline{\text{Pred}}\ y)))$ . Then the function f defined by primitive recursion is a fixpoint of U, that is  $f = \Theta U$ .**
  - **U is a term where the variables g and h are FREE.**

# Lambda calculus and computability theory

---

- **The Schema of Primitive Recursion (PR)**
  - Let PR(g,h) be the function defined by primitive recursion from g(x) and h(x,y,z)
  - **Mathematical Definition in lambda calculus style of the schema PR:**
  - PRghxy = (Zero? y)(gx)(hx(Pred y)(PRghx(Pred y)))
  - **Mathematical Definition as fixpoint:**
    - PR =  $\lambda ghxy.(\underline{\text{Zero?}}\ y)(gx)(hx(\underline{\text{Pred}}\ y)(\underline{\text{PR}}ghx(\underline{\text{Pred}}\ y)))$
- Let  $U \equiv \lambda u.\lambda ghxy.(\underline{\text{Zero?}}\ y)(gx)(hx(\underline{\text{Pred}}\ y)(ughx(\underline{\text{Pred}}\ y)))$ . Then PR is a fixpoint of U, that is PR =  $\Theta U$ .
  - If  $g \equiv I$  ed  $h \equiv \lambda xyz.\underline{\text{Succ}}\ z$ , PRgh is the addition + defined by primitive recursion
  - If  $g \equiv \lambda x.0$  ed  $h \equiv \lambda xyz.+xz$ , PRgh is the product defined by primitive recursion
- Hence, PR represent the schema of primitive recursion.

# Lambda calculus and computability theory

---

- The schema of composition “;”
  - Mathematical Definition:  $f;g(x) = g(f(x))$
  - Lambda term:  $;\equiv \lambda fgx.g(fx)$
- The schema of exponentiation (definite iteration) “exp”
  - Mathematical Definition:  $\text{exp}(f)(x,y) = f(f(\dots f(x)\dots))$  y times
  - This schema corresponds to the “for” of imperative programming languages
    - Definition in lambda calculus style:  $\text{exp}fxy = (\text{Zero?}y)x(\text{exp}f(fx)(\text{Pred } y))$
    - Fixpoint:  $\text{exp} = \lambda fxy.(\text{Zero?}y)x(\text{exp}f(fx)(\text{Pred } y))$
  - Let  $U \equiv \lambda u.\lambda fxy.(\text{Zero?}y)x(uf(fx)(\text{Pred } y))$ . Then the schema exp is a fixpoint of U.
  - Lambda term:  $\text{exp} \equiv \Theta(\lambda u.\lambda fxy.(\text{Zero?}y)x(uf(fx)(\text{Pred } y)))$

# Lambda calculus and computability theory

---

- **The Schema of indefinite iteration “W”**
  - The schema “W” corresponds to the construct “while” of the imperative programming languages.
  - **Mathematical Definition:**  
 $f^{g=0}(x) = f^k(x)$  if  $k$  is the least natural number such that:
    - $g(f^i(x))$  is defined for every  $0 \leq i \leq k$ ;
    - $g(f^i(x))$  is not equal to 0 for every  $0 \leq i < k$ , while  $g(f^k(x)) = 0$ ; $f^{g=0}(x) = \text{undefined}$  if the above condition is not satisfied.
  - The schema W is a fixpoint of the following lambda term:
    - $U \equiv \lambda u. \lambda g f x. (\underline{\text{Zero?}}(g x)) x (u g f (f x))$
- $W \equiv \Theta(\lambda u. \lambda g f x. (\underline{\text{Zero?}}(g x)) x (u g f (f x)))$

# Some Problems

---

- **Non determinism**
  - Recall that a redex in a lambda term  $P$  is any subterm of  $P$  having the form  $(\lambda x.M)N$ . A lambda term may have contemporaneously more than one redex. It follows that the path of reduction of the redexes is non deterministic.
  - The value (result) of a computation is a lambda term without redexes (a normal form). Is it true that the order of reduction of the redexes does not affect the final result of the computation? The answer is yes (see below).
- **The problem of the bound variables (or formal parameters)**
  - The lambda term  $K \equiv \lambda xy.x$  is the projection on the second component . But we have
    - A wrong reduction:  $(\lambda xy.x)yz \rightarrow_{\beta} (\lambda y.y)z \rightarrow_{\beta} z$
    - We must change the name of the formal parameter  $\lambda y....$  before making the substitution. The first argument  $y$  represents a generic argument and it has no relationship with the formal parameter with the same name  $y$ . The problem is that the operator  $\lambda y$  is a binder!
    - The correct reduction:  $(\lambda xy.x)yz = (\lambda xu.x)yz \rightarrow_{\beta} (\lambda u.y)z \rightarrow_{\beta} y$

# Beta Reduction

---

- We assume that the set of variables of lambda calculus is well ordered.
- **Definition.** The term  $M[x:=N]$ , obtained by substituting  $N$  for  $x$  in  $M$ , is defined by induction over the complexity of  $M$ :
  - $x[x:=N] = N$
  - $y[x:=N] = y$
  - $(PQ)[x:=N] = P[x:=N]Q[x:=N]$
  - $(\lambda x.P)[x:=N] = \lambda x.P$
  - $(\lambda y.P)[x:=N] = \lambda y.P[x:=N]$  if  $y$  does not occur free in  $N$
  - $(\lambda y.P)[x:=N] = \lambda z.P[y:=z][x:=N]$  if  $y$  occurs free in  $N$  ( $z$  is the first variable that does not occur free in  $N$  and  $P$ )
- Two lambda terms  $M$  and  $N$  will be not distinguished if they are equal after a renaming of bound variables, that is, if  $M = N$  with respect to the least compatible equivalence relation ( $\alpha$ -conversion) including  $\lambda x.M =_{\alpha} \lambda y.M[x:=y]$ , where  $y$  is not free in  $M$ . For example,  $\lambda x.x =_{\alpha} \lambda y.y$ .

# Beta Reduction

---

- Consider the following binary relation on lambda terms:
  - $\beta = \{ ( (\lambda x.M)N , M[x:=N] ) \mid M,N \text{ are lambda terms} \}$
- Then  $\beta$  induces the one-step  $\beta$ -reduction  $\rightarrow_{\beta}$  (i.e., the compatible closure of  $\beta$ ) inductively defined as follows:
  - $(M,N) \text{ in } \beta \implies M \rightarrow_{\beta} N$ ;
  - $M \rightarrow_{\beta} N \implies MP \rightarrow_{\beta} NP$ ;
  - $M \rightarrow_{\beta} N \implies PM \rightarrow_{\beta} PN$ ;
  - $M \rightarrow_{\beta} N \implies \lambda x.M \rightarrow_{\beta} \lambda x.N$ .
- The relation  $\twoheadrightarrow_{\beta}$  is the reflexive and transitive closure of  $\rightarrow_{\beta}$ .